Enhanced Hot Spot Detection Heuristics for Embedded Java Just-in-Time Compilers*

Seong-Won Lee Soo-Mook Moon Seong-Moo Kim

School of Electrical Engineering
Seoul National University
{swlee, smoon, bluable}@altair.snu.ac.kr

Abstract

Most Java just-in-time compilers (JITC) try to compile only hot methods since the compilation overhead is part of the running time. This requires precise and efficient *hot spot detection*, which includes distinguishing hot methods from cold methods, detecting them as early as possible, and paying a small runtime overhead for detection. A hot method could be identified by measuring its running time during interpretation since a long-running method is likely to be a hot method. However, precise measurement of the running time during execution is too expensive, especially in embedded systems, so many counter-based heuristics have been proposed to estimate it. The *Simple* heuristic counts only method invocations without any consideration of loops [1], while Sun's *HotSpot* heuristic counts loop iterations as well, but does not consider loop sizes or method sizes [2,14]. The static analysis heuristic estimates the running time of a method by statically analyzing loops or heavy-cost bytecodes but does not measure their dynamic counts [3]. Although the overhead of these heuristics is low, they do not estimate the running time precisely, which may lead to imprecise hot spot detection.

This paper proposes a new hot spot detection heuristic which can estimate the running time more precisely than others with a relatively low overhead. It dynamically counts only important bytecodes interpreted, but with a simple arithmetic calculation it can obtain the precise count of *all* interpreted bytecodes. We also propose employing a static analysis technique to predict those hot methods which spend a huge execution time once invoked. This static prediction can allow compiling these methods at their first-invocation, complementing the proposed dynamic estimation technique. We implemented both, which led to a performance benefit of 10% compared to the *HotSpot* heuristic.

Categories and Subject Descriptors D.3.4 [**Programming Languages**]: Processors – *optimization, runtime environments, code generation*

General Terms Performance, Experimentation, Languages.

*This work was supported in part by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD) (KRF-2007-311-D00628).

LCTES'08 June 12-13, 2008, Tucson, Arizona, USA.

Keywords Java just-in-time compilation, hot spot detection, Sun's HotSpot heuristic, Java virtual machine, J2ME CDC

1. Introduction

Java has been popularly employed as a standard software platform for embedded systems, due to its support for platform independence, security, and faster development of reliable software contents [4]. Platform independence is achieved by installing the Java virtual machine (JVM) on each platform, which executes Java's compiled executable called *bytecode* via *interpretation* [5]. Since this software-based execution is much slower than hardware-based execution, compilation techniques that translate the bytecode into machine code have been employed, such as *just-in-time compilers* (JITC) [6]. JITC performs the translation during runtime, often on a method-by-method basis.

Since the translation overhead is part of the running time, most JITCs employ *adaptive compilation*, where a method is interpreted initially, and then is compiled only when it is found to be hot [13]. This requires precise and efficient *hot spot detection*. Generally, hot spot detection in the middle of execution is a difficult problem. A method detected as a hot spot can easily become a cold spot since we cannot know its future behavior. Also, hot spots should be detected early enough because even a long-running method cannot lead to a performance improvement if detected and compiled too late, while a short-running method can be a hot spot if it is compiled early enough. Moreover, the overhead spent for hot spot detection is part of the running time, so we cannot use an elaborate technique that takes too much time.

Many heuristics have been proposed for hot spot detection and all of them share a common wisdom, which can be stated informally as follows: *a long-running method is likely to be a hot spot*. That is, a method that has been running long so far is likely to be running long in the future, so its compilation is likely to lead to a performance benefit that can offset its compilation overhead. In order to determine if a method has been running long enough, we need some information on the running time of the method. The difference among heuristics is how to obtain such information, how precise it is, and how much overhead is needed to get it.

The most precise way of obtaining the running time of a method is simply measuring the real time difference between its entry and exit and accumulating the time whenever the method is executed. Unfortunately, such a timing function is costly to invoke and its frequent call due to short Java methods can cause a big overhead.

One popular technique is resorting to *sampling*, where the call stack is sampled in a regular interval using a separate thread [7]. Those methods frequently observed on the sampled call stack are regarded as running long. Sampling can identify hot methods on a hot call chain as a whole, and its overhead is relatively small. Unfortunately, sampling is not very effective on embedded systems since the timer interval in embedded CPU is too long.

Most techniques attempt to estimate the running time based on software *counters* such that they count some interpreted bytecodes at runtime. The estimated running time of a method is obtained with these counter values and if it is higher than a given threshold, the method is regarded as hot. Although the counter-based estimation is conceptually simpler than sampling and is more effective in identifying hot spots earlier (which is important in embedded systems where running time tends to be shorter than in servers), its counting overhead would directly increase the execution time. Therefore, most techniques try to reduce the overhead. Some techniques count only the method invocations [1] or loop iterations as well [2,14], and then estimate the runtime after multiplying some constant values. Some technique even attempt to analyze the runtime of a method statically [3]. Although these techniques are involved with a low overhead, they may lead to compilation of cold methods or delayed or failed compilation of hot methods, which can affect performance.

This paper proposes a novel counter-based runtime estimation technique called *flow-sensitive runtime estimation* (*FSRE*). FSRE is as precise as if we count *all* interpreted bytecodes following the execution control flow, yet is not involved with a serious overhead. The idea is counting only important bytecodes in a flow-sensitive manner and then estimating the total count of bytecodes with a simple arithmetic calculation. We also propose a *static-FSRE* which allows some hot methods to be compiled in the first-invocation via flow-sensitive static analysis of methods, complementing FSRE. We implemented FSRE in our JITC on a CDC RI JVM [10]. FSRE shows a tangible performance benefit compared to the original HotSpot heuristic. When we add the static-FSRE, the performance benefit gets even higher.

The rest of this paper is composed of as follows. Section 2 reviews previous approaches to hot spot detection. Section 3 introduces our FSRE technique, followed by our static-FSRE in Section 4. Section 5 describes how we merge the proposed two techniques as a hot spot detection heuristic. Section 6 shows our experimental results. A summary follows in Section 7.

2. Previous Approaches to Hot Spot Detection

This section reviews previous counter-based hot spot detection heuristics. Most heuristics are involved with an inequality for detecting hot methods, which often has the following form:

$$Threshold < T[m]$$

Here, T[*m*] can be interpreted as the estimated *future* running time of a method *m* such that if T[*m*] becomes higher than *Threshold*, we will compile *m*. The reason that the future running time is needed for hot spot detection is that if it is long enough, the benefit of JITC (i.e., the reduced future running time due to execution of compiled code) will also be high enough. Actually, if we interpret *Threshold* as the compilation cost, the inequality means that if the benefit of JITC is higher than the cost of JITC, it is better to compile. This is often called the cost-benefit model [8].

Most techniques estimate T[*m*] primarily based on *m*'s estimated past running time since a method that has been running long so far is likely to be running long in the future, or vice versa. While *Threshold* is a constant value obtained thru extensive tuning, T[*m*] really differentiates the approach each heuristic takes, so we focus on how each heuristic estimates T[*m*].

Generally, the quality of hot spot detection heuristics can be judged based on three features, as follows:

- *Preciseness*: we should detect as many hot methods as possible but should not misjudge cold methods as hot ones
- *Detection time*: for hot methods, we should detect and compile them as early as possible
- *Detection overhead*: the detection overhead should be small

We will review existing heuristics based on these as below.

2.1 Simple Heuristic

The idea of *Simple heuristic* is that a simpler one would be better for hot spot detection [1]. It measures only the method invocation count for estimating the future invocation count (a similar approach is used in [15]). It also uses the static method size for estimating the compilation overhead. Based on both, the Simple heuristic estimates the future running time. More precisely, the future running time of a method *m*, T[*m*], is estimated as follows:

T[*m*] = C1 * invocation count of *m* + C2 * method size of *m*

Although the detection overhead of the Simple heuristic would be minimal, it does not count any dynamic events within a method such as loops or branches, but primarily resorts to the invocation count. The *Threshold* proposed in [1] is a somewhat small constant, so it may compile many cold methods as well as hot methods, although hot methods are detected earlier, as will be seen in our experimental results in Section 6.

2.2 HotSpot Heuristic

The hot spot detection heuristic of Sun's HotSpot CDC JVM counts the backward branch as well as the method invocation, in order to reflect the running time of loops [2]. More precisely, the future running time of a method *m*, T[*m*], is estimated as follows[1]:

T[*m*] = C3 * invocation count of *m* + C4 * backward branch count in *m*

Counting backward branch is simple to implement since we can just add the instrumentation code at the switch-case statement of each branch bytecode in the interpreter loop. Although this allows HotSpot to consider loops, estimating the running time more precisely than the Simple heuristic, HotSpot completely ignores the size of a loop and a method, or any control flows within a method. Still, HotSpot is a commercially accepted heuristic with its well-tuned constants and *Threshold*, so we will take its preciseness and detection time as a standard to compare against.

[1] The original inequality of HotSpot has one more term added in its right, (C*transition invocation count of *m)*, where transition invocation means a method invocation from a JITC method to *m* (which is being interpreted) or from *m* to a JITC method [2]. In the HotSpot JITC, such a transition invocation takes additional overhead. There is no such an overhead in our JITC on CVM RI, so we omitted the term.

2.3 Static Analysis Heuristic

There is an approach that statically analyzes the runtime of a method [3]. The runtime for a single invocation of a method *m*, S[*m*], is analyzed in two ways. One is when compiling *m* to un-optimized machine code (inaccurate analysis) and the other is when compiling the un-optimized code into more optimized code (accurate analysis). In the former case of inaccurate analysis,

S[m] = method size of *m* + C5 * (number of big loops in *m*) + C6 * (number of small loops in *m*)

In the latter case of accurate analysis, more elaborate analysis is performed for identifying the control flow of basic blocks and the loop hierarchy based on back-edge list. S[*m*] is given as follows:

$$S[m] = \text{method size of } m + \sum_{\forall loop \in m} (C7 * loop \text{ size})$$

This heuristic gives different costs to different bytecodes, so the method or the loop size is not just the bytecode size but is a sum of the bytecode costs in it. In accurate analysis, the loop size is multiplied by C7 in a nested way if there is a loop hierarchy.

This heuristic can identify method sizes or loop sizes with heavy-cost bytecodes unlike in HotSpot. However, even the accurate analysis statically predicts the loop iteration count of every loop as a constant C7, which would lower the preciseness of hot spot detection. Moreover, computing the control flow of basic blocks or sorting the back-edge lists may cause a serious overhead.

The *Threshold* value used with S[*m*] in the heuristic inequality differs from methods to methods, and varies as the program is running [3]. It is not clear if there is any real benefit in exploiting the imprecise estimation S[*m*] in such a complicated manner.

3. Flow-Sensitive Runtime Estimation

Previous runtime estimation techniques oversimplify either loop iteration counts or loop/method sizes in order to reduce the estimation overhead. This can lead to imprecise hot spot detection. In this section, we introduce a new runtime estimation technique, called *flow-sensitive runtime estimation* (FSRE), which can improve the preciseness with a relatively small overhead.

The proposed technique attempts to obtain the precise count of *all* interpreted bytecodes. However, it does not actually count all bytecodes but "*important*" bytecodes only, and then calculate the total count based on them in a control-flow sensitive manner. There are two types of important bytecodes in FSRE.

The first type is *heavy-weight* bytecodes. We classify the Java bytecode instructions into simple bytecodes and heavy bytecodes. Simple bytecodes are those which take a short time to execute and are given a weight of their byte sizes (e.g., `iadd` and `ificmpgt` whose byte size is one and three have a weight of one and three, respectively). Heavy bytecodes are those whose execution takes a longer time than simple bytecodes such as method invocations or field accesses. They are given a weight of their byte sizes plus additional weight (e.g., `invokestatic` whose byte size is 3 and whose additional weight is 18 is a given a total weight of 21). The weight of a bytecode will be regarded as its running time in FSRE, which seems to be reasonable and simplifies our algorithm, as will be seen shortly. The classification of simple and heavy bytecode or the weight of each heavy bytecode is determined by measuring its real execution time on a given platform.

The other type is *control-flow* bytecodes such as branches or returns. For a branch bytecode, we need to know if it is a forward or a backward branch, and the *offset* of bytes that it jumps over. When we encounter branch bytecodes, we update the estimated running time of a method by adding or deleting the offset.

Now, we describe the FSRE algorithm. For a method *m*, T[*m*] is its estimated running time. We want T[*m*] to have the sum of the weights of all bytecodes executed so far. There are four events during the execution of the program, which can update T[*m*].

(1) Whenever the method *m* is invoked, T[*m*] is first incremented by its method size, available from *m*'s method block, which would be the sum of byte sizes for all static bytecodes in *m*:

T[*m*] += (method size of *m*)

If there are no important bytecodes in the method (no branch or no heavy bytecode), the method size will simply be the estimated running time of *m* for this invocation. However, if there are important bytecodes in *m*, T[*m*] will be rectified correctly when those important bytecodes are executed through (2)-(4) below.

(2) When a heavy bytecode is executed, T[*m*] is incremented by its additional weight (not including its byte size)

T[*m*] += (additional weight of the heavy bytecode).

By the time when this heavy bytecode is executed, its byte size should have already been added to T[*m*], so we just need to add its additional weight for more precise calculation of T[*m*].

(3) When a backward branch bytecode is executed, T[*m*] should be incremented because executing a backward branch means a new iteration of a loop being started. So, we add the byte size of the loop to T[*m*], which equals to the branch offset plus the byte size of the branch bytecode (since the offset is simply the difference of addresses between the branch and the target, it does not include the byte size of the branch itself)

T[*m*] += (offset + byte size of the backward branch)

On the other hand, when a forward branch is executed, T[*m*] should be decreased because T[*m*] already includes the byte sizes of all bytecodes between the branch and the branch target, which equals to the branch offset minus the byte size of the branch (since the offset already includes the byte size of the branch)

T[*m*] -= (offset - byte size of the forward branch)

(4) When a return bytecode is executed but if it is not the last bytecode of the method, we need to decrease T[*m*] by the byte sizes of all bytecodes between the return and the last bytecode of *m* since T[*m*] already includes these due to the step (1)

T[*m*] -= (method size – (address of return bytecode - start address of *m* + byte size of return))

The above description indicates that FSRE computes T[*m*] in a more flow-sensitive manner than the HotSpot or the Simple heuristics by updating T[*m*] following the execution control flow. It does not build a control flow graph or a back-edge list, though, unlike the Static analysis heuristic.

Figure 1 illustrates FSRE for the following example method *m*:

```
public static void m() {
    int a = 0;
    for( int i1 = 0; i1 < 2; i1++ ) {
       boo();
       for( int k = 0; k < 2; k++ ) a++;
       if ( a > 2 ) return;
    }
    return;
}
```

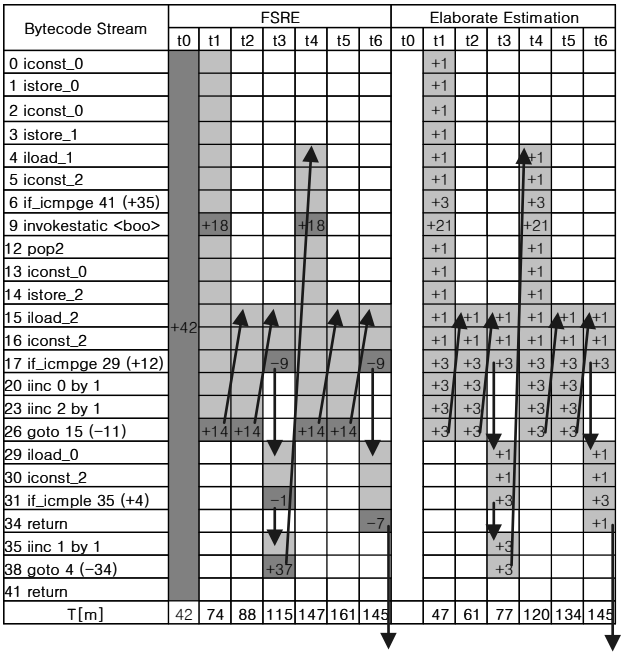

Figure 1. An example of FSRE and elaborate computation.

Each column in Figure 1 shows the updated trace of T[*m*] for each iteration of the inner loop or the outer loop. The left columns of Figure 1 illustrate FSRE. When the method is invoked, T[*m*] is initialized by m()'s size which is 42 bytes. When we meet a call to `boo()` in the first iteration of the outer loop, T[*m*] is incremented by its additional weight, 18. When the backward branch of the inner loop is taken, the offset (11) plus the byte size of the branch (3) is added to T[*m*]. When the inner loop exits at the forward branch, the offset (12) minus the byte size of the branch (3) is subtracted from T[*m*]. This process continues until the return bytecode is executed, when T[*m*] of 145 is produced.

The right columns of Figure 1 illustrate when we update T[*m*] at *every* interpreted bytecode by adding its weight to T[*m*]. Although FSRE updates T[m] at only important bytecodes, it produces the same T[m] that this elaborate computation produces[1].

As to the FSRE overhead, the fraction of important bytecodes among all executed bytecodes is small (12% on average in our experiments), so the overhead would be small. Also, FSRE is easy to implement since we just add the counting code at the switch-case statement of each important bytecode in the interpreter loop.

[1] Actually, there can be some minor differences between the two T[*m*] if there are some null bytecodes padded by `javac`.

4. Static-FSRE for First-Invocation Compilation

Although FSRE can estimate the hitherto running time of a method precisely, it is an "after-the-fact" estimation in the sense that it estimates for those methods that have been interpreted at least once. This would be useful for detecting hot methods which are invoked frequently. However, there are also hot methods which are not invoked many times but take a long time to execute once invoked. An example would be a method which spends a long execution time due to huge loops, thus constituting a hot spot, but being invoked just a couple of times (or only once in an extreme case). In fact, our experiments do show such loops.

With FSRE, this type of a hot method can still be compiled at its second-invocation at the earliest, when it is found to be a hot spot after its first-invocation and interpretation with FSRE. If a technique called on-stack replacement (OSR) would be employed [9,16], we might be able to compile the method in the middle of interpretation in its first invocation and continue to execute the compiled method thereafter. However, OSR is relatively complex and its benefit is rather low [17]. Therefore, it might be desirable to compile and execute the method in its first-invocation after identifying the method as a hot spot somehow before execution.

In order to complement FSRE, we propose a *static*-FSRE which statically *predicts* the runtime of a method spent for its single invocation. Static-FSRE will be performed before executing a method at its first invocation and the result will be used for deciding if we should compile the method right away. The predicted runtime for a method *m* is denoted by P[*m*]. Unlike the original, dynamic FSRE, we cannot follow the execution control flow to compute P[*m*] since we are not executing the method *m*. Instead, we perform a single, sequential traversal of the bytecode stream of *m* to estimate P[*m*]. As we did with the original FSRE, we update P[*m*] only when we meet four types of important bytecodes, as follows:

(1) As with FSRE, we initialize P[*m*] with the bytecode size of *m*

P[*m*] = method size of *m*

(2) As to the branch, we cannot know if a branch will be taken or not, or how many times it will be taken. Since a precise analysis would be too costly, we simply predict that a backward branch is always taken for some constant number of times, while a forward branch is never taken. When we meet a backward branch during a sequential traversal, it usually means the end of a loop, so we simply multiply the branch offset by some predetermined constant C, which is then added to P[*m*]:

P[*m*] += C * (offset + byte size of the backward branch)

If there is a nested loop, we can detect this if the target address of a backward branch (corresponding to an outer loop) is earlier than the target address of a previously-visited backward branch (corresponding to an inner loop). If so, we add (outer_loop_offset – inner_loop_offset) and (inner_loop_offset + byte size of inner loop branch) * (C-1), then multiply this by C in order to get the correct time spent in the nested loop (we subtracted 1 in C-1 since a single running time of the inner loop was already added to P[*m*] at the inner loop branch). We add this product to P[*m*]:

P[*m*] += C * (outer_loop_offset – inner_loop_offset + (C-1) * (inner_loop_offset + byte size of the branch))

(3) As to the heavy bytecode, we simply add its additional weight to P[*m*] when it is met during the traversal. Since we cannot consider the execution control flow, its additional weight is added to P[*m*] only once unlike in FSRE:

P[*m*] += (additional weight of a heavy bytecode)

(4) Most methods have a single return bytecode at the end of their bytecode stream, but even if not, we ignore any intermediate returns and continue to the last bytecode of the stream.

If we apply static-FSRE to the example in Figure 1 with C=2 (this is for illustration of this example where both inner loop and outer loop iterate twice, but in our real implementation, we set C=32), we initialize P[m] by 42 and add 18 at `invokestatic`. Then we add 2*14 at the backward branch of the inner loop and 2 * (34 - 11 + 1 * 14) at the backward branch of the outer loop to P[*m*], whose final value will be 162, around 10% deviated from T[*m*].

The proposed static-FSRE cannot be as precise as the original FSRE, yet it can be obtained with a minimal overhead, and some important control flows such as nested loops are considered. We will exploit P[*m*] usefully by detecting some of the hot methods early and compiling them even without any interpretation.

5. Merged Heuristic of Dynamic and Static FSRE

Previous two sections described our proposed dynamic and static runtime estimation techniques. In this section we describe how to decide their thresholds, using that of the HotSpot heuristic. We also discuss how to merge and exploit them for hot spot detection.

5.1 Threshold of FSRE

In order to exploit the proposed dynamic and static FSRE as hot spot detection heuristics, we have to decide the threshold value to be placed in the left of their inequalities. Since the threshold value is dependent on the runtime estimation technique and is obtained with extensive tuning with it, we cannot directly use the threshold of existing techniques. In this paper, we propose threshold values for FSRE based on that of Sun's HotSpot heuristic because it is a well-tuned, widely used heuristic on a commercial JVM. The inequality of HotSpot heuristic for a method *m* is as follows:

T<C3* invocation count of *m*+C4* backward branch count in *m*

The constant T in the left of the inequality is the threshold while the right is HotSpot's estimated running time, T[*m*], described in Section 2.2. We want to replace the right of the inequality by our FSRE, T[*m*], introduced in Section 3. Now the question is what would be an appropriate constant value that can be placed in the left of the inequality. The original T is tuned for HotSpot, hence not directly applicable to FSRE[1]. However, we want to decide a new threshold using the well-tuned T value of HotSpot.

If we compare the T[*m*] of HotSpot and the T[*m*] of FSRE, we can find that only the invocation count is common in both T[*m*]'s. So we can consider a case where only the invocation count is used in both T[*m*]'s, which is when a method *m* is composed of simple bytecodes with no branches. In this case, the HotSpot inequality for the method *m* would be T < C3 * (invocation count of *m*), where the method *m* will be compiled when the invocation count is higher than T/C3. When the invocation count is T/C3 for this method, T[*m*] of FSRE, which is (invocation count) * (size of *m*), will have a value T/C3 * (size of *m*). So, if we compile the method *m* when T[*m*] of FSRE is higher than T/C3 * (size of *m*), both the FSRE heuristic and the HotSpot heuristic will compile the method after the same number of interpretations. Based on this simple reasoning, we decide the threshold as T/C3 * (size of *m*), so the inequality of the FSRE heuristic is:

$$T/C3 * (\text{size of } m) < T[m]$$

We can use the same threshold for the static-FSRE heuristic, but in this case P[*m*] is a predicted time for a single invocation, so it should be multiplied to some constant C8 before being compared to the threshold. The inequality of the static-FSRE heuristic is:

$$T/C3 * (\text{size of } m) < P[m] * C8$$

In the current implementation, we used five for C8

5.2 Merged Heuristic

We now describe a merged heuristic of both. We perform the static-FSRE for a method even before it is invoked (we can do this at loading time for every method in a class without any significant overhead) and decide if the method should be compiled when invoked for the first time using its inequality. If so, we compile and execute the method when it is actually called for the first time. Otherwise, we interpret the method based on FSRE from that point, and compile it when its inequality is satisfied. This process is depicted in Figure 2.

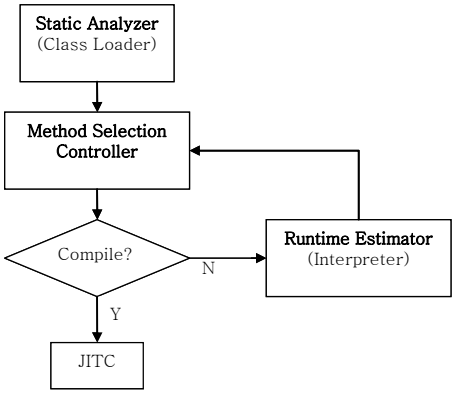

Figure 2. A merged FSRE heuristic.

6. Experimental Results

Previous section described our proposed FSRE heuristics. In this section, we evaluate them compared to HotSpot and others.

6.1 Experimental Environment

The experiments were performed with our JITC implemented on CVM RI version build 1.0.1_fcs-std-b12 [10]. Java methods are initially executed by the CVM interpreter until they are determined to be hot spots, and then are compiled into native code. Our JITC performs many traditional optimizations including method inlining. Our JITC passed most of the compatibility tests.

[1] When we actually experiment with the original T with FSRE, the performance is much worse (-18%) than the original HotSpot heuristic.

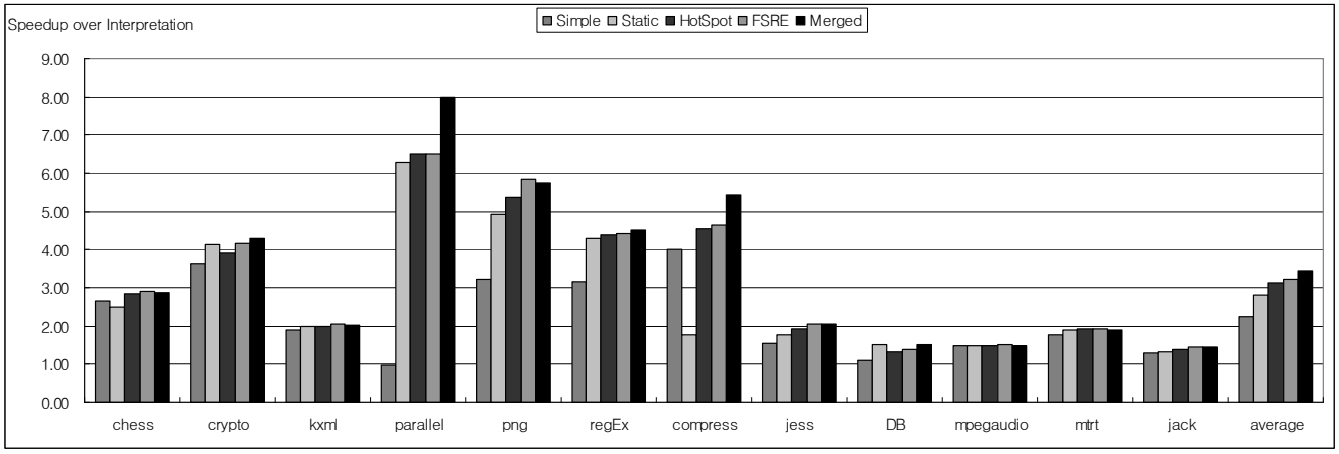

Figure 3. Performance ratio of the JITC performance with five heuristics compared to the interpreter performance.

Our CPU is a MIPS-based SoC called ATI Xilleon which is popularly employed in Digital TVs. The MIPS CPU model is 4Kc V0.7 with a clock speed of 300MHz. It has an I-cache of 16KB, a D-cache of 16KB, and a 128MB main memory. The OS is an embedded linux (kernel v2.4.18). The benchmarks we used are SPECjvm98 (except for javac) [11] and EEMBC [12] [1]. For performance evaluation, we measured 10 times for each benchmark, chose three numbers in the middle, and took their average. We did this because of severe fluctuations in some benchmarks (e.g., `regex` or `mtrt`), which would be due to the embedded environment whose performance is more sensitive and fluctuating than in the desktop environment.

6.2 Evaluation Heuristics

For evaluation of FSRE, we experimented with five heuristics: the Simple heuristic, the HotSpot heuristic, the Static analysis heuristic, the FSRE heuristic, and the merged-FSRE heuristic, which will be denoted by *Simple*, *HotSpot*, *Static*, *FSRE*, and *Merged*, respectively. The details of each heuristic are as follows:

(1) *Simple*, described in Section 2.1 has the following inequality:

12,000 < 150 * invocation count of *m* + 40 * method size of *m*

The original threshold was 6000, but it compiles too many methods, causing an overflow of the code cache. So we increased it, and 12,000 showed the best performance result. If the method size is more than 300 bytes, the inequality is satisfied even before any invocation, allowing its first-invocation compilation.

(2) *HotSpot*, described in Section 2.2 has the following inequality:

T<C3 * invocation count of *m*+C4 * backward branch count in *m*

Here, T, C3, and C4 equal to 20,000, 20, and 4, respectively (both the source code and the manual use these constant numbers [2]).

(3) *Static*, similar to the one in Section 2.3 has the inequality:

T/C3 * (size of *m*) < P[*m*] * invocation count of *m*

P[*m*] is a predicted running time of a single invocation of *m*. It is computed using our static-FSRE in Section 4, but is almost identical to S[*m*] of accurate analysis in Section 2.3. However, the threshold is fixed unlike in Static analysis, and is based on that of FSRE since it will be more consistent with other HotSpot-based heuristics. There is no first-invocation compilation in *Static*.

(4) *FSRE*, described in Section 5 means dynamic-FSRE only.

(5) *Merged* includes both the static-FSRE and dynamic-FSRE described in Section 5 so as to allow first-invocation compilation.

6.3 Performance of the Five Heuristics

Figure 3 shows the ratio of the JITC performance compared to the interpreter performance, when the JITC is equipped with each heuristic. Figure 4 is for comparing the performance ratio among all heuristics with the HotSpot performance as a basis of 100%.

On average, *Simple* shows the worst performance, and *Static* also shows a worse performance than *HotSpot*. Since both count no dynamic information other than method invocations, they appear to suffer from imprecise hot spot detection than *HotSpot*, which counts loop iterations in addition. And, multiplying a statically-predicted runtime to the invocation count in *Static* seems to be much better than multiplying a constant in *Simple*.

FSRE showed a better or equal performance than *HotSpot* in *all* benchmarks consistently, which would be due to more precise hot spot detection. The average benefit of *FSRE* over *HotSpot* is 3.4%.

Merged shows the best performance, an average of 9.9% better than *HotSpot*, yet there are some variations. *Merged* shows a tangibly better performance than *FSRE* in `parallel`, `compress`, and `DB`. We found that these benchmarks include hot methods that can benefit from first-invocation compilation by spending a long running when invoked. For example, `parallel` includes a method which takes 69% of the execution time but is invoked only 32 times. It is compiled after the first invocation in *FSRE* or *HotSpot*, but is compiled in the first invocation in *Merged*. There is also such a method in compress (22%, three times) and DB (7%, four times), and its earlier compilation leads to better performance.

On the other hand, `mpegaudio` also includes a hot (18%) method, compiled in the first invocation with *Merged*, yet it is called more than 32K times and compiled in the 22nd call in *FSRE*. So, its

[1] Any performance numbers for these benchmarks shown in this paper are relative numbers to demonstrate the value of our JITC, so they should **not** be interpreted as official scores.

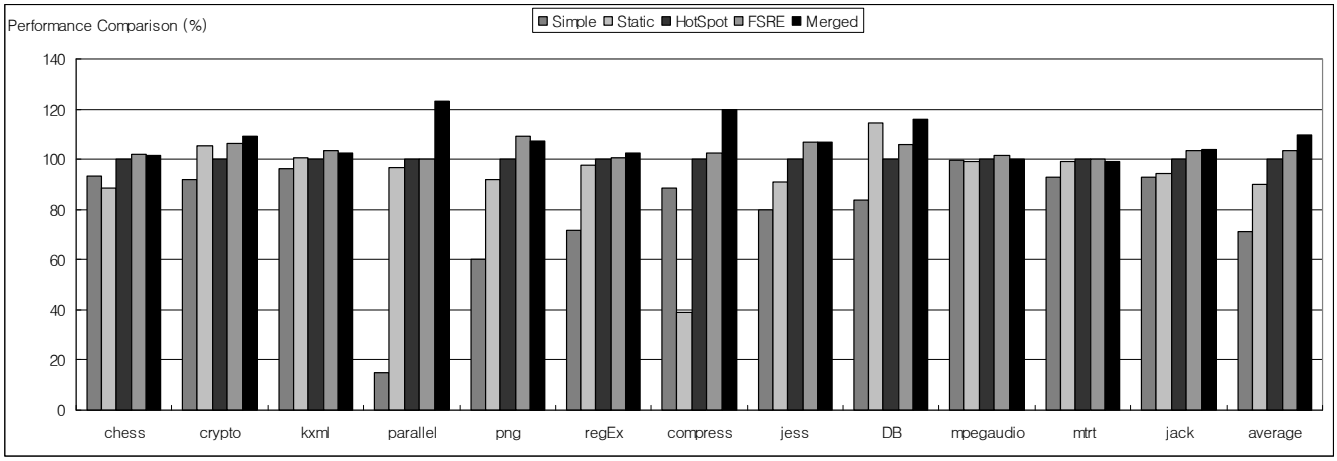

Figure 4. Performance ratio among heuristics with the HotSpot performance as a basis of 100%.

earlier compilation cannot affect the performance much. It should be noted that static-FSRE is not precise, so it can make cold methods be compiled mistakenly. For example, *Merged* compiles nine cold methods in `mpegaudio` in the first invocation and four of them are not compiled at all with *FSRE*. This leads to slight performance degradation, and similar degradation can be found in other benchmarks where cold spots are compiled only in *Merged*.

We now attempt to evaluate the five heuristics based on three features discussed in Section 2, which are the preciseness of hot spot detection, the detection time, and the detection overhead.

6.4 Preciseness of Hot Spot Detection

We first define *hot methods* in each benchmark as those whose execution time takes more than 0.05% of the total execution time. In order to identify such hot methods, we used the JProfiler and ran it in interpretation-only mode. Figure 5 shows the execution time coverage of all hot methods in each benchmark. On average, these hot methods cover 86.9% of the total execution time.

We then measured how many methods are compiled by each heuristic and how many of them are hot methods. In Figure 6, the top bar and the bottom bar show the ratio of compiled methods and compiled hot methods by each heuristic compared to all executed methods, respectively. On average, *HotSpot* compiles only 8.3% of executed methods, around half of which are hot methods. *FSRE, Merged,* and *Simple* compile slightly more methods than *HotSpot* in this order, but they compile almost the

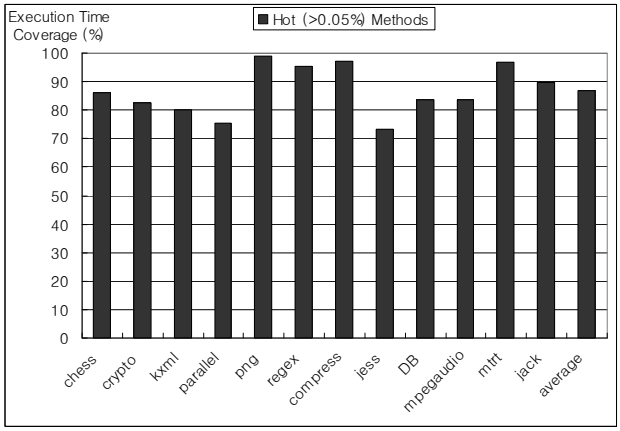

Figure 5. Execution time coverage of all hot methods.

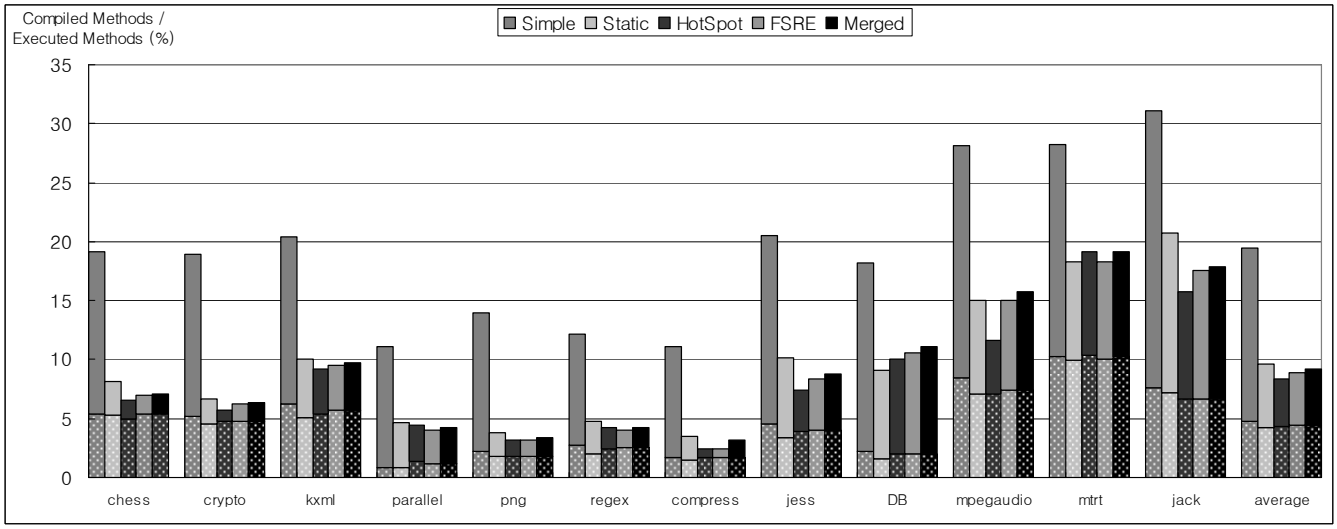

Figure 6. The ratio of compiled methods and compiled hot methods compared to all executed methods.

same number of hot methods. This indicates that there is no big difference in detection preciseness among these heuristics, in terms of the number of methods or hot methods compiled.

On the other hand, *Simple* compiles more than twice methods than *HotSpot*, yet its number of hot methods is similar. This means that *Simple* compiles many cold methods to compile equivalent hot method, which will increase the compilation overhead, though.

Table 1 compares the bytecode size of all cold methods compiled by each heuristic, with the HotSpot bytecode size as a basis of 100%. It shows that *Simple* requires compiling 50 times more cold bytecodes than HotSpot. Since the compilation overhead would increase proportional to the bytecode size, this would affect the performance of *Simple* seriously, as seen in Figure 4.

We also checked the quality of compiled hot methods. Figure 7 shows the weighted ratio of hot methods compiled by each heuristic to all hot methods where the weight is the execution percentage. It shows that *HotSpot* compile 97% of hot methods, while *FSRE* and *Merged* compile equal or slightly more hot methods, consistently in all benchmarks. This means that *FSRE* and *Merged* achieve equal or slightly better preciseness of hot spot detection compared to *HotSpot*.

On the other hand, the coverage of hot methods compiled by *Simple* and *Static* is significantly lower than HotSpot, although their number of compiled hot methods was similar to HotSpot in Figure 6. This means that they miss compiling *important* hot methods, achieving less precise hot spot detection. One extreme case is `parallel` of *Simple* where only 5% of hot methods are compiled. This is due to a failure of compiling the hottest method in `parallel` which takes 69% of the execution time. This explains its extraordinarily low performance in Figure 4.

One thing to note from Table 1 and Figure 7 is `compress` of *Simple* and *Static*. The cold bytecode size of both is much higher than in other benchmarks, so they should suffer from high compilation overhead. On the other hand, *Simple* compiles 100% of hot methods, while *Static* compiles only 75%. We believe this is the reason that *Static* achieves a seriously worse performance than *Simple* in `compress` in Figure 4 unlike in other benchmarks. That is, the high compilation overhead in *Simple* is offset by executing compiled hot methods instead of interpreting them. This can also be observed through hot spot detection time below.

Table 1. Comparing bytecode size of all cold methods compiled.

	Simple	Static	HotSpot	FSRE	Merged
`chess`	5404.2	414.0	100.0	82.3	85.7
`crypto`	2865.7	109.7	100.0	132.5	133.7
`kxml`	1704.4	202.3	100.0	127.2	128.5
`parallel`	3836.3	147.1	100.0	94.4	95.6
`png`	2693.2	240.0	100.0	99.7	101.7
`regex`	3974.3	567.4	100.0	97.5	100.2
`compress`	34012.3	2450.8	100.0	100.0	213.8
`jess`	1651.3	308.2	100.0	124.2	148.2
`DB`	1121.2	164.5	100.0	99.8	103.4
`mpegaudio`	979.2	147.1	100.0	111.5	113.5
`mtrt`	405.0	70.4	100.0	92.1	93.1
`jack`	1281.4	269.2	100.0	156.8	158.8
average	4994.0	424.2	100.0	109.8	123.0

6.5 Hot Spot Detection Time

Another important requirement of hot spot detection is that hot methods should be detected and compiled as early as possible. In order to evaluate the detection time of hot methods, we should measure the time spent for interpretation before being compiled. Unfortunately, we could not measure such a repeated, short period of interpretation time precisely in our embedded MIPS board. Instead, we estimate this follows. For each hot method we first calculate the ratio of its invocation count before being compiled to the total invocation count. Then, we run each benchmark in interpretation-only mode where we measure the total interpretation time for each hot method, relatively more precisely. We multiply the ratio and the total interpretation time, and it gives an estimated interpretation time of the method before being compiled, which we call the *detection time*.

As we saw in Figure 7, the set of hot methods compiled by each heuristic differs somewhat. So, we sum up the detection time for a

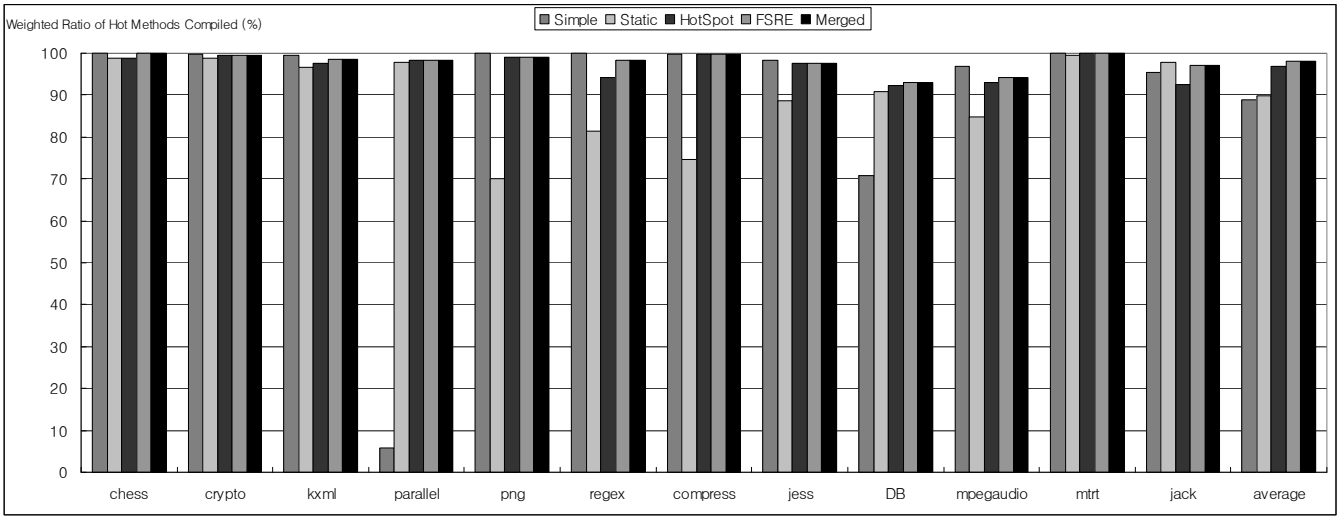

Figure 7. The ratio of hot methods compiled by each heuristic to all hot methods

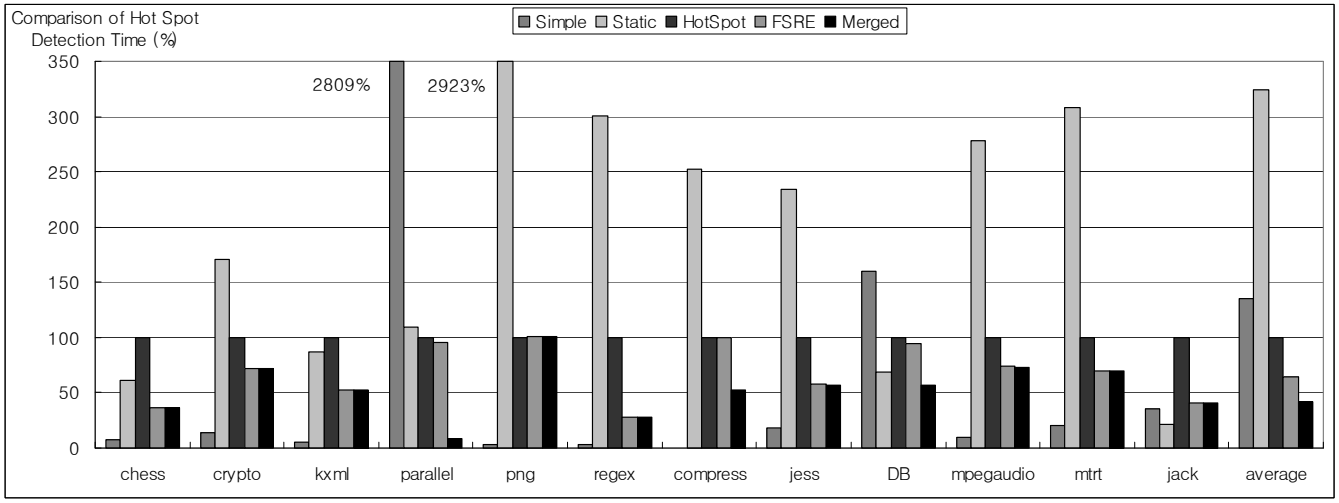

Figure 8. Hot spot detection time of heuristics compared to the HotSpot detection time as a basis of 100%.

union of methods compiled by each heuristic. If a hot method is not compiled at all by some heuristic, then its total interpretation time will be added naturally. Then, the sum of detection time for each benchmark can be regarded as the *hot spot detection time*.

Figure 8 shows the ratio of the hot spot detection time of each heuristic compared to the *HotSpot* detection time as a basis of 100%. In most benchmarks, *FSRE* has a much shorter hot spot detection time than *HotSpot*, which would be the main reason for its performance advantage (i.e., more tangible than preciseness of hot spot detection in Section 6.4). And *Merged* spends even less detection time due to its first-invocation compilation in `DB`, `parallel`, and `compress`, consistent with our observation in Section 6.3. On average, *FSRE* and *Merged* spends 36% and 59% less time for hot spot detection than *HotSpot*, respectively.

Comparing *Static* to *HotSpot*, *Static* spends less detection time for some benchmarks while it spends much more time for others. Since *Static* has a lower hot method compilation ratio as seen in Figure 7, the detection time is much longer for some benchmarks (e.g., `png`, `compress`, `regexp`, `jess`, and `mpegaudio`)[5].

This is also true for *Simple* in `parallel`, but *Simple* compiles hot methods much earlier than *HotSpot* in other benchmarks. This seems to be due to its relatively low threshold of 12,000. In fact, when we compute the weighted invocation count of hot methods before being compiled for both *Simple* and *HotSpot*, we found that *Simple* compiles 10 times earlier than *HotSpot* on average. Even if we increase the threshold by doubling up until 120,000, we could not get any better performance since even if higher threshold keeps some cold methods from being compiled, it will also increase the detection time of hot methods. So, the problem is *Simple*'s runtime estimation based mostly on invocation counts.

6.6 Hot Spot Detection Overhead

As we described in Section 3, *FSRE* (or *Merged*) counts only important bytecodes, and the execution of the instrumentation code itself is involved with a low overhead (a simple arithmetic calculation with no branches). So we expect its overhead is low.

Figure 9 shows the dynamic ratio of important bytecodes to all bytecodes, which is an average of 12%. It includes an average 2.5% of backward branches, as shown in the bottom. So, FSRE counts only a fraction of the bytecodes and includes additional overhead to HotSpot, both of which are hard to evaluate, though.

On the other hand, we can evaluate the overhead when we count all bytecodes, as we did in the right column of Figure 1 where we

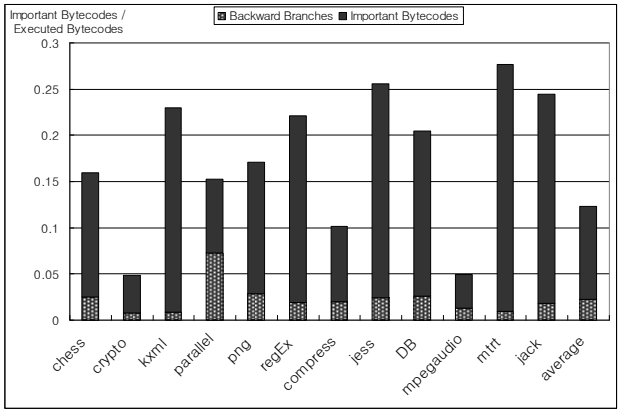

Figure 9. Ratio of important bytecode and backward branches

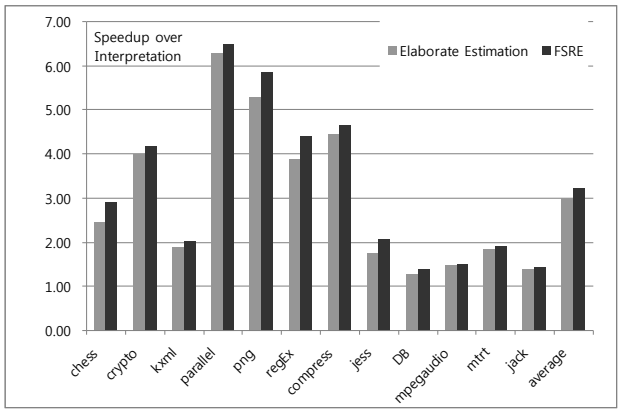

Figure 10. Comparison of FSRE and elaborate estimation

[5] Although the hot method compilation ratio in `mtrt` is high, its detection time is long. This is due to a hot (11%) method which is called 748 times. While others compile this method after the first invocation, *Static* compiles it at the 287th call, causing a relatively longer detection time.

update the T[*m*] in every byetcode executed. Figure 10 shows the performance of this elaborate version of FSRE compared to the original FSRE, which is 8% lower on average. This means that FSRE is much more efficient than an elaborate estimation.

7. Summary

Previous hot spot detection heuristics do not estimate the running time precisely because they oversimplify the important dynamic information. This may lead to compilation of cold methods, or delayed or failed compilation of hot methods. In this paper, we proposed a more precise dynamic runtime estimation technique with a small overhead, and a static runtime estimation technique for detecting candidate methods for first-invocation compilation. Our experimental results show that FSRE heuristics have a similar or better preciseness of hot spot detection than HotSpot but detect hot methods much earlier, improving its performance tangibly.

Although we need to optimize and tune it further, while hot spot detection is a difficult and somewhat arbitrary problem, we think FSRE appears to be a promising approach to detecting hot spots.

References

[1] Jonathan L. Schilling. *The Simplest Heuristics May Be the Best in Java JIT Compilers*. SIGPLAN Notices, 38(2):36-46, 2003.

[2] Sun Microsystems. *CDC Runtime Guide for the Sun Java Connected Device Configuration Application Management System version 1.0.* (Page 58), http://java.sun.com/javame/reference/docs/cdc_runtime_guide.pdf. The source code can be downloaded at http://download.java.net/mobileembedded/phoneme/advanced/phoneme_advanced-mr1-rel-src-b06-10_nov_2006.zip

[3] K. Kumar. *When and What to Compile/Optimize in a Virtual Machine?* SIGPLAN Notices, 39(3):38-45, 2004

[4] Sun Microsystems. *White Paper "CDC: An Application Framework for Personal Mobile Devices"*

[5] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification Reading*, Addison-Wesley, 1996.

[6] J. Aycock. *A Brief History of Just-in-Time*, ACM Computing Surveys, 35(2), Jun 2003

[7] J. Whaley. *A Portable Sampling-based Profiler for Java Virtual Machines*. Proceedings of ACM 2000 Java Grande Conference, June 2000, pp. 78-87.

[8] M. Arnold, S.Fink, D. Grove, M.Hind, and P.F. Sweeney. *Adaptive Optimization in the Jalapeno JVM*. Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) pp47-65, Oct. 2000.

[9] U. Holzle, C. Chambers, and D. Ungar. *Debugging Optimized Code with Dynamic Deoptimization*, Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, 1992.

[10] Sun Microsystems. *CDC CVM*, http://java.sun.com/products/cdc

[11] SPEC JVM98 Benchmarks. http://www.SPEC.org/jvm98.

[12] EEMBC GrinderBench Benchmarks. http://www.eembc.org.

[13] M. Arnold, S.J. Fink, D. Grove, M. Hind and P.F. Sweeney. *A Survey of Adaptive Optimization in Virtual Machine*. IBM Research Report RC23143, May 2004.

[14] M. Paleczny, C. Vick and C. Click. *The Java HotSpot Server Compiler*. Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01), pages 1-12. USENIX, April 2001.

[15] IBM. IBM Technology for Java Virtual Machine in IBM i5/OS, http://www.redbooks.ibm.com/redbooks/pdfs/sg247353.pdf

[16] S. Fink and F. Qian. *Design, Implementation, and Evaluation of Adaptive Recompilation with On-Stack Replacement*. Proceedings of 2003 International Symposium on Code Generation and Optimization (CGO), Mar. 2003.

[17] J. Smith and R. Nair. *Virtual Machines*, pages 314-316, Morgan-Kaufmann, 2005.