공학박사 학위논문

Some Optimizations for Accelerating Application Startups on Language Runtimes

프로그래밍 언어 런타임에서의

응용프로그램 시작 가속을 위한 최적화

2015 년 8 월

서울대학교 대학원

전기컴퓨터 공학부

이 성 원

Some Optimizations for Accelerating Application Startups on Language Runtimes

지도 교수 문 수 묵

이 논문을 공학박사 학위논문으로 제출함

2015 년 **7** 월

서울대학교 대학원

전기컴퓨터 공학부

이 성 원

이성원의 공학박사 학위논문을 인준함

2015 년 **7** 월

위 원 장 백 윤 흥 (인)

부위원장 문 수 묵 (인)

위 원 이 혁 재 (인)

위 원 이 재 진 (인)

위 원 정 동 헌 (인)

초 록

자바나 자바스크립트와 같은 프로그래밍 언어를 수행하는 런타임 환경은 응용프로그램의 이식성을 장점으로 하여 임베디드 소프트웨어 플랫폼으로써 널리 사용되고 있다. 자바 응용프로그램은 바이트코드의 형태로 배포되어 디지털 텔레비전이나 안드로이드 플랫폼에서 동작하며 자바스크립트는 소스 코드 형태로 웹 플랫폼에서 수행된다. 그러나 프로그래밍 언어 런타임에 의한 이식성은 본질적으로 성능 문제를 야기할 수 있는데, 하드웨어가 아닌 인터프리터와 같은 소프트웨어에 의해 응용프로그램의 바이트코드나 소스 코드를 수행하기 때문이다. 따라서 더 나은 성능을 얻기 위해 수행 중 바이트코드나 소스 코드를 기계어로 번역하는 적시 컴파일러나 inline caching과 같이 반복 수행되는 동작에 특화된 최적화를 프로그래밍 언어 런타임에 적용하기도 한다.

한편, 임베디드 시스템에서 동작하는 자바 응용프로그램이나 웹페이지의 로딩 중 수행되는 자바스크립트는 안정된 상태에서의 동작보다는 급격한 변화를 수반하는 시작 과정의 행태가 더 두드러진다. 따라서 비교적 짧은 수행시간을 가지고, 동일한 동작을 반복하는 경향이 낮으며, 수행시간에서의 비중이 높은 핫스팟이 드문 특징을 가진다. 그러나 핫스팟에 효과적인 적시 컴파일러나 반복되는 동작에 특화된 최적화는 이와 같은 응용프로그램 시동의 행태에 대하여 성능을 향상시키기 어려울 수 밖에 없다.

이 논문을 통하여 기존의 방식 보다 정교하게 추정한 수행시간을 근거로 작동하는 핫스팟 감지 기법을 제안함으로써 핫스팟이 불분명한 상황에서 자바 적시 컴파일러에 의한 수행 속도의 향상을 꾀하였다. 그 결과 응용프로그램 시작의 행태를 보이는 벤치마크 프로그램의 첫번째 수행시간을 기존의 HotSpot 자바 가상머신의 핫스팟 감지 기법 대비 약 10% 가속화할 수 있었다. 그리고 실제 응용프로그램으로서 디지털 방송에 의해 배포된 Xlet의 시작에 걸리는 수행시간 역시 약 7%가 개선되었다.

또한, 자바스크립트 적시 컴파일러에서 생성되는 기계어의 용량을 줄이기 위하여 축소된 명령어 집합에 최적화된 기계어를 생성하는 기법을 제안하였다. 이를 통하여 약 29%에 해당하는 기계어의 크기를 줄일 수 있었고, 이 결과는 웹페이지 자바스크립트의 시작 과정에서 수행되는 대량의 자바스크립트에서 더욱 효과적일 수 있다.

그리고 적시 컴파일러만을 사용하여 자바스크립트를 수행하는 환경에서 웹페이지 자바스크림트 시작 속도의 성능 저하가 나타남을 발견하였고, 이를 개선하기 위하여 인터프리터 수행을 기반으로 선택적 컴파일을 시도함으로써 적시 컴파일러에 의한 성능 저하를 최소화 하였다.
마지막으로 웹페이지 자바스크립트 시작의 수행 행태에 대하여 분석을 실시한 결과, 빈번하게 발생하는 객체에 대한 접근을 가속화할 수 있는 바이트코드 수준의 최적화를 제안한다. 인터프리터 수행에 적시 컴파일러를 추가로 적용하여도 웹페이지 자바스크립트 시작의 성능 향상은 없었던 반면, 제안한 바이트코드 수준의 최적화는 수행시간을 약 3% 가속화함으로써 웹페이지 자바스크립트 시작에 더 효과적인 것을 확인할 수 있었다.

주요어 : 적시 컴파일러, 핫스팟 감지, 코드 크기 최적화, 선택적 컴파일, 객체 접근, 바이트코드 수준 최적화
학 번 : 2007-21042

Contents

Chapter 1. Introduction 1
1.1 Hot Spot Detection 1
1.2 Memory Consumption of JIT Compiled Code 4
1.3 Web Page JavaScript Performance with JITC 5

Chapter 2. Enhanced Hot Spot Detection 8
2.1 Previous Approaches to Hot Spot Detection 8
2.1.1 Simple Heuristic 8
2.1.2 Hot Heuristic 9
2.1.3 Static Analysis Heuristic 10
2.2 Flow-Sensitive Runtime Estimation 11
2.3 Static-FSRE for First-Invocation Compilation 15
2.4 Merged Heuristic of Dynamic and Static FSRE 18
2.4.1 Threshold of FSRE 18
2.4.2 Merged Heuristic 19
2.5 Experimental Results 19
2.5.1 Benchmark Results 19
2.5.1.1 Experimental Environment 19
2.5.1.2 Evaluation Heuristics 20
2.1.1.3 Performance of the Five Heuristics 21
2.1.1.4 Preciseness of Hot Spot Detection 23
2.1.1.5 Hot Spot Detection Time 28
2.1.1.6 Hot Spot Detection Overhead 29
2.5.2 Digital TV Java Xlet Results 31
2.5.2.1 DTV Environment and Java Xlet application 31
2.5.2.2 Heuristic Adjustments 33
2.5.2.3 Performance Improvement and Comparison 33

Chapter 3. Code Size Optimization for JITC 40
3.1 JavaScript JITC in SFX and Thumb2 40
3.1.1 JavaScript and Execution Semantics 40
3.1.2 SquirrelFish Extreme and the Bytecode 41
3.1.3 SFX JITC Architecture 43
3.1.4 JITC Code Generation for Thumb2 45
3.2 SFX JITC Optimizations for Thumb2 45
3.2.1 Code Generation with Register Re-map 45
3.2.2 Constant Pool Aggregation 46
3.2.3 Patching PC-relative Branches 49
3.3 Experimental Result 52
3.3.1 Experimental Environment 52
3.3.2 Code Size Result 52
3.3.3 Performance Result 55

Chapter 4. Selective JITC for Web Page JavaScript 56
4.1 JavaScript and SFX JITC 56
4.1.1 JavaScript and Interaction with DOM 56

4.1.2 SFX JITC and Its Architecture....59
4.1.3 Benchmark JavaScript and Web Page JavaScript....62
4.2 Selective JITC for the SFX....64
4.2.1 Selective JITC....64
4.2.2 Selective JITC Implementation for the SFX....65
4.3 Experimental Result....66
4.3.1 Experiment Environment....66
4.3.2 Web Page JavaScript and SunSpider Benchmark....66
4.3.3 Web page JavaScript Execution Time....71
4.3.4 Comparison to Benchmark Execution Time....73
4.3.5 Evaluation of the Selective JITC Heuristic....74
4.3.6 Discussions....76

Chapter 5. Bytecode Level Optimizations....78
5.1 Analysis on Web Page JavaScript Execution....78
5.2 Overhead in Property Accesses....82
5.3 Super-Bytecode Construction (SBC)....85
5.4 Bytecode Chaining (BC)....86
5.5 Experimental Evaluation....87
5.5.1 **Performance Result**....88
5.5.2 Performance Analysis....89
5.5.2.1 Optimized Runtime Services with SBC....89
5.5.2.2 Removed Runtime Services with BC....90

Chapter 6. Related Work....92

Chapter 7. Conclusion....94

Bibliography....97

Abstract....103

List of Tables

Table 2-1. Comparing bytecode size of all cold methods compiled.....25
Table 4-1. Number of executed JavaScript functions 68
Table 4-2. Number of iterations, loops, and the ratio 70
Table 4-3. Dynamic bytecode size to static bytecode size.....71
Table 4-4. Ratio of JITCed functions to all executed functions 75
Table 4-5. The data of Table 4-2 for JITCed functions only 76

List of Figures

Figure 2-1. An example of FSRE and elaborate computation.....14
Figure 2-2. Performance ratio among heuristics with the HotSpot performance as a basis of 100%.....22
Figure 2-3. Fluctuation range for the FSRE performance.....23
Figure 2-4. Execution time coverage of all hot methods.....24
Figure 2-5. The ratio of compiled methods and compiled hot methods compared to all executed methods.....25
Figure 2-6. The ratio of hot methods compiled by each heuristic to all hot methods.....26
Figure 2-7. Comparison between T[m] of FSRE and the actual execution time.....27
Figure 2-8. Hot spot detection time of heuristics compared to the HotSpot detection time as a basis of 100%.....29
Figure 2-9. Ratio of important bytecode instructions and backward branches.....30
Figure 2-10. Comparison of FSRE and elaborate estimation.....31
Figure 2-11. DTV Xlet performance ratio with the HotSpot performance as a basis of 100%.....34
Figure 2-12. The ratio of compiled methods to all executed methods in DTV Xlet.....35
Figure 2-13. CDF of executed methods for their invocation counts in Benchmark and Xlet.....36
Figure 2-14. Ratio of dynamic bytecode size before compilation to dynamic bytecode size with interpretation only.....37
Figure 2-15. Ratio of dynamic size of bytecode interpreted for compiled methods compared to HotSpot.....38
Figure 2-16. DTV Xlet performance for the weighted static-FSRE compared to the original static-FSRE.....39
Figure 3-1. A JavaScript function and its SFX bytecode.....41
Figure 3-2. SFX macro assembler hierarchy for op_bitor.....44
Figure 3-3. Constant Pool Aggregation.....47
Figure 3-4. Load-to-Branch Patch in SFX-on-ARM.....50
Figure 3-5. Target Address Insertion in SFX-on-Thumb2.....51
Figure 3-6. Two-level jump in SFX-on-Thumb2.....52
Figure 3-7. Code size comparison.....53
Figure 3-8. Component Ratio of Branches in Executable.....54
Figure 3-9. Runtime Comparison.....55
Figure 4-1. An example JavaScript and the window screen.....59
Figure 4-2. SFX macro assembler hierarchy for op_bitxor.....62
Figure 4-3. Sample functions in the SunSpider benchmark.....63
Figure 4-4. Ratio of JavaScript execution time to loading time.....67
Figure 4-5. Distribution of the number of function calls.....69
Figure 4-6. Web page JavaScript execution time.....72

Figure 4-7 Comparing web page and benchmark execution time 74
Figure 5-1. JavaScript JITC Speed-up over Interpretation 79
Figure 5-2. Distribution of JavaScript execution during loading 80
Figure 5-3. Portion of property access in runtime services 81
Figure 5-4. Portion of runtime services in property access time 81
Figure 5-5. An example of the object property accesses 83
Figure 5-6. Redundant runtime service requests 84
Figure 5-7. An example of the super-bytecode construction 85
Figure 5-8. An example of the speculative caching 87
Figure 5-9. An example of the lazy caching 87
Figure 5-10. Speed-up over JITC only execution 88
Figure 5-11. Runtime service invocations replaced by optimized runtime services for SBC 89
Figure 5-12. Runtime services invocations removed by BC 90

Chapter 1. Introduction

1.1 Hot Spot Detection

Java has been popularly employed as a standard software platform from enterprise servers to embedded systems, due to its support for platform independence, security, and reliability [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 at runtime, often on a method-by-method basis (Some JITC translates only hot portions of a method [36], but we assume a method-based JITC in this paper).

Since the translation overhead is a part of the running time, most JITCs employ *adaptive compilation*, where a method is interpreted or compiled by the baseline compiler initially, and then is compiled with optimizations only when it is found to be hot [13]. The method can be compiled again with more optimizations if it is found to be really hot [8]. 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 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 a part of the running time, so we cannot use an elaborate technique that takes too much time.

Effective hot spot detection is especially important for the embedded systems. Running an embedded Java application is often composed of the application loading and the event-driven computation. Application loading initializes the JVM, the framework or the middleware classes, and displays the initial screen. Event-driven computation executes the corresponding event handlers based on the user interaction and displays the updated screen. Both loading and event handling tend to be short, to provide a fast user response time. Also, users often switch to different applications frequently, instead of sticking to a single application. This is

in sharp contrast with enterprise Java servers, which often run for a long time (e.g., hours, days, or months) with a heavily executed set of hot methods. According to the previous studies [20, 26], the performance with this "steady-state" of definite, hot methods is affected little by the hot spot detection technique since all of them will be detected and compiled eventually, and the long running time would make the detection overhead and speed less important. On the other hand, running an embedded application would behave as the "start-up" phase of hot methods, where the hot methods execute far less frequently than in the steady-state of the server applications, thus less definite. So precise, fast, and low-overhead hot spot detection would be important for the performance of embedded systems.

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. To decide if a method has been running long enough, we need an information on the running time of the method. The differences among the heuristics are how to obtain such information, how precise it is, and how much overhead is involved with.

The most precise way to obtain the running time of a method is to simply measure the real time difference between its entry and exit and to accumulate the time whenever the method is executed. If there are method calls within a method, the running time of the callee methods is excluded from the running time of the method. Unfortunately, such a timing function is costly to invoke in the embedded systems since it is often involved with a system call overhead to read the hardware timer [19]. Also, Java methods tend to be short, thus being called frequently, and this would make the time measurement 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 if the sampling interval is long enough. Unfortunately, sampling is not very effective for embedded applications since a relatively long sampling interval (similar to the one

used in server systems) compared to their short running time is likely to miss the hot spots, while a short sampling interval based on frequent sampling would suffer from the overhead of frequent timer event handling. Moreover, sampling might not detect those hot methods executed in-between of the sampling periods (e.g., a timer event handler executed on a regular interval which misses the sampling period) [29]. This is especially true in the start-up phase (which would correspond to the execution behavior of embedded application as noted above), since a relatively large working set of methods are executed in a short period of time [26].

Many techniques attempt to estimate the running time based on software *counters* such that they count some interpreted bytecode instructions at runtime. The estimated running time of a method is obtained based on these counter values and if it is higher than a given threshold, the method is regarded as a hot spot. The counter-based approach appears to be more effective than the sampling in the embedded systems, because it can count over the whole execution period of the embedded application, identifying hot methods earlier and more precisely. Unfortunately, this can increase the execution time due to the counting overhead, so most techniques try to reduce it. The *Simple* heuristic counts only method invocations without considering loops [1], while Oracle'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 bytecode instructions but does not measure their dynamic counts [3]. Although these techniques can reduce the runtime overhead, they may affect the quality of hot spot detection such that cold methods are compiled or hot methods are delayed or failed to be compiled, which can affect the overall performance negatively.

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 bytecode instructions following the execution control flow, yet is not involved with a serious overhead. The idea is counting only important bytecode instructions in a flow-sensitive manner and then estimating the total count of bytecode instructions with a simple arithmetic calculation. We also propose a *static-FSRE* which allows some hot methods to be compiled in their first-

invocation via one-pass static analysis of the methods, complementing FSRE. We implemented FSRE in a JITC on a CDC JVM [10]. We evaluated it with the SPECjvm98 [11] and EEMBC [12] benchmarks by running them once, which is regarded as behaving as a start-up phase according to the previous studies [8]. We also experimented with a DTV Java Xlet application on a commercial TV, which behaves more of a start-up phase even than the benchmarks due to fewer method calls and loop iterations. In both experiments, FSRE shows a tangible performance benefit compared to the Oracle's HotSpot heuristic.

1.2 Memory Consumption of JIT Compiled Code

Full web browsing is becoming one of major killer applications of smart phones [46], because mobile phone users are moving from traditional mobile content such as games, SMS, wallpapers, and ring tones, to web-based content such as searching, maps, blogs, SNS, and messengers [43]. This will be accelerated by the advent of *rich internet application* (RIA) [47], a web-based application distributed and maintained through web browsers, because mobile RIA is also emerging. Even without a full web browser, mobile *widget*s allow accessing specific web sites for customized information display and interaction [53]. Even *mobile Web OS* such as Palm WebOS [45] or Chrome OS [48] are announced.

One issue for mobile web browsing is JavaScript performance. Most popular web sites are programmed with the JavaScript code, so it can be downloaded to the mobile browser and executed. Although the JavaScript code embedded in the traditional web pages used to be simple, its execution with a low-performance mobile CPU would require a running time much longer than with a desktop CPU. The mobile widgets are also programmed via JavaScript for accessing the web sites and for handling the user interface. Moreover, RIA is programmed heavily with JavaScript these days to reduce plug-in based ones (Flash, Silverlight). These trends will definitely require high-performance JavaScript engines.

One promising and proven solution for accelerating the JavaScript performance is employing just-in-time compiler (JITC), which translates JavaScript code to machine code at runtime, so as to execute JavaScript in a native form. A few high-performance JITCs have been announced such as TraceMonkey for the FireFox brower from Mozilla [51], V8 for the Chrome browser from Google [52], and the

SquirrelFish Extreme (SFX) for the WebKit based browser [49]. In the mobile area, SFX is getting a dominant popularity and is being employed in product smart phones such as iPhone, Palm pre, or S60, due to the WebKit as an open-source browser engine and some proven performance benefit.

One of the issues of mobile JITC is that since the mobile phone suffers from tight memory constraints, the JITC needs to keep a small memory footprint by generating small-sized machine code. For this, many mobile CPUs support half-sized encoding as in the ARM Thumb2 [38], with small performance degradation. So, it is necessary to provide a JavaScript JITC for such CPUs.

This paper describes our experiences in the code generation and optimization for a mobile JavaScript JITC, in the context of the SFX and the ARM Thumb2. In order to reduce the code size of an existing ARM JITC while affecting the performance little, we try to generate as many 16-bit half-sized instructions as possible and reduce the data area that saves large constants or addresses. One constraint is that WebKit, the on-going open source project, guides the SFX JITC developers to conform to its rigid code generation structure, which actually makes code optimization difficult. However, we could obtain a competitive result of code size and performance with our code generation and optimization.

1.3 Web Page JavaScript Performance with JITC

JavaScript is an object-oriented programming language designed to be executed by the interpreter [54]. JavaScript allows modifying the structure of an object at runtime based on *prototypes* and accessing the variables of outer functions from inner functions using *closures*. It also supports *dynamic typing*. For web browsing environment where web pages are loaded and displayed, these features can be utilized to partly change the previously painted screen dynamically, on a user's request thru the mouse click or the keyboard input.

Web pages are often programmed using JavaScript to control the document object model (DOM), an object defined to depict the web page on the web browser. JavaScript source program is transmitted from the web server to the web client, along with the HTML document describing the structure of the web page components and the CSS document depicting the visual effect of those components. Many web pages actively use the client-side JavaScript to follow the Web 2.0 trend

which enables web pages to be reconstructed frequently with the user interaction. This is accelerated by the advent of *rich internet application* (RIA) [16], a web-based application distributed and maintained through browsers, because RIA is programmed heavily with JavaScript these days to reduce plug-in based ones (e.g., Flash, Silverlight). Consequently, loading of web pages requires the execution of JavaScript code, which can sometimes cause a noticeable delay, especially in embedded web clients such as smart phones.

Smart phones are equipped with a full-web browser so as to access the web pages as in desktops. Due to its low-performance CPU and memory constraints, full web browsing by smart phones might suffer more from JavaScript performance than in the desktops. This problem can also be worsened as the mobile RIA and the web OS are popularized since they are heavily programmed with JavaScript. These trends will definitely require high-performance JavaScript engines.

One promising and proven solution for accelerating JavaScript performance is employing just-in-time compiler (JITC), which translates JavaScript code to machine code at runtime, so as to execute JavaScript in a native form. A few JITC-enabled JavaScript engines are being used such as TraceMonkey for the FireFox brower from Mozilla [51], V8 for the Chrome browser from Google [52], and the SquirrelFish Extreme (SFX) for the WebKit based browser [60]. In the mobile area, SFX is getting popular along with V8, and SFX is employed in many product smart phones including the iPhone [57].

We found that the SFX on a smart phone actually achieves a much better performance with JITC than with interpretation for the SunSpider JavaScript benchmark [50], as expected. The benchmark represents the typical workload of mathematical and logical computation. For real web pages, however, we found that the JITC-enabled SFX performs worse than the interpreter-based SFX. And the reason is that the web page JavaScript functions are not reused frequently as in the benchmark, which makes the compilation overhead higher than its benefit. Since the SFX JITC compiles all executed functions at their first invocation, this behavior would make such a JITC less effective. So the problem is how to make a JITC work better for the web page JavaScript. It is also important to keep the performance advantage for JavaScript benchmarks as well, since they would represent the workload of future mobile RIA or the web OS.

We propose a *selective* JITC for JavaScript engine so as to compile only hot functions detected during interpretation. We can expect that this adaptive compilation reduces the JITC overhead compared to compiling all executed functions, while achieving a good performance for JavaScript benchmark. We implemented selective compilation for the SFX on a real smart phone and experimented with some JavaScript-heavy web sites and the SunSpider benchmark. Our preliminary performance results show that selective compilation meets our expectation somehow, yet there are some differences and difficulties compared to other adaptive compilation environment (e.g., Java HotSpot), which we will discuss and analyze in detail in this paper.

First, we performed a study on the web page JavaScript behavior on a real smart phone platform unlike previous studies on the desktops. Despite of difficulties in measurements such as fluctuations of embedded platforms or variations caused by the change of web page contents (e.g., commercials), we could obtain some consistent, real mobile JavaScript behavior. Secondly, we introduced selective compilation to the product JavaScript JITC and evaluate its benefits and problems for the web page JavaScript as well as the benchmark JavaScript. These results will be useful for understanding JavaScript JITC and designing an efficient one.

Chapter 2. Enhanced Hot Spot Heuristic

2.1 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 factors, as follows:

[1] *Preciseness*: we should detect as many hot methods as possible but should not misjudge cold methods as hot ones.

[2] *Detection time*: for hot methods, we should detect and compile them as early as possible.

[3] *Detection overhead*: the detection overhead should be small.

We will review existing heuristics based on these factors as below.

2.1.1 Simple Heuristic

Simply considering the number of execution time and the method size is efficient at identifying hot spots [1]. *Simple heuristic* 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 compiling large methods earlier. 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:

$$T[m] = C1 * \text{invocation count of } m + C2 * \text{method size of } m$$

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.

2.1.2 HotSpot Heuristic

The hot spot detection heuristic of Oracle's HotSpot CDC JVM counts the backward branch as well as the method invocation, 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]:

$$T[m] = C3 * \text{invocation count of } m + C4 * \text{backward branch count in } m$$

Counting backward branch is simple to implement since we can just add the instrumentation code at the switch-case statement of each branch bytecode instruction in the interpreter loop. Although this allows HotSpot heuristic to consider loops, estimating the running time more precisely than the Simple heuristic, HotSpot heuristic completely ignores the size of a loop and a method, or any control flows within a method. Still, HotSpot heuristic 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 heuristic 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.

2.1.3 Static Analysis Heuristic

Unlike other heuristics that estimate the runtime of a method, T[*m*], based on dynamic events such as method calls or backward branches, there is an approach which statically analyzes the runtime of a method, S[m] [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 one (accurate analysis). In the former case of inaccurate analysis,

$$\text{S[m]} = \text{method size of } m + C5 * (\text{number of big loops in } m)$$
$$+ C6 * (\text{number of small loops in } m)$$

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:

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

This heuristic gives different costs to different bytecode instructions, so the method size or the loop size is not just the bytecode size but is a sum of the bytecode instruction 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 bytecode instructions unlike in HotSpot heuristic. 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 benefit in exploiting the imprecise estimation S[*m*] in such a complicated manner.

2.2 Flow-Sensitive Runtime Estimation

Previous runtime estimation techniques oversimplify loop iteration counts or loop/method sizes to reduce the estimation overhead. This can lead to imprecise hot spot detections. For example, the Simple heuristic does not consider loop iterations, so it cannot detect a hot method with heavy loop iterations if it is called infrequently. Similarly, the HotSpot heuristic ignores the method size or the loop size, so it might miss a hot method called infrequently but with a huge method body or loop body. The Static analysis heuristic leads to imprecise estimation of the running time due to the static decision of the loop iteration count. 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 bytecode instructions. However, it does not actually count all bytecode instructions but "*important*" bytecode instructions only, and then calculate the total count based on them in a control-flow sensitive manner. There are two types of important bytecode instructions in FSRE.

The first type is *heavy-weight* bytecode instructions. We classify the Java bytecode instructions into simple bytecode instructions and heavy bytecode instructions. Simple bytecode instructions 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 bytecode instructions are those whose execution takes a longer time than simple bytecode instructions 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 given a total weight of 21). The weight of a bytecode instruction 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 instructions or the weight of each heavy bytecode instruction is determined by measuring its real execution time on a given hardware platform as follows. We made a micro-benchmark with a loop which does nothing. We compiled the benchmark and modified the class file so

that the loop executes a bytecode instruction repetitively. Then, we measured the running time for each bytecode instruction. We found that some bytecode instructions show a tangibly long running time compared to simple bytecode instructions, so we gave a weight to them proportional to their running time.

The other type is *control-flow* bytecode instructions such as branches or returns. For a branch bytecode instruction, 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 a branch bytecode instruction, 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 bytecode instructions executed so far. There are four events during the execution of a 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 bytecode instructions in *m*:

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

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

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

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

By the time when this heavy bytecode instruction 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 instruction is executed, T[*m*] should be incremented because executing a backward branch means that a new iteration of a

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

T[*m*] -= (method size – (address of return bytecode instruction

- 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 2-1 illustrates FSRE for an example method *m()* in the right hand side. Each column in Figure 2-1 shows the updated trace of T[*m*] for each iteration of the inner loop or the outer loop. The left columns of Figure 2-1 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 instruction is executed, when T[*m*] of 145 is produced.

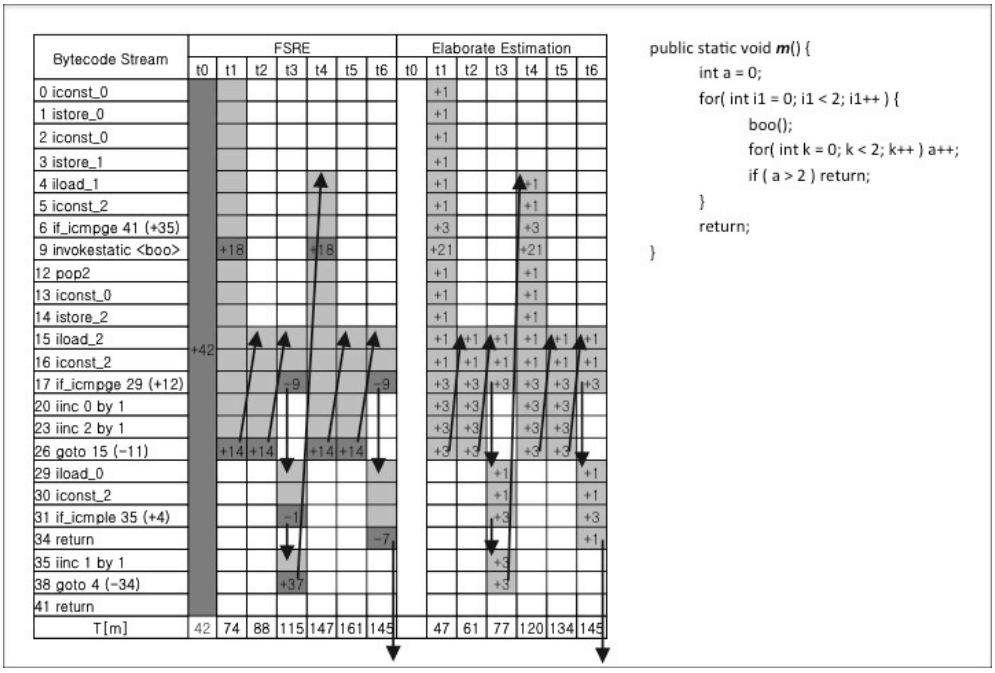

Figure 2-11. An example of FSRE and elaborate computation.

The right columns of Figure 2-1 illustrate when we update T[*m*] at *every* interpreted bytecode instruction by adding its weight to T[*m*]. Although FSRE updates T[m] at only important bytecode instructions, it produces the same T[*m*] that this elaborate computation produces②. Our experimental result shows that there is a clear correlation between our T[*m*] and the actual running time of a method *m*, indicating the preciseness of our T[m] (see Figure 2-7 in Section 3.6.1.4).

As to the FSRE overhead, the fraction of important bytecode instructions among all executed bytecode instructions 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 instruction in the interpreter loop.

② Actually, there can be some minor differences between the two T[*m*] if there are some null bytecode instructions padded by javac.

2.3 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 bytecode instructions, as follows:

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

$$\text{P}[m] = \text{method size of } m$$

(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*]:

$$P[m] \mathrel{+}= C * (\text{offset} + \text{byte size of the backward branch})$$

(3) If there is a doubly 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 two terms to P[m]. The first term is C * (outer_loop_offset – inner_loop_offset), which is the outer loop part not belonging to the inner loop, multiplied by C since the outer loop will make it iterate by C times. The second term is (C-1) * C * (inner_loop_offset + byte size of inner loop branch), which is the inner loop part, multiplied by (C-1) * C since the outer loop will make it iterate (C-1) * C times (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 these products to P[*m*]:

$$P[m] \mathrel{+}= C * (\text{outer\_loop\_offset} - \text{inner\_loop\_offset}) + (C-1) * C * (\text{inner\_loop\_offset} + \text{byte size of the branch})$$

Actually, the first term can be viewed as the length of the outer loop not belonging to the inner loop, multiplied by C, and the second term can be regarded as the one added to P[m] in the inner loop in (2), multiplied by C-1. So, it can be described as:

$$P[m] \mathrel{+}= C * (\text{length of the outer loop excluding the inner loop}) + (C-1) * (\text{the amount added to P[m] by the inner loop})$$

(4) We can extend (3) to multiply-nested outer loops, so when we meet an outer loop branch, we add the following to P[*m*]:

$$P[m] \mathrel{+}= C * (\text{length of the outer loop excluding any inner loops}) + (C-1) * \sum_{\forall\ inner\ loops} (\text{the amount added to P[m] by each inner loop})$$

So, the length of the outer loop not belonging to any inner loops is multiplied by C, and the amount previously added to P[m] by each inner loop is added together, which is then multiplied by C-1. These two terms will be added to P[m].

(5) As to the heavy bytecode instruction, 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 instruction)

We may want to add the weights of heavy bytecode instructions in a nested loop considering the nesting depth, which would make P[m] more precise. However, this will raise an overhead issue. For each heavy bytecode instruction we met during the traversal, we need to remember its location so that when we meet a backward branch, we need to check if the heavy bytecode instructions are within the offset and to decide how to add the weight, considering the nesting depth. This will increase the overhead of computing P[m], especially when there are many heavy bytecode instructions and many nested loops. In fact, we need to compute P[m] for all methods, so the overall overhead would be non-trivial. We actually implemented this idea and found that this affects the running time tangibly (see Figure 2-16 in Section 3.6.2.2).

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

If we apply static-FSRE to the example in Figure 2-1 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.

2.4 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.

2.4.1 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 Oracle'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:

$$T < C3 * \text{invocation count of } m + C4 * \text{backward branch count in } m$$

The constant T in the left of the inequality is the threshold while the right is HotSpot heuristic'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 heuristic, hence not directly applicable to FSRE③. However, we want to decide a new threshold using the well-tuned T value of HotSpot heuristic.

If we compare the T[*m*] of HotSpot heuristic 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 bytecode instructions with no branches. In this case, the HotSpot heuristic's 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,

③ When we actually experiment with the original T with FSRE, the performance is much worse (-18%) than the original HotSpot heuristic.

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:

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

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:

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

In the current implementation, we used five for C8.

2.4.2 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, and compile it when its inequality is satisfied.

2.5 Experimental Results

Previous section described our proposed FSRE heuristics. In this section, we evaluate them compared to HotSpot heuristic and others. We experiment with benchmarks first, followed by the real Java application used in the digital TV environment.

2.5.1 Benchmark Results

We first evaluated the five heuristics for benchmarks on an embedded board.

2.5.1.1 Experimental Environment

We performed the experiments 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.

Our CPU is a MIPS-based SoC called ATI Xilleon. 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 javac④) [11] and EEMBC [12]⑤. Each benchmark is run once, which is often used to observe hot spot behavior of the start-up phase (in contrast, they are run 5 or 10 times continuously to observe the steady-state behavior) [8, 26]. We run each benchmark 10 times, chose three numbers in the middle, and took their average. We did this because of severe fluctuations in some benchmarks (e.g., regex, chess, or mtrt), which would be due to the embedded environment whose performance is more sensitive and fluctuating than in the desktop environment; limited resources in the embedded systems are likely to make the computations not fit in the cache or the DMA, executing them unstably [34].

2.5.1.2 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 6,000, 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.

④ We excluded javac because the memory overflows when one method is compiled by the JIT compiler, crashing the JVM.

⑤ 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.

(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.

2.5.1.3 Performance of the Five Heuristics

Figure 2-12 is for comparing the performance ratio among all heuristics with the *HotSpot* performance as a basis of 100%. We also show the performance of the interpreter (denoted by Intpr) for comparison. Figure 2-3 shows the median numbers of *FSRE,* those which are almost the same as the average numbers in Figure 2-2, with the highest and the lowest numbers to show the fluctuation range.

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% (geometric mean).

Merged shows the best performance, an average of 7.4% 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 time 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 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*.

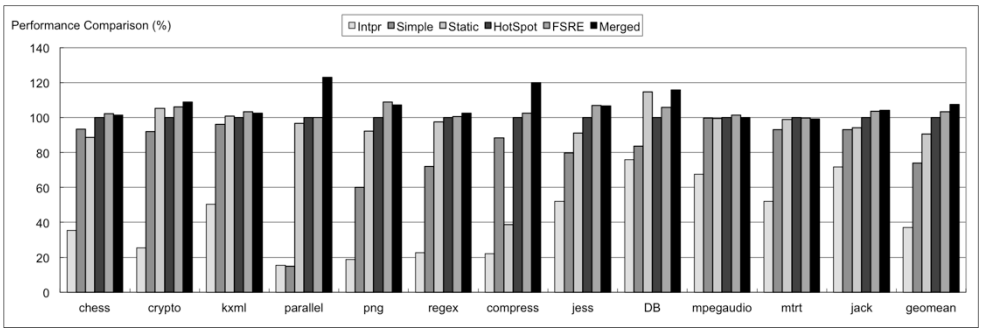

Figure 2-12. Performance ratio among heuristics with the *HotSpot* performance as a basis of 100%.

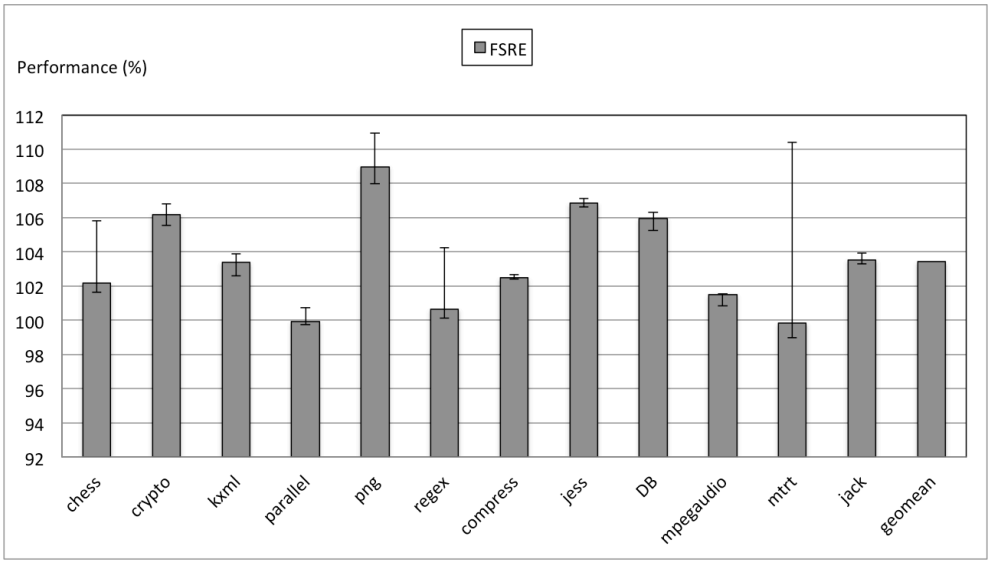

Figure 2-13. Fluctuation range for the *FSRE* performance.

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.

2.5.1.4 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. To identify such hot methods, we used the JProfiler [30] and ran it in interpretation-only mode on the desktop (JProfiler cannot be installed in the target SoC). Figure 2-4 shows the execution time coverage of all hot methods in each benchmark. On average, these hot methods cover 86.5% of the total execution time.

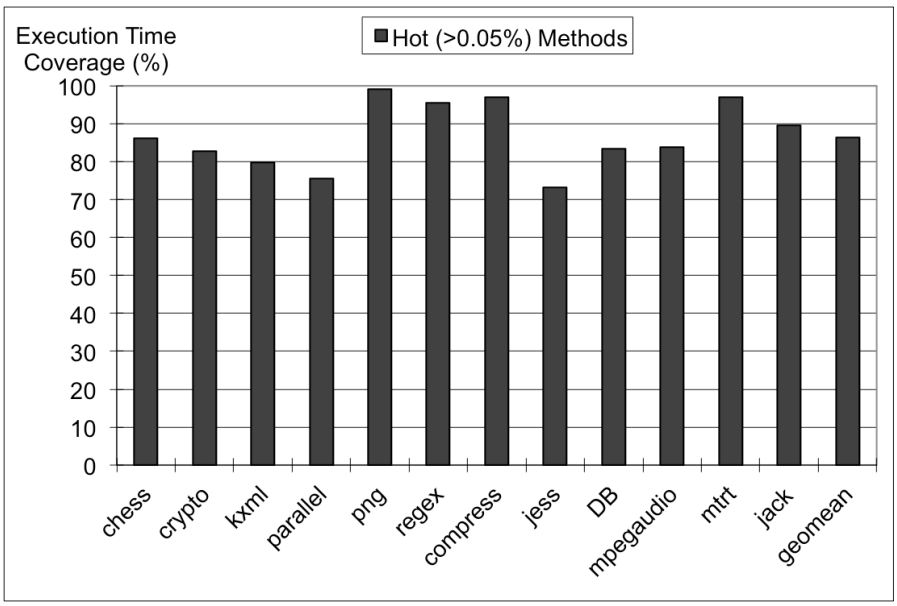

Figure 2-14. Execution time coverage of all hot methods.

We then measured how many methods are compiled by each heuristic and how many of them are hot methods. In Figure 2-5, 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 7% of executed methods, around half of which are hot methods. *FSRE, Merged,* and *Static* compile slightly more methods than *HotSpot* in this order, but they compile almost the 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.

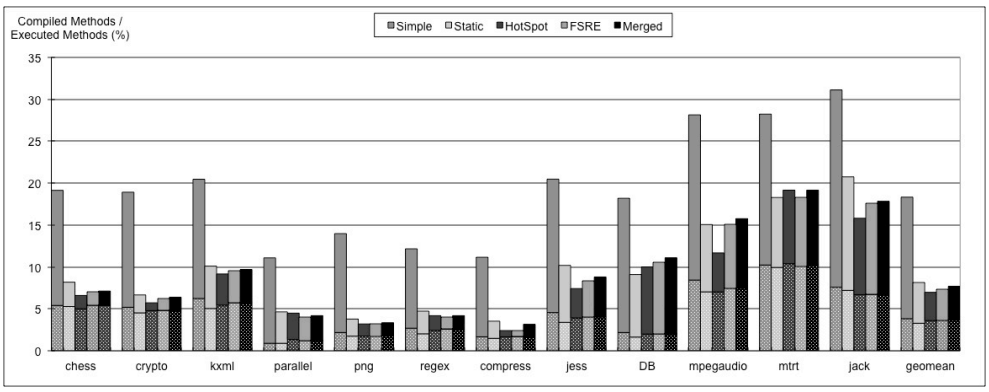

Figure 2-15. The ratio of compiled methods and compiled hot methods compared to all executed methods.

Table 2-1 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 24 times more cold bytecode instructions 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 2-12.

Table 2-2. 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
geomean	2389.7	251.4	100.0	108.1	118.8

We also checked the quality of compiled hot methods. Figure 2-6 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 2-5. 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 2-12.

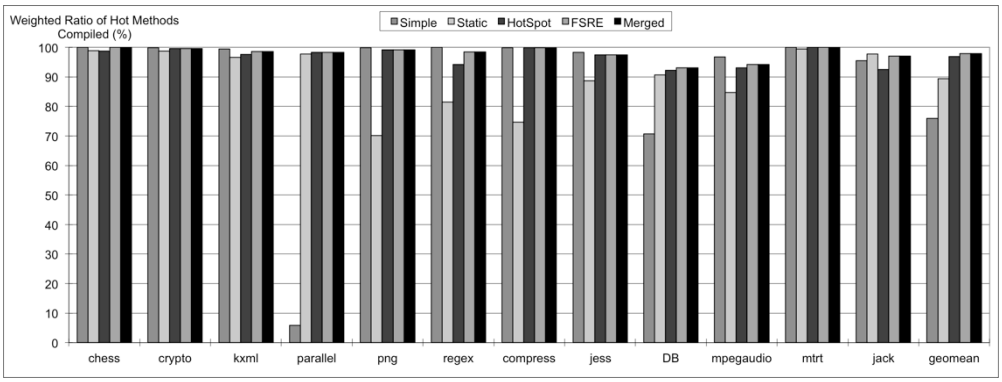

Figure 2-16. The ratio of hot methods compiled by each heuristic to all hot methods.

One thing to note from Table 2-1 and Figure 2-6 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 2-12 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 in the next subsection.

As to the preciseness, we perform one more experiment. Figure 2-7 shows T[*m*] computed by *FSRE* on our SoC, compared to the actual execution time obtained by the JProfiler on the desktop, for each hot method when we run the benchmarks in the interpreter mode. Although they were computed and measured on different environments, Figure 2-7 shows a clear correlation between T[*m*] and the actual execution time, indicating that *FSRE* estimates the running time relatively precisely.

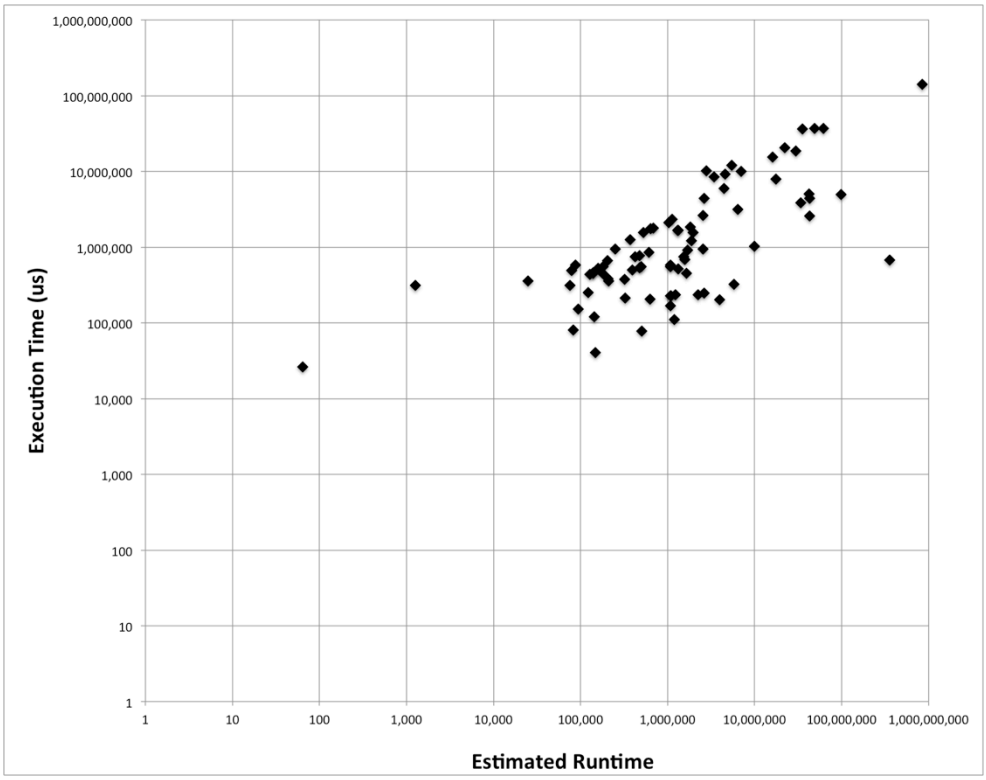

Figure 2-17. Comparison between T[m] of *FSRE* and the actual execution time.

2.5.1.5 Hot Spot Detection Time

Another important requirement of good hot spot detection is that hot methods should be detected and compiled as early as possible. Evaluating the hot method detection time requires measuring the time spent for interpretation before being compiled. Unfortunately, we cannot measure such a repeated, short period of interpretation time precisely in our embedded MIPS board. Instead, we estimate as 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 the interpretation-only mode to find its total interpretation time, which is then multiplied by the execution proportion of each hot method, obtained from the JProfiler. We multiply this total interpretation time of each hot method by the ratio of invocation count, which gives an estimated interpretation time of the method before being compiled, which we call the *detection time*.

As we saw in Figure 2-6, the set of hot methods compiled by each heuristic differs somewhat. So, we summed up the detection time for a *union* of methods compiled by any of the heuristics. If a hot method is not compiled at all by some heuristic, the method's total interpretation time will be added to the detection time. Now, the sum of detection time for each benchmark can be regarded as the *hot spot detection time*.

Figure 2-8 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 *6.1.4*). And *Merged* spends even less detection time due to its first-invocation compilation in DB, parallel, and compress, consistent with our observation in Section *6.1.3*. On average, *FSRE* and *Merged* spends 37% and 53% 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 2-6, the detection time is much longer for some benchmarks (e.g., png, compress, regex, jess, and mpegaudio)⑥.

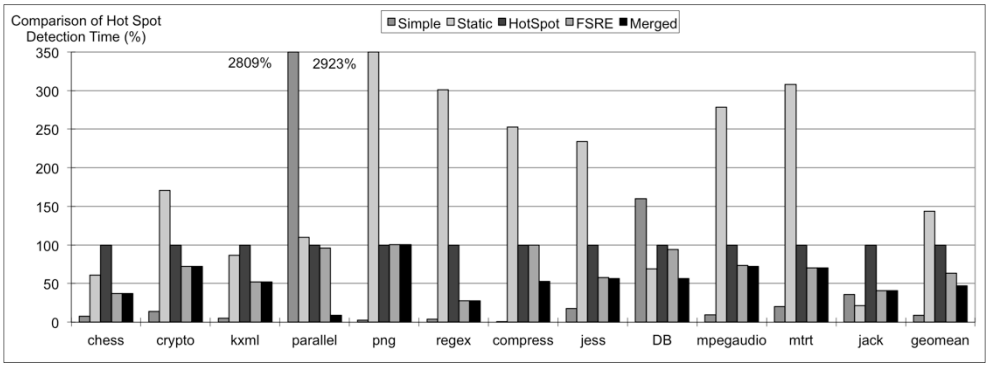

Figure 2-18. Hot spot detection time of heuristics compared to the *HotSpot* detection time as a basis of 100%.

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.

2.5.1.6 Hot Spot Detection Overhead

As we described in Section 3, *FSRE* (or *Merged*) counts only important bytecode instructions, 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 2-9 shows the dynamic ratio of important bytecode instructions to all bytecode instructions, which is an average of 12%. It includes an average 1.9% of

⑥ 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.

backward branches, as shown in the bottom. So, *FSRE* counts only a fraction of the bytecode instructions and includes additional overhead to *HotSpot*, both of which are hard to evaluate, though.

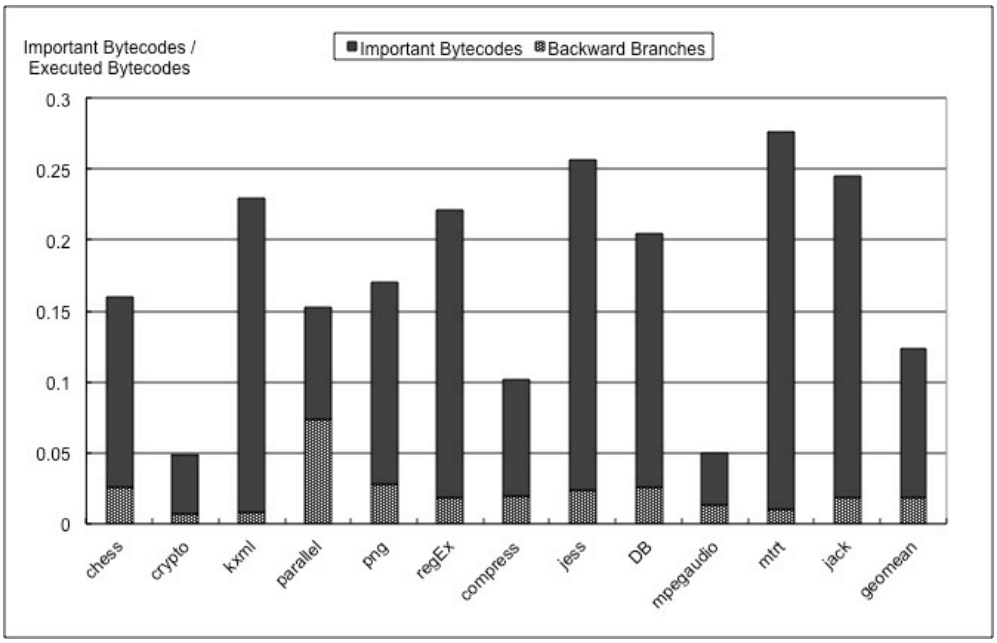

Figure 2-19. Ratio of important bytecode instructions and backward branches.

On the other hand, we can evaluate the overhead when we count all bytecode instructions, as we did in the right columns of Figure 2-1 where we update the T[*m*] in every byetcode executed. Figure 2-10 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.

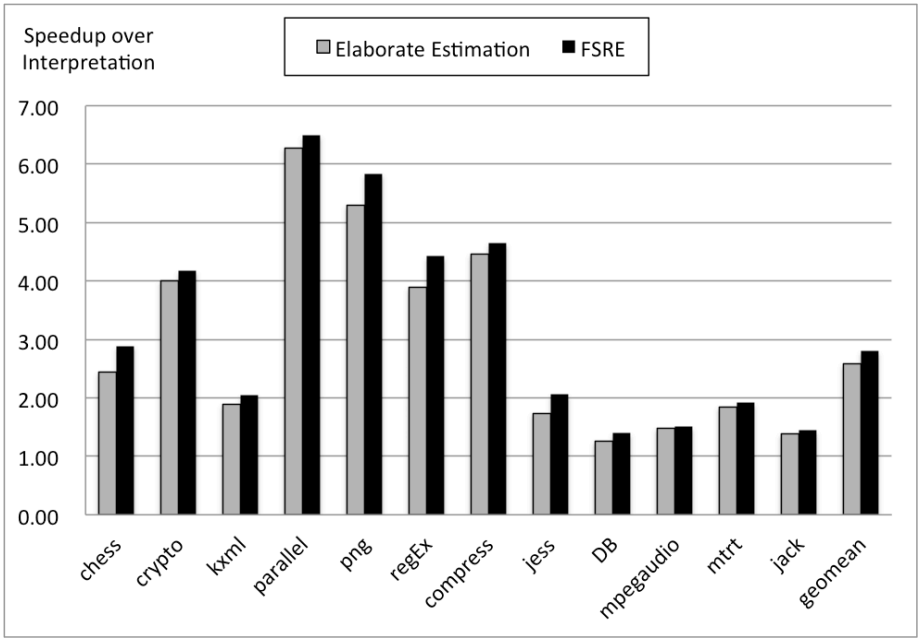

Figure 2-20. Comparison of *FSRE* and elaborate estimation.

2.5.2 Digital TV Java Xlet Results

We also evaluated the five heuristics on a commercial digital TV (DTV) platform with an on-air Java application. We first describe the DTV environment and the behavior of the Java application compared to the benchmarks. We then show how the response time of the DTV Java application is improved by the proposed heuristics.

2.5.2.1 DTV Environment and Java Xlet application

The DTV can broadcast data in addition to picture/sound due to higher bandwidth, and the key technology of the data broadcasting is Java. The Java-based data broadcasting is programmed using a Java *Xlet application*, which is composed of the Xlet class files and the image/text files [18]. The Xlet application is broadcasted to the DTV set-top box and executed there interacting with system and middleware classes such as Advanced Common Application Platform (ACAP). It is an event-driven program where on a user request the chosen information such as weather, stock, news, traffic, and program schedule is displayed on the TV screen. Details of the DTV software architecture and the Xlet life cycle can be found in [28].

Our DTV experimental environment is as follows. We used a product DTV provided by a global manufacturer. It has a 333Mhz MIPS CPU with a 128MB memory. Our target DTV platform employs an open source version of Oracle's Connected Device Configuration (CDC) JVM, called the phoneMe Advanced MR2 [31], which includes a JITC based on Oracle's HotSpot technology. The OS is Linux kernel 2.6.12. We experiment with the Xlet application broadcasted by a TV station in Korea, whose size is around 2MB. When we execute the Xlet application, we measure the running time of displaying the chosen information on the TV screen when each menu item is selected using the remote control. There are five menu items: a) "news" to print out the headline news broadcasted in real time, b) "weather" to show the local weather forecast, c) "traffic" to graphically show traffic conditions of major streets, d) "stock" to show the information of the stock market and draw charts for the stock prices, and e) "program" to display a time table for TV programs of the channel. How these menu items are run and displayed on the DTV can be found from a video clip in [35].

Unlike the benchmark experiments in Section 6.1, it is not possible to use the JProfiler to obtain the runtime portion of each Xlet method. That is, it is impossible to run a stand-alone Xlet application on the desktop JVM with the ACAP middleware and the event-driven computation on a TV screen. Also, the JProfiler cannot be installed correctly on the DTV platform. Adding timestamps to the method invocation would not compute method running time precisely, as in our benchmark experiments on the embedded board. This makes it difficult to find out hot methods precisely and estimate those data in Figure 2-4 to 2-8 and Table 2-1. So, in the Xlet experiment, we mainly interested in the running time of each heuristic for each menu item, and we measure other relevant data based on the bytecode size to understand the result indirectly.

The running time is measured as follows. We added a timing routine to measure the time between when an event handler for a chosen menu item starts execution and when it completes execution and loads the updated screen. The delay for the transmission of data is not included in the runtime because we experimented after the Xlet applications including class, image, and text files are completely downloaded from the TV station.

2.5.2.2 Heuristic Adjustments

We evaluated the five heuristics on the phoneME advanced MR2 JITC. As we mentioned in Section 2.2 (footnote), the original HotSpot JITC including the one in the phoneME advanced MR2 takes the transition overhead between interpreted methods and JITC methods into consideration when estimating the running time of a method, unlike the CVM RI we used for the benchmark experiments. So, the HotSpot heuristic's inequality in Section *6.1.2* is modified as follows:

T < C3 * invocation count of *m* + C4 * backward branch count in *m* + **C5 * transition count of m**, where C5 is 30.

The transition count for a method *m* means a method invocation count from a JITC method to *m* (which is being interpreted) or from *m* to a JITC method. We added the transition overhead to the other four heuristics to make a consistent and fair evaluation. We then adjusted some of the constants for each heuristic to make the best performance for Xlet execution. For *Simple*, we adjusted the threshold from 12,000 to 20,000 while we adjusted C5 from 30 to 200 for *Static*, *FSRE*, and *Merged*.

2.5.2.3 Performance Improvement and Comparison

Figure 2-11 shows the performance ratio of each heuristic and its fluctuation range, with *HotSpot* as a basis of 100% for those five Xlet menu items, when each of the five heuristics is employed. We also show the performance of the interpreter (denoted by Intpr) for comparison. On average, *Merged* performs the best, followed by *FSRE*, *HotSpot*, *Static*, and *Simple* in this order, as in the benchmark results in Figure 2-2. However, there are two things to note, which are in sharp contrast to the benchmark results.

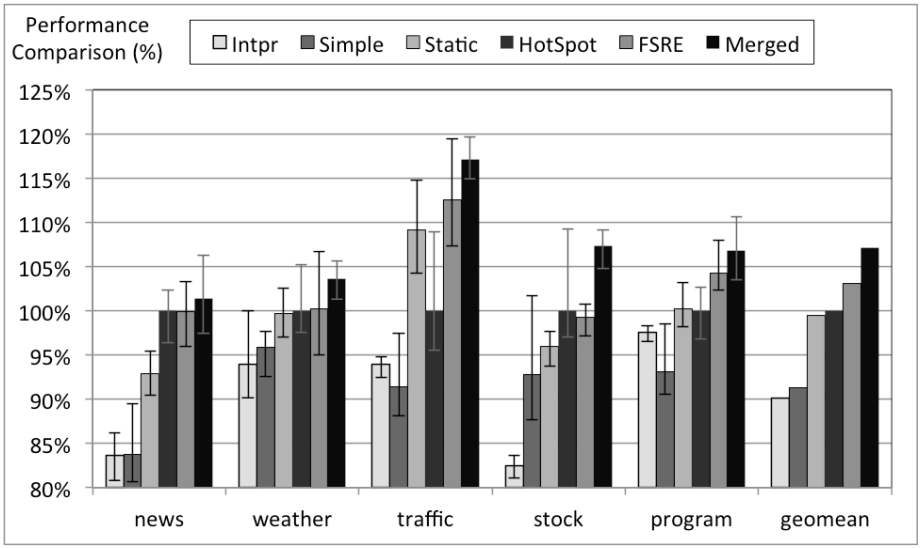

Figure 2-11. DTV Xlet performance ratio with the *HotSpot* performance as a basis of 100%.

The first thing to note in Figure 2-11 is that the performance improvement itself over interpretation is much smaller than the benchmark result in Figure 2-2. Even *Merged* achieves only 19% improvement for the Xlets, while it achieved more than 190% improvement for the benchmarks. To understand this behavior, we measure the number of executed methods and the number of compiled methods during the Xlet execution, and Figure 2-12 shows the ratio of compiled methods to executed methods. It also shows the number of executed methods in the parenthesis at the bottom of each menu item, which is around 4,000 methods. This is much larger than the benchmark results, where jess, for example, executes the biggest number of methods among the benchmarks, yet it is only 943. Moreover, the ratio in Figure 2-11 for the Xlets is around 3~6%, while the ratio in Figure 2-15 for the benchmarks is around 8~10%. This means that the Xlets execute a much larger number of methods, yet compile a smaller fraction of methods compared to the benchmarks. So, Xlet would spend much of its execution for interpretation, lowering the benefit of JITC, which appears to be one of the reason for its lower speedup of JITC.

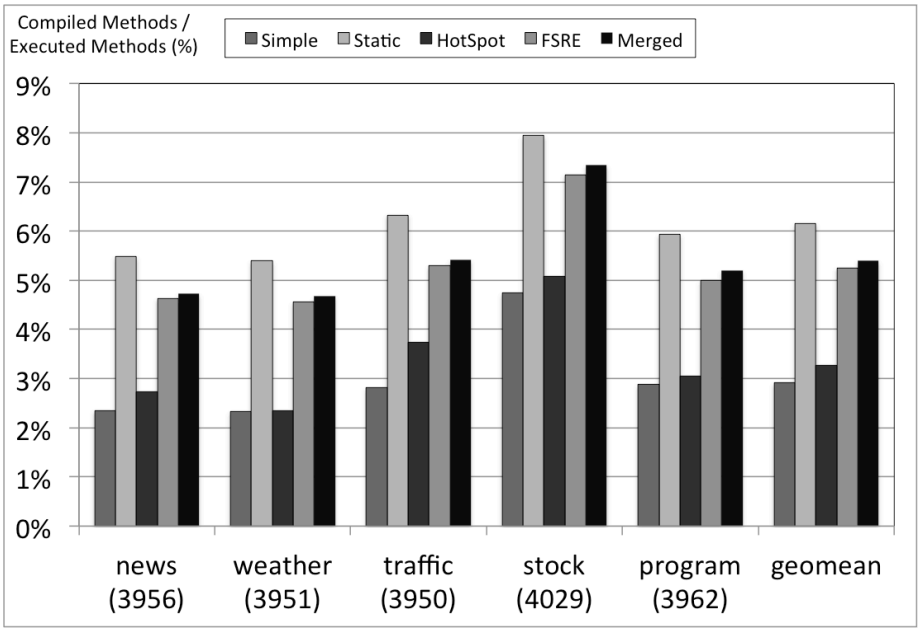

Figure 2-12. The ratio of compiled methods to all executed methods in DTV Xlet.

We also compare the execution profile of the Xlets to that of the benchmarks. We first measure the call count for each method to see the distribution of methods according to the call count. Figure 2-13 (a) and (b) show the cumulative distribution functions (CDF) for the benchmarks and the Xlets, respectively. For the benchmarks, around 30% of methods are called only once and 65% of methods are called fewer than 10 times. For the Xlet applications, 40% of methods are called only once and 70% of methods are called fewer than 10 times. Considering the much larger number of executed methods in the Xlets, cold spots in the Xlets would be more dominant than in the benchmarks, making the Xlets spend much time on interpretation, which supplement the above analysis.

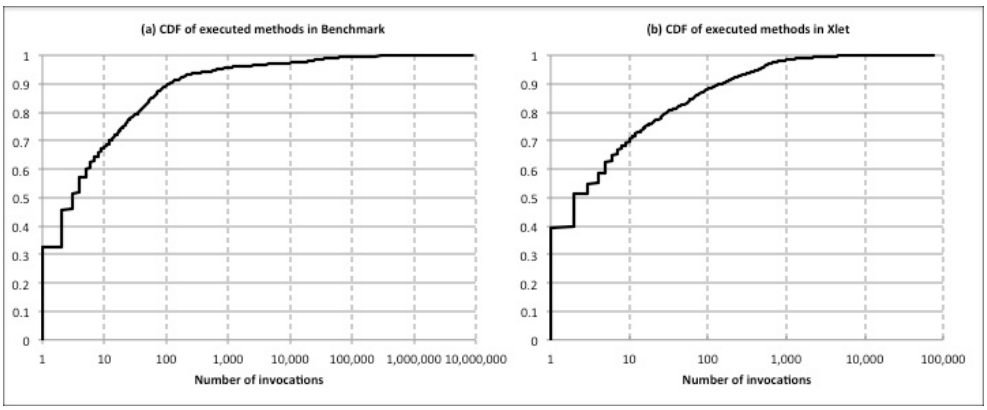

Figure 2-13. CDF of executed methods for their invocation counts in Benchmark and Xlet.

On the other hand, 5% and 3% of methods are executed more than 1,000 times and 10,000 times in the benchmarks, but less than 1% of methods are executed more than 1,000 times, and there are no methods executing more than 10,000 times in the Xlets. This means that real hot methods are rare in the Xlets. We also measure the number of times the backward branches are taken, and the number in the Xlet applications is 1/30 times the number in the benchmarks, meaning that the Xlet loops iterate far fewer than the benchmark loops. It appears that the Xlets have more cold or warm spots, but fewer hot spots, which would lead to a lower speedup of the JITC due to lower benefit of compilation.

These profile results has another important implication on the behavior of the Xlet. Previous researches indicate that running a benchmark once shows the start-up phase behavior of an application [8] where many methods for class loading and initialization are executed, but they are not executed heavily, making hot spot detection important [20, 26]. Our profile results show that in the Xlet application, methods are called fewer and loops iterate fewer than in the benchmarks, so the Xlets show more of a start-up phase behavior rather than a steady-state behavior. This means hot spot detection for the embedded application would also be important.

Another thing to note in Figure 2-11 is that the performance penalty of *Simple* and *Static* compared to *HotSpot* is much lower than that in the benchmarks. *Simple* and *Static* were much worse than *HotSpot* in the benchmarks (see Figure 2-12), yet even *Simple* achieves 90% of the *HotSpot* performance and *Static* achieves a

similar performance to *HotSpot* in the Xlets. This appears to be due to the smaller fraction of compiled methods in Xlets, which would make a wrong hotspot detection affects the performance much less. *FSRE* achieves an average of 3.1% improvement over *HotSpot*, mostly due to traffic and program. *Merged* obtains the best performance among all heuristics in all menu items, achieving an average of 7.1% improvement over *HotSpot*.

To understand the performance result, we measured some data based on the *dynamic bytecode size*, which means the total amount of all executed bytecode in bytes. This would be proportional to the interpretation time because it includes the overhead of handling opcodes and operands by the switch-case statements of the interpreter. For each compiled method, we measured (1) the dynamic size of the bytecode interpreted before it is compiled. Then, we measured (2) the dynamic size of the bytecode interpreted for the method when we run the Xlet application by the interpreter only. We computed the ratio (1)/(2) and took an average for all compiled methods by each heuristic. It would indicate how early a method is detected as a hot spot and compiled. Figure 2-14 shows the result. *FSRE* and *Merged* show a smaller ratio than *HotSpot*, roughly indicating that they compile methods earlier than *HotSpot*. *Static*'s ratio is also smaller than *HotSpot*'s, yet it does not help much since *Static* compiles too many methods as shown in Figure 2-12.

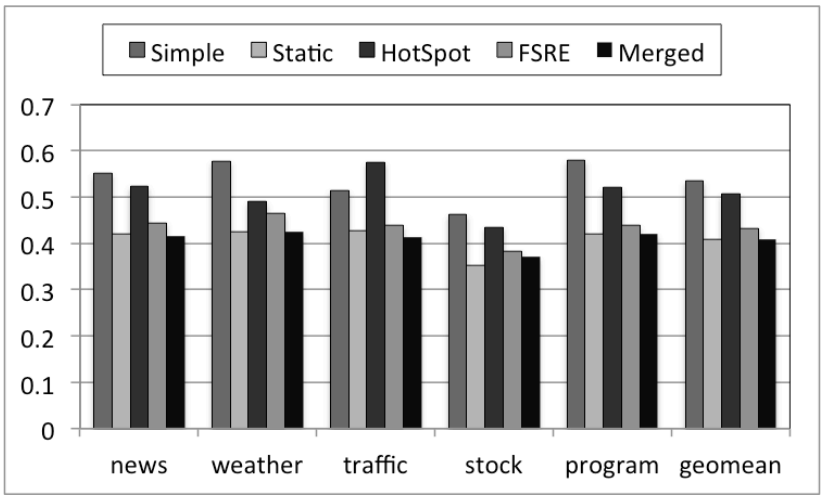

Figure 2-14. Ratio of dynamic bytecode size before compilation to dynamic bytecode size with interpretation only.

We also sum up (2) for all compiled methods, i.e., the dynamic size of bytecode interpreted when executed in the interpreter mode. We compared the size for each heuristic to that of *HotSpot,* and the ratio is shown in Figure 2-15, which would depict the relative execution coverage of the compiled methods by each heuristic compared to *HotSpot*. The graph shows that *FSRE* and *Merged* has a similar size to *HotSpot*, roughly indicating that they compile methods with a similar preciseness to *HotSpot*. *Static*'s size is smaller than *HotSpot*'s while it compiles much more methods. This seems to indicate that its hot spot detection is less precise than *HotSpot*. *Simple* indicates even worse preciseness.

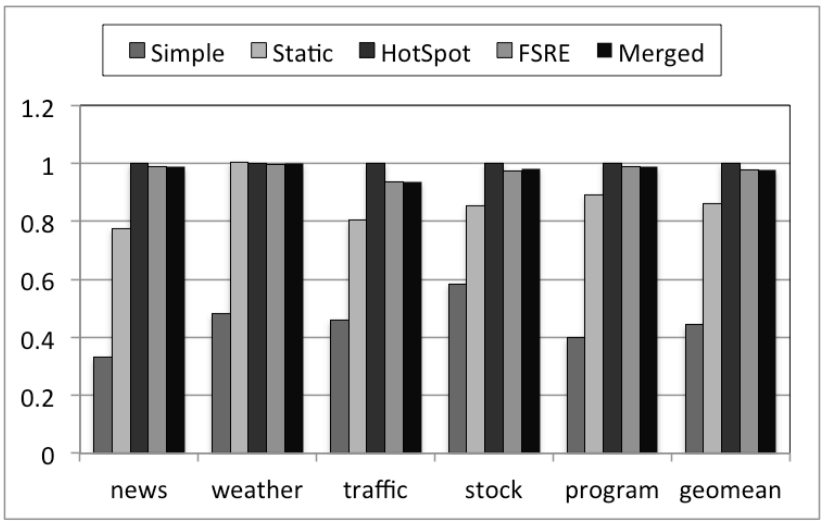

Figure 2-15. Ratio of dynamic size of bytecode interpreted for compiled methods compared to *HotSpot*.

We performed one more experiment in the DTV environment, regarding the preciseness of the static-FSRE when the weights of heavy bytecode instructions are considered additionally for the loop nests, as explained at the end of Section 4. Figure 2-16 shows the performance of this weighted static-FSRE with those additional weights, compared to the original static-FSRE. The C8 value in T/C3 * (size of *m*) < P[*m*] * C8 is reduced from five to three because otherwise the weighted static-FSRE compiles many cold methods, which degrades the performance by around 5%. Even if we change the C8 value, Figure 2-16 shows that there is an average of 3% performance degradation, which appears to be due to the additional computation overhead. Since the static-FSRE cannot be precise (it gives the same loop count for all loops and does not consider the forward branch or

early return), more precise estimation appears to be an overkill of cost without any definite benefit, so the current static-FSRE seems to be a right balance.

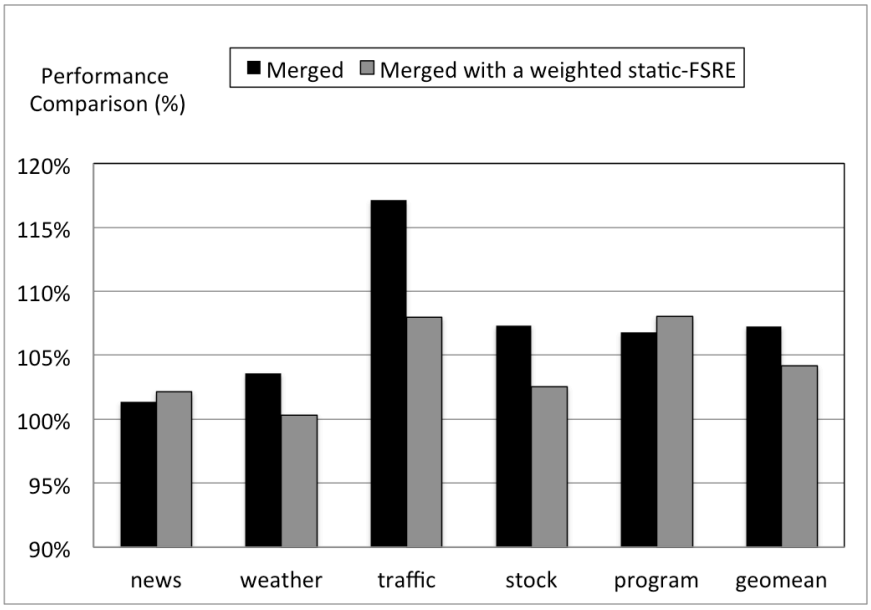

Figure 2-16. DTV Xlet performance for the weighted static-FSRE compared to the original static-FSRE.

Chapter 3. Code Size Optimization for JITC

3.1 JavaScript JITC in SFX and Thumb2

This section describes a brief background on the JavaScript programming language with its execution semantics and the SFX JavaScript engine with its JITC architecture. We also mention code size and performance for ARM and Thumb2.

3.1.1 JavaScript and Execution Semantics

JavaScript is an object-oriented scripting language for web client-side programming, primarily used for interactive and dynamic user interfaces for web sites [41]. JavaScript programs are often embedded in the HTML web pages and executed for launching pop-up windows or performing simple calculations, combined with the Document Object Model (DOM). These days, however, JavaScript performs more substantial computations, especially for implementing RIA using the Asynchronous JavaScript and XML (Ajax) and the XMLHttpRequest protocol, or for implementing the (mobile) widgets using the browser APIs.

As a programming language, JavaScript allows easy programming with its C-style syntax, but includes a few features that disallow efficient execution or compilation. First, JavaScript is a dynamically-typed language such that the type of a variable is determined at runtime and can be changed during execution. Similarly, JavaScript is an object-based language, yet not class-based but *prototype-based* such that objects can be created by cloning existing objects as a form of inheritance and the prototype of an object can also be changed during execution. Lack of type declaration or class declaration for an object would essentially make field accesses or methods lookup more inefficient than statically-typed, class-based languages as Java. Second, JavaScript functions are first-class, meaning that functions are objects which can be passed as arguments or return values and be assigned to variables. It also supports inner functions and a *closure*, a first-class function that uses free variables declared in its enclosing functions (scopes). Figure 3-1 (a) shows a closure example where f and dx are free variables. These free variables in a closure survive even after the functions in which they are declared finish, as long as the closure itself is still alive. This means that even if a function returns, all of its local variables cannot necessarily be de-allocated as in most languages, which

would somewhat complicate the conventional call stack-based scope implementation.

```
function derivative(f, dx) {
  return function (x) {return (f(x+dx) -
f(x))/dx;};
}
```
(a)

Bytecode for the function derivative()

```
enter_with_activation r0 // Make new activation &add to scope chain
new_func_exp      r1, f0 // Create a new object for closure
tear_off_activation   r0 // Copy the activation to scope chain
ret               r1     // Return the function object
```

Bytecode for the closure function

```
enter                          // start a function
// var at index -10 of the 0th scope is loaded to r0 (which is the function object f)
get_scoped_var     r0, -10, 0
mov                r1, null  // Set argument r1 null (this object)
// var at index -9 of the 0th scope is loaded to r3 (which is dx)
get_scoped_var     r3, -9, 0
add                r2, r-9, r3 // Set argument r2 by (x+ dx)
// Call f   (r0) with two arguments (r1, r2),   sliding 11 registers; r0 gets return
value
call               r0, r0, 2, 11
// var at index -10 of the 0th scope is loaded to r0 (which is the function object f)
get_scoped_var     r1, -10, 0
mov                r2, null // Set argument r2 null (this object)
mov                r3, r-9  // Set argument r3 by   x
// Call f   (r1) with two arguments (r2, r3),   sliding 12 registers; r1 gets return
value
call               r1, r1, 2, 12
sub                r0, r0, r1 // f(x+ dx) - f(x)
// var at index -9 of the 0th scope is loaded to r1 (which is dx)
get_scoped_var     r1, -9, 0
div                r0, r0, r1// (f(x+ dx) - f(x)) / dx
```

Figure 3-1. A JavaScript function and its SFX bytecode

3.1.2 SquirrelFish Extreme and the Bytecode

SquirrelFish Extreme (SFX) is a JavaScript engine included in the WebKit browser engine, which uses JITC. The previous version called SquirrelFish supported interpretation only. The initial SFX on the x86 platform employed a somewhat minimal JITC based on the context-threading interpreter, which is for improving the branch prediction hits for indirect branches executed during interpretation [44]. The recent SFX includes a more substantial JITC which translates more of JavaScript code to machine code.

SFX first translates the JavaScript code into an intermediate representation called the *bytecode*, and then translates the bytecode into machine code. The unit of translation is a JavaScript function, such that when a function is called for the first time, it is translated to the machine code, which is then executed thereafter when called. This is different from the V8 JITC which has no bytecode. It also differs from the TraceMonkey JITC whose translation unit is a hot path, not a function [37].

The architecture of the SFX includes two major data structures, a *sliding register file* and a *scope chain*. The sliding register file is similar to the conventional call stack and saves local variables including arguments and temporaries. When a function is called, the sliding *frame pointer* advances on the register file, and when it returns, the frame pointer backtracks. Meanwhile, the scope chain is a linked list of nodes, each representing a static scope for JavaScript functions such that every instantiated function object has a pointer to one of the nodes, which is the static scope of the function. For example, there is a node for the global scope, which will be pointed by all global-level functions. When a function that includes closures are invoked, SFX adds a new node to the scope chain, which will become the static scope for the closures and be pointed by the closure objects. An *activation* of the function composed of its local variables and symbol tables will be attached to the node such that the closures can access them even after the function returns. Figure 3-1 (b) depicts the bytecode streams of derivative() and its closure, which show some of this mechanism.

There are around 109 bytecode instructions defined for the SFX. Compared to the Java bytecode, it is register-based instead of stack based. They include instructions for creating a new object, for reading/writing registers, for performing arithmetic, for accessing properties of an object, for branching and looping, for calling functions defined in the JavaScript engine, for managing scopes involved with function calls, etc. Many of these bytecode instructions are involved with somewhat substantial execution semantics, so they are often handled by a call to a dedicated SFX function called *CTI functions* instead of being handled directly by the machine code generated by the JITC. That is, 51 bytecodes are always handled by the CTI functions. For 40 bytecodes, machine code is generated partially if the type of their operands allow, otherwise the CTI functions are called. The remaining

18 simple bytecodes are fully compiled to machine code.

3.1.3 SFX JITC Architecture

The WebKit open source community led by Apple recommends developers to follow its JITC code generation architecture for platform portability. When the JITC translates a bytecode to the machine code, it must call a pre-defined function called the *macro assembler* corresponding to the bytecode. It is actually a hierarchy of macro assemblers, and the JITC developer is allowed to modify only the body of the low-level macro assemblers for a given target architecture. For example, Figure 3-2 shows a macro assembler emit_op_bitor() for the bytecode op_bitor for ARM. The JITC must call this function to compile op_bitor, which will call other macro assemblers for generating the machine code. For this example, we first generate machine code for loading the two arguments from the SFX register file to the virtual registers by calling emitGetVirtualRegisters(). Then, we call orPtr() for the generation of the actual bitwise OR instruction (ORR in ARM). Finally, we generate the code for storing the result virtual register to the register file by calling emitPutVirtualRegister(). The virtual registers are globally mapped to physical registers as shown in Figure 3-2. Consequently, if the bytecode is op_bitor r-19, r-18, r-17, the following ARM code will be generated: ARM::r0=load@r-18; ARM::r1=load@r-17; ARM::r0= ORR ARM_r0 ARM_r1; store ARM_r0 @r-19;

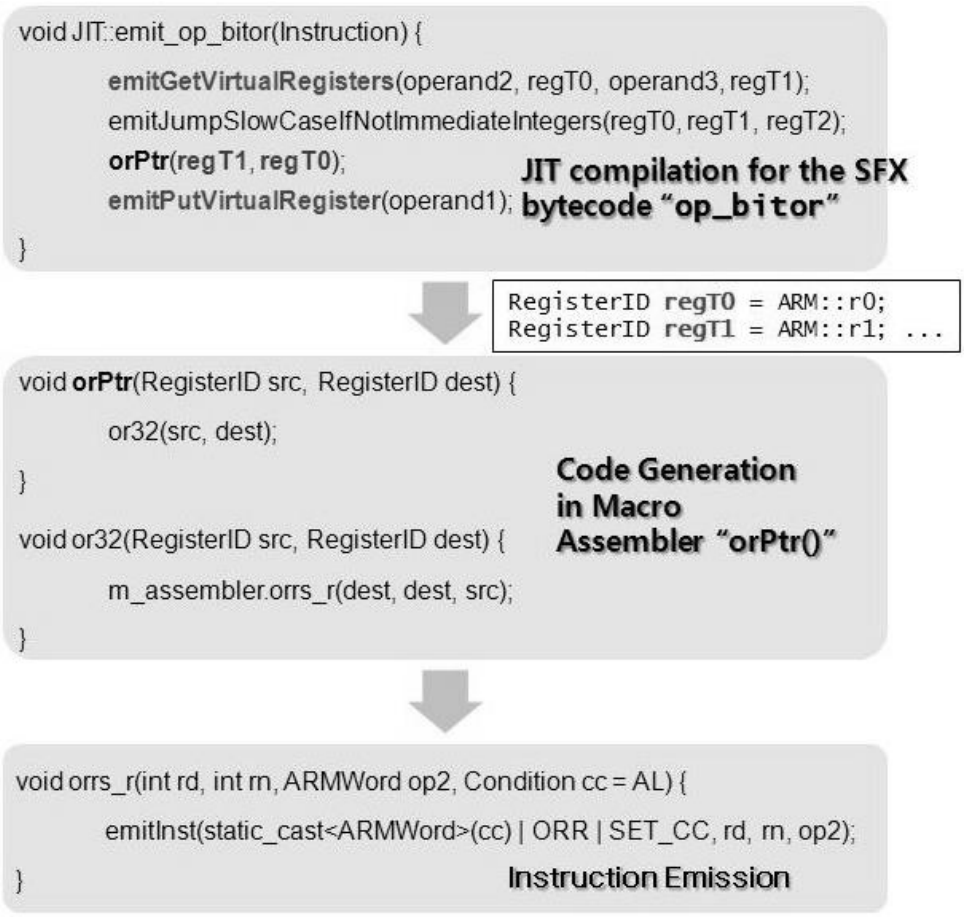

Figure 3-2. SFX macro assembler hierarchy for op_bitor

Although this code generation based on a fixed hierarchy of macro assemblers would accelerate the retargeting of the JITC for a new CPU platform, it restricts code optimization seriously. That is, each bytecode is translated separately from other bytecode, which makes the execution of its machine code keep the same states for the SFX data structures exactly as when the bytecode is interpreted. So no optimized code generation beyond bytecode boundaries is allowed. Even for each bytecode, there is little leeway for optimizing the machine code since the assembler hierarchy decides the instruction sequence and we can modify only the body of some leaf assemblers. Also the virtual registers are globally mapped to fixed physical registers with no need for separate register allocation. Many complex bytecodes are translated to direct calls to CTI functions as mentioned above, so there is no much room for optimizations around them, either.

3.1.4 JITC Code Generation for Thumb2

Our target CPU for the SFX JITC is based on ARMv7 architecture which includes Thumb2 ISA [38]. Thumb2 has 16-bit instructions as well as 32-bit instructions. It is aimed at improving the performance of the Thumb ISA which has only 16-bit instructions, thus suffering seriously from performance degradation; it is known that Thumb leads to a code size reduction of 30%, yet a performance degradation of 20% compared to ARM [42]. Thumb2 makes a compromise between the code size and the performance such that it is claimed to achieve a code size reduction of 25% and a performance degradation of 2~3% compared to ARM [42].

The above claim is for native code generation with a static compiler. For the case of the JITC of virtual machines, the JITC overhead is a part of the running time, so the JITC cannot perform full optimizations during its dynamic code generation. This may lead to a worse result of code size and performance. For example, the Thumb2 code generated by the Java VM JITC achieves a code size reduction of 15% and a performance degradation of 6%, compared to the ARM [39].

Our proposed SFX JITC for Thumb2 will have the same optimization issues as the Java VM JITC due to the JITC overhead. And what is worse, we need to follow the macro assembler hierarchy for the JITC code generation, which would reduce the optimization opportunities further. Our challenging goal is achieving code size and performance results comparable to the native Thumb2 compilation, by fully exploiting any remaining optimization opportunities.

3.2 SFX JITC Optimizations for Thumb2

This section describes our SFX JITC implementation for Thumb2 with code size reduction and performance optimizations.

3.2.1 Code Generation with Register Re-map

We start with the code base of the SFX JITC for the ARM ISA (*SFX-on-ARM*) and convert it to the JITC for the Thumb2 ISA (*SFX-on-Thumb2*). The SFX-on-ARM JITC strictly follows the macro assembler hierarchy discussed in Section 2.3. In order to generate smaller code using Thumb2 instructions, we simply generate the code in the following preference order. We first try to replace a 32-bit ARM instruction by an equivalent 16-bit instruction as much as possible. If this is not applicable, we try to replace it by an equivalent 32-bit Thumb2 instruction. For example, an ARM instruction ANDS R0, R0, R2 can be replaced by a single 16-bit

Thumb2 instruction, but an ARM instruction ANDS R0, R0, #0x1 cannot be and should be replaced by a 32-bit Thumb2 instruction. There are even cases where an ARM instruction cannot be replaced by a single 32-bit Thumb2 instruction. For example, some ARM instruction allows conditional execution but such field cannot be encoded in the Thumb2 instruction. So, a condition evaluation instruction (IT) which decides whether or not to execute the following instructions must be generated additionally, requiring more than one instruction, possibly more than 32-bits in size.

As to the register allocation for the generated code, there is no leeway in optimizing register assignment since the mapping of virtual registers to physical registers is already decided globally. The only thing we can do is changing the register map so as to facilitate the generation of 16-bit instructions. For example, SFX-on-ARM uses r0~r8 mapped to virtual registers in its macro assemblers, but the 16-bit Thumb2 instructions can include r0~r7 due to its 3-bit register field, so a macro assembler whose virtual register is mapped to r8 must generate 32-bit Thumb2 instructions.

In order to generate more 16-bit instructions, we need to update the register mapping. We analyzed the usage of the mapped registers and found that the virtual register mapped to r5 is used for a special-purpose of checking the time-out of the JavaScript code, thus not being used frequently. On the other hand, the pseudo register mapped to r8 is used as a general-purpose register, thus being utilized heavily. So, we swapped the mapping of both registers, which led to generation of more 16-bit instructions.

3.2.2 Constant Pool Aggregation

Unlike CISC machines such as x86, an address constant or a big constant cannot be fit into a single ARM instruction. So we need to generate code to assign such a big constant to a register. One approach is using multiple instructions for setting high-order bits and low-order bits of the register separately. The other approach is inserting a constant pool (CP) in the code to save the constant, and a PC-relative load instruction is used to read it to a register, as shown in Figure 3-2 (a). SFX-on-ARM takes this approach.

Although the CP-based approach can reduce the number of generated instructions, the instruction area and the data area are intermixed as in Figure 3-3 (a). This might

affect the cache behavior negatively, and more seriously, unconditional branches need to be inserted to jump over the CP, affecting both the performance and the code size. Our SFX-on-Thumb2 would suffer more from this problem since the 16-bit instructions are likely to generate more CPs due to their shorter immediate fields.

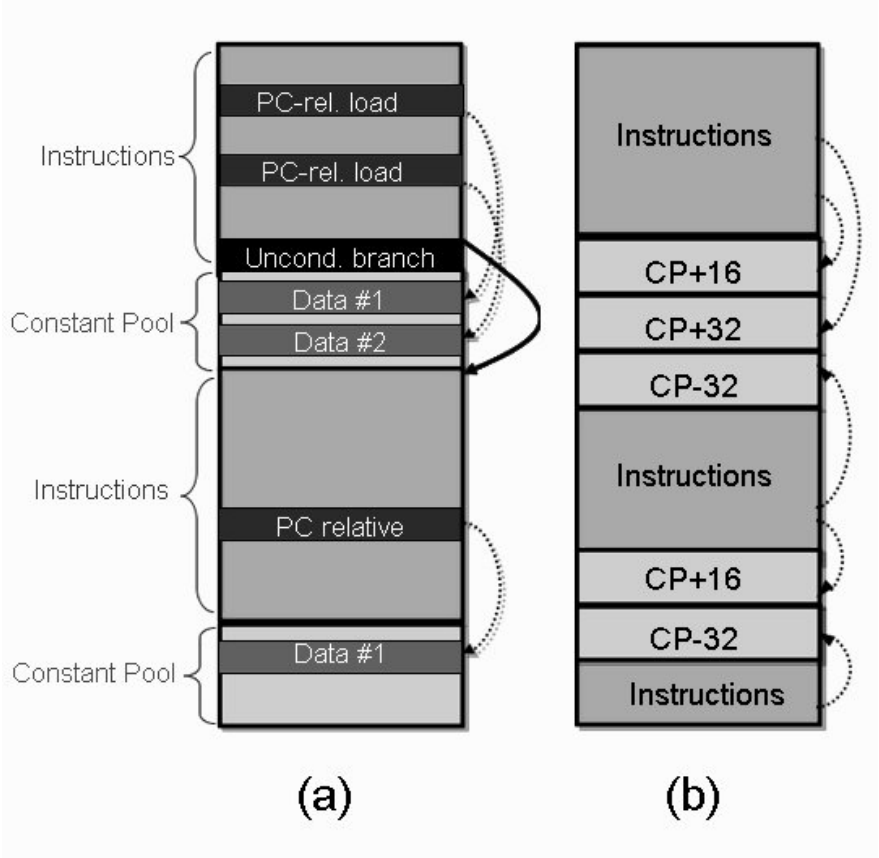

Figure 3-3. Constant Pool Aggregation

In order to mitigate this problem, we try to reduce the number of CPs by merging otherwise separate CPs into a single CP although the total size of the CPs remains the same. This might reduce unconditional branches and improve the cache behavior. Our idea is creating a larger CP that can be referenced with a negative offset as well as a positive offset, and with an offset in the 16-bit instructions and an offset in the 32-bit instructions.

Before we describe the CP generation algorithm, we should mention that we normally generate 16-bit PC-relative loads for loading the big constants but we also need to generate 32-bit PC-relative loads if the target register is PC or SP since

these registers are located beyond r7 (r14 and r15, respectively), hence accessible by 32-bit instructions only. Also, the 32-bit PC-relative load in the Thumb2 ISA allows a negative offset as well as a positive offset (the ARM ISA also allows both but the SFX-on-ARM utilizes only the positive offset), while the 16-bit PC-relative load allows only a positive offset. This allows us to distinguish a CP into three types: a CP accessible with a positive offset of 16-bit instructions (CP+16), a CP accessible with a positive offset of 32-bit instructions (CP+32), and a CP accessible with a negative offset of 32-bit instructions (CP-32). Our idea is merging them into a single CP to achieve a larger one, as shown in Figure 3-3 (b).

Our data structures for code generation are as follows. There is an instruction buffer, a CP buffer for 16-bit instructions (CP16), and a CP buffer for 32-bit instructions (CP32), with a flag indicating if the CP32 is for a positive offset (+) or for a negative offset (-). The flag is initialized to +. We fill the three buffers as the algorithm proceeds, and when we find the distance between any load and its constant is out of its offset range, we flush the buffers to the *code buffer*, which has the final stream of instructions and the CPs, and continue the algorithm. Our algorithm flushes to the code buffer in a unit of (a) [instructions, CP+16, CP+32], or (b) [CP-32, instructions, CP+16], or (c) [CP-32]. This will lead to two CP patterns: {CP+16, CP+32, CP-32} or {CP+16, CP-32} (see Figure 3-3 (b)). When the JITC for a function completes, the code buffer will be copied in the *code cache* of the SFX engine.

The algorithm proceeds as follows. The JITC generates new instructions to the instruction buffer one by one. When we need to generate a constant loaded by instruction L, we add the constant to the CP32 if it is a 32-bit load and to the CP16 if it is a 16-bit load. If the first 16-bit load L' in the instruction buffer becomes out-of-offset due to the insertion of L (which can be determined by the distance between L and L'), we flush in a unit of (a) if the flag is +, in a unit of (b) if the flag is -. We then switch the flag. Similarly, if the flag is + and if the first 32-bit load L" in the instruction buffer becomes out-of-offset due to the insertion of L (which can be determined by the distance between L and L", plus the current size of CP16), we flush in a unit of (a) and set the flag -. On the other hand, if the flag is – (meaning that the 32-bit loads are saving their constants in CP-32) and the size of the instruction buffer is out-of-offset for a 32-bit load, we flush CP32 in a unit of (c)

and set the flag +. These will lead to the desired larger CPs.

3.2.3 Patching PC-relative Branches

The ARM ISA allows two types of branches: a PC-relative branch and a load-based branch. The former is a conventional branch (i.e., PC = PC+offset). The latter is based on loading the target address constant saved at the CP to the PC using the PC-relative load (i.e., PC = load @(PC+offset)), as discussed in Section 3.2. The PC-relative branch takes a single cycle, while the load-based branch takes two cycles and requires one more word [40].

SFX-on-ARM generates branches as follows. It first generates load-based branches only whenever a branch is needed. When it is found later during linking that the distance between a branch and its target is within the offset range of a PC-relative branch, the branch is patched by a PC-relative branch for better performance. This is illustrated in Figure 3-4. The reason why SFX-on-ARM generates load-based branches first is related to its single-pass code generation discussed in Section 3.2. Both the instruction buffer and the CP buffers are flushed to the code buffer as soon as they need to be flushed. So when a branch is flushed, it is not clear if the branch target will be located within the PC-relative branch offset or not. It would be risky to generate a PC-relative branch optimistically because if the branch target is later found to be out of the offset range during linking, we cannot easily patch it by a load-based branch because there might be no CP reachable with its offset and creating a new CP for its target address or adding the target address in an existing CP might affect all the addresses or offsets generated so far. For those branches that jump to the CTI functions, the branch offset can be decided only after the code buffer is copied to the code cache, so generating PC-relative braches for them is also risky. Therefore, generating load-based branches first and patching them later by PC-relative branches appears to be a simple and reasonable approach.

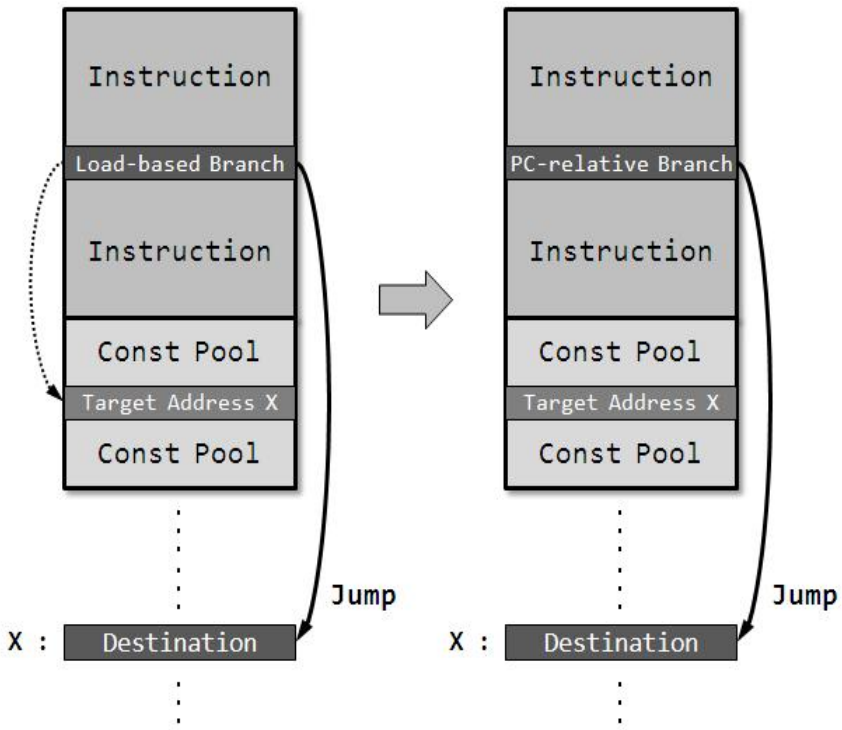

Figure 3-4. Load-to-Branch Patch in SFX-on-ARM

There is one problem, though. For the patched branches, their corresponding CP entries are useless, wasting the code space. We cannot remove them at this point to condense the CP since it would require changing the address offsets in all PC-relative load instructions. In order to reduce the code size, our SFX-on-Thumb2 takes a different approach to branch generation.

We generate branches in a way opposite from SFX-on-ARM. We first generate PC-relative branches only whenever a branch is needed. When it is found later during linking that the distance between a branch and its target is out of the offset range, the branch is patched by a load-based branch. The problem is where to insert the target address in the CP without affecting the address offset in any PC-relative load instructions. One perfect location to insert the address constant is the area between the plus-offset CP region and the negative-offset CP region of any CP reachable within the offset of the load-based branch (such an area exists in both CP patterns generated by our CP aggregation). Inserting into that area cannot affect the address offsets of any load instructions because the relative distance between the loads and the corresponding CP entries remains the same even after the insertion.

This is illustrated in Figure 3-5. Consequently, no useless entries will exist in the CPs and we can save the CP space by as many as the number of un-patched PC-relative branches. And this optimization was possible because of CP aggregation.

There is one exception, though. For those PC-relative branches to the CTI functions, we must insert CP entries before the code buffer is copied to the code cache since adding new CP entries in the code cache is difficult. So, even if the branch remains as PC-relative branch because the target address is within its offset range, the added CP entry is wasted (these branches are designated PC-relative-CP in our experimental results in Section 4.2).

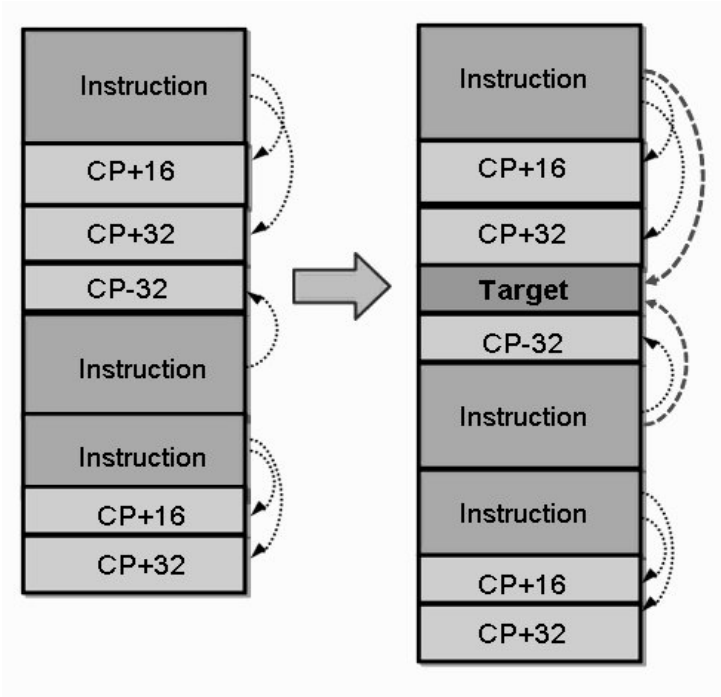

Figure 3-5. Target Address Insertion in SFX-on-Thumb2

One problem is when no CP is located within the offset of the load-based branch. In this case, we replace the load-based branch by a branch to the end of the function while adding the load-based branch followed by its target address at the end of the function, as shown in Figure 3-6. This adds one more instruction to the code and increase the running time by one cycle, but fortunately, this rarely occurs according to our experiments (see Section 4.2).

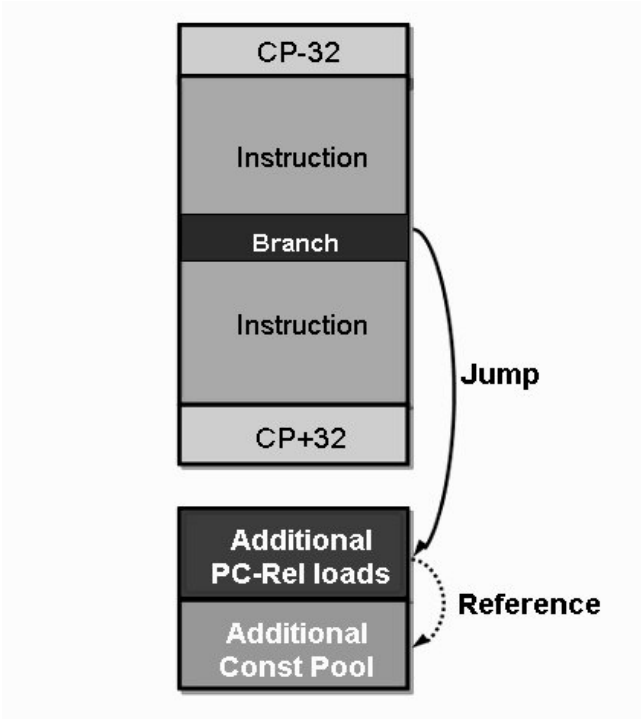

Figure 3-6. Two-level jump in SFX-on-Thumb2

3.3 Experimental Result

We evaluate SFX-on-Thumb2 with the proposed optimizations, compared to the existing SFX-on-ARM on the same platform. This will reveal how the code size and the performance are affected, when we move from ARM to Thumb2 in the SFX JITC.

3.3.1 Experimental Environment

Our experimental hardware is a beagle board employing the Cortex-A8 CPU based on the ARMv7 architecture, which is similar to what the latest smart phones are employing currently [40]. It has a 32KB I-cache/D-cache. The platform is equipped with the Linux 2.6 and the SFX in the Webkit-r44282 for ARM (SFX-on-ARM). We implemented SFX-on-Thumb2 and compared to SFX-on-ARM. We experimented with the stand-alone SFX, not with the one in the browser. Our benchmarks are the *SunSpider* benchmark which is composed of 26 JavaScript programs [50]. For evaluation we measure the total size of JITC generated code and the total running time for all programs.

3.3.2 Code Size Result

Figure 3-7 depicts the code size result. It shows the code size of the SFX-on-ARM

as 100% (leftmost bar) and compares SFX-on-Thumb2 with each optimization in Section 3 enabled cumulatively (right bars). The graph shows instruction area result and the CP area result separately, with their sum result at the end.

For the instruction area, when we generate the Thumb2 instructions instead of the ARM instructions, the code size is reduced by 14% due to its 16-bit instructions (Thumb2 in Figure 3-6). When we swap register r5 and r8 as described in Section 3.1, we could achieve an additional 9% size reduction (Thumb2-swap in Figure 3-6) due to additional replacement of 32-bit instructions by 16-bit instructions. Compared to the ARM code, we found that 46% of 32-bit instructions are replaced by 16-bit instructions. Most of the remaining 32-bit instructions are load-based branches to the CTI functions or those that do not have equivalent 16-bit instructions such as TST.

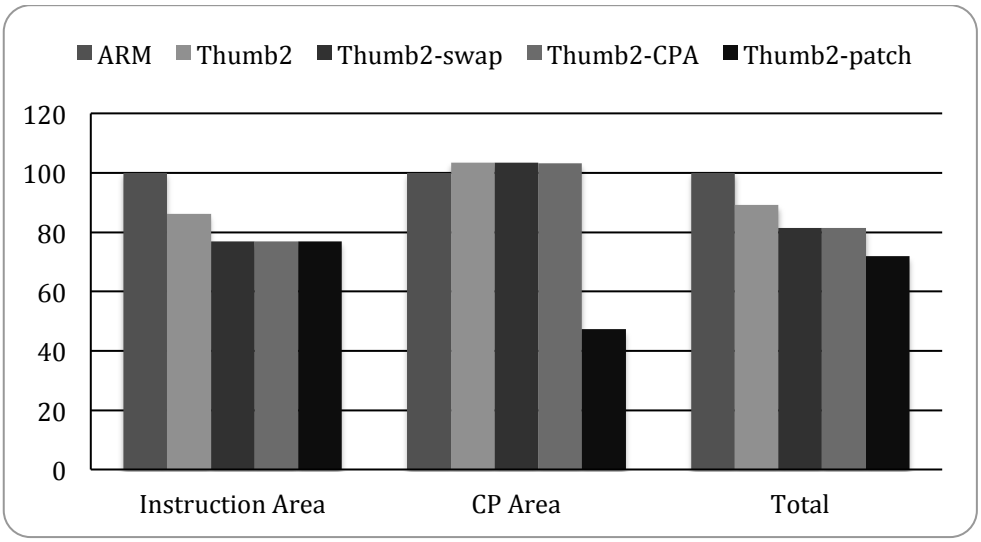

Figure 3-7. Code size comparison

For the CP aggregation optimization, there is no tangible impact on the instruction area size (Thumb2-CPA), although we found that the number of CPs is reduced from 746 to 352. This will reduce the number of unconditional branches by 394, yet it takes a tiny portion of the total number of instructions generated, hence affecting little. Branch patching in Section 3.3 is not supposed to affect the instruction count, so there is no change (Thumb2-patch). Consequently, SFX-on-Thumb2 can reduce the size of the instruction area of SFX-on-ARM by 23%.

For the CP area, there is a slight increase of the size as we move from ARM to Thumb2 in Figure 3-7, due to the shorted immediate fields in the 16-bit Thumb2 instructions. For example, a constant 0xF000000F can be encoded to 0x2FF and fit in the immediate field of an ARM instruction, while such an encoding is impossible for the Thumb2 immediate field. So, the constant should be saved in the CP and we need a load instruction to read it into a register. However, if we employ the branch patching, the size of the CP area is reduced by half, due to the removal of unnecessary CP entries. In total, the code size is reduced by 29% in Figure 3-7.

Figure 3-8 depicts the distribution of branches after the branch patching is applied. PC-relative is those which remain as PC-relative branches even after the patching. PC-relative-CP is those PC-relative branches to the CTI functions that have useless CP entries as discussed in Section 3.3. load-patched is those branches which are patched by load-based branches since the distance between them and their target addresses are out of the offset. Finally, load-after-branch is those branches which do not have any nearby CP entries when they should be patched to load-based branches, so a branch to the function end and a load-based branch with a CP entry are added.

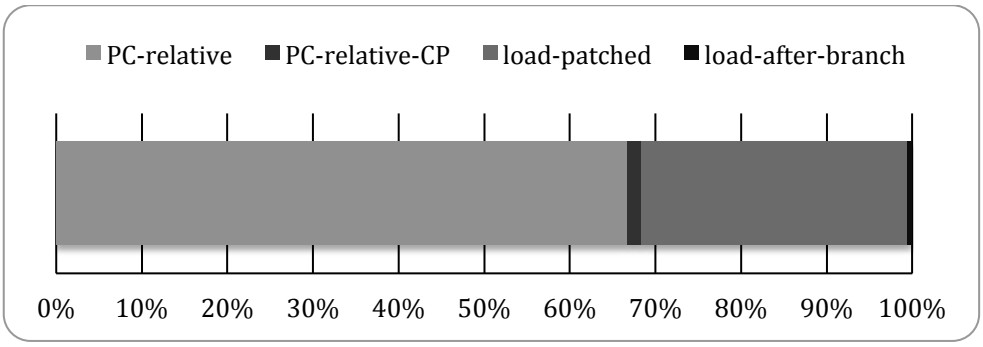

Figure 3-8. Component Ratio of Branches in Executable

PC-relative does not use the CP, while PC-relative-CP and load-patched require 4-byte CP entries. load-after-branch uses 8-byte CP entries (4-byte PC-load and 4-byte PC entry). Compared to the branch generation approach of SFX-on-ARM, we can reduce the CP area by as many as the number of PC-relative while we increase the CP area by as many as the number of load-after-branch. As we can see in Figure 3-7, 67% of branches are PC-relative while PC-load-after-branch takes only

0.54%. So this leads to the 53% reduction of the CP area in Figure 3-6. Actually, PC-load-after-branch can be observed only in three benchmarks in Sunspider, so its overhead would be tiny.

3.3.3 Performance Result

Figure 3-9 shows the total running time of the Sunspider for the SFX-on-Thumb2 configurations, with SFX-on-ARM as 100%. As we generate the Thumb2 instructions, there is a running time increase of around 3.5%. Since we need to replace some ARM instruction by multiple Thumb2 instructions, there should be a cycle increase. Other optimizations lead to an additional running time increase by 1%, which is not significant.

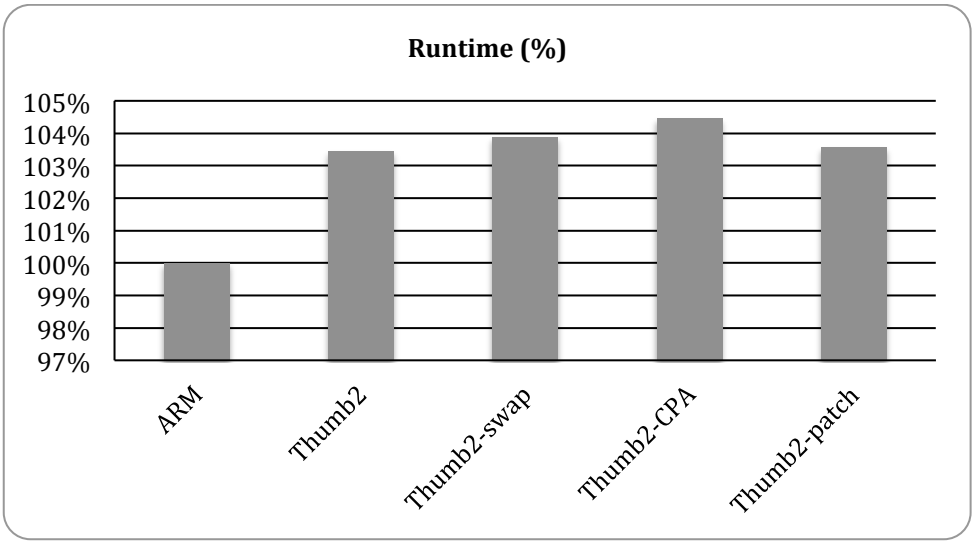

Figure 3-9. Runtime Comparison

Thumb2-CPA did not improve the running time even though it would lead to less execution of unconditional branches and better cache behavior. Since the cache size of Cortex-A8 is large enough, the cache impact would occur only when the cache pressure is higher, such as running substantial JavaScript programs in web pages or RIA applications on top of a browser with others.

Overall, SFX-on-Thumb2 increases the running time of SFX-on-ARM only by 3.5%, which is competitive to the native Thumb2 result and much better than the JVM JITC result.

Chapter 4. Selective JITC for Web Page JavaScript

4.1 JavaScript and SFX JITC

This section provides a brief background on JavaScript programming language with its use as a web-client program. We then give an overview of the SFX JavaScript engine with its JITC. Finally, we review the web page JavaScript behavior.

4.1.1 JavaScript and Interaction with DOM

JavaScript is a de facto standard scripting language for web client-side programming, primarily used for interactive user interfaces for web pages [2]. JavaScript programs are embedded in the HTML web pages and executed for launching pop-up windows or performing simple calculations, combined with the Document Object Model (DOM). These days, however, JavaScript performs more substantial computations, especially for implementing RIA using the Asynchronous JavaScript and XML (Ajax) and the XMLHttpRequest protocol, or for implementing the (mobile) widgets using the browser APIs.

As a programming language, JavaScript allows easy programming with its C-style syntax, but includes a few features that disallow efficient execution or compilation. First, JavaScript is a dynamically-typed language such that the type of a variable is determined at runtime and can be changed during execution. Similarly, JavaScript is an object-based language, yet not class-based but *prototype-based* such that objects can be created by cloning existing objects as a form of inheritance and the prototype of an object can also be changed during execution. Lack of type declaration or class declaration for an object would essentially make field accesses or methods lookup more inefficient than statically-typed, class-based languages such as Java. Second, JavaScript functions are first-class, meaning that functions are objects which can be passed as arguments or return values and be assigned to variables. It also supports inner functions and a *closure*, a first-class function that uses local variables declared in its enclosing functions (scopes).

Figure 4-1 shows an example of an HTML web page embedded with JavaScript code. A web browser allocates a global object called a window, which manages all the contents displayed for a downloaded HTML page. The window has a property (which corresponds to a field of an object) called a document, which also has

properties specified in the HTML page such as texts, images, etc., and they form a tree structure. The format and structure of the document object has been standardized with the DOM. The window object and document object can be referenced in JavaScript program, as shown in Figure 4-1.

The HTML parser in the browser scans through the downloaded HTML page and builds the DOM tree of the properties for the document. Each component of the HTML page is separated by the *tag* such as <head> or <title>, so the parser can identify each component and its hierarchy, which is then added to the tree. When a <script> tag is encountered, JavaScript code within it is executed by JavaScript engine. In Figure 4-1, JavaScript execution will start from window.onload=function(){}, which is supposed to be executed when the page is loaded. A property belonging to a tree can be changed or a new one is added to the tree via JavaScript execution using the DOM APIs such as createElement(), createTextNode(), and getElementByID() in Figure 4-1. When the parsing is done, JavaScript execution is also done, so the DOM tree is finalized and displayed by the browser.

Figure 4-1 shows an example of manipulating the DOM tree by JavaScript using its features of closure or prototype [63]. The function object Tune() defines a property concat() using a closure which accesses the two variables, title and artist, defined in Tune(). The happySong object created by cloning (inheriting) the Tune object will also have the concat() property which returns a string composed of the title (“Putting on the Ritz”) and the artist (“Ella Fitzgerald”) with a space. A new property addCategory() is added to the prototype of the object Tune(), which defines a new property, category. This allows the happySong object to inherit addCategory() and category, so that its category value is set by “Swing” in the statement happySong.addCategory("Swing");. Now, the variable song is initialized with a string using the two properties of the happySong.

The remaining JavaScript is for adding the string to the DOM tree. We first create a paragraph node p. Then we create a text node txt with the string of song, which becomes a child node of p. Finally, we access the node corresponding to the logical block whose ID is “song” located at the end of the HTML page, using getElementByID().We then add p as its child. The document object manipulated in this way by JavaScript will have the tree structure in Figure 4-1 when the parsing is

done, and will be displayed in the browser window screen as shown.

We can also make JavaScript execution manipulate the DOM tree more dynamically based on the runtime status. For example, JavaScript executes differently depending on the commercials on the web page which may differ depending on the loading time.

```
<!DOCTYPE html>
<head>
<title>Tune Object</title>
<script>
function Tune(song,artist) {
  var title = song;
  var artist = artist;
  this.concat = function() {
    return title + " " + artist;
  }
}

window.onload=function() { // Starting point

  var happySong = new Tune("Putting on the Ritz",
                           "Ella Fitzgerald");

  // extend the object
  Tune.prototype.addCategory = function(categoryName) {
    this.category = categoryName;
  }

  // add category
  happySong.addCategory("Swing");

  // print song out to new paragraph
  var song = "Title and artist: " + happySong.concat() +
             " Category: " + happySong.category;

  var p = document.createElement("p");
  var txt = document.createTextNode(song);
  p.appendChild(txt);
  document.getElementById("song").appendChild(p);
}
</script>
</head>
<body>
<h1>Tune</h1>
<div id="song">
</div>
</body>
</html>
```

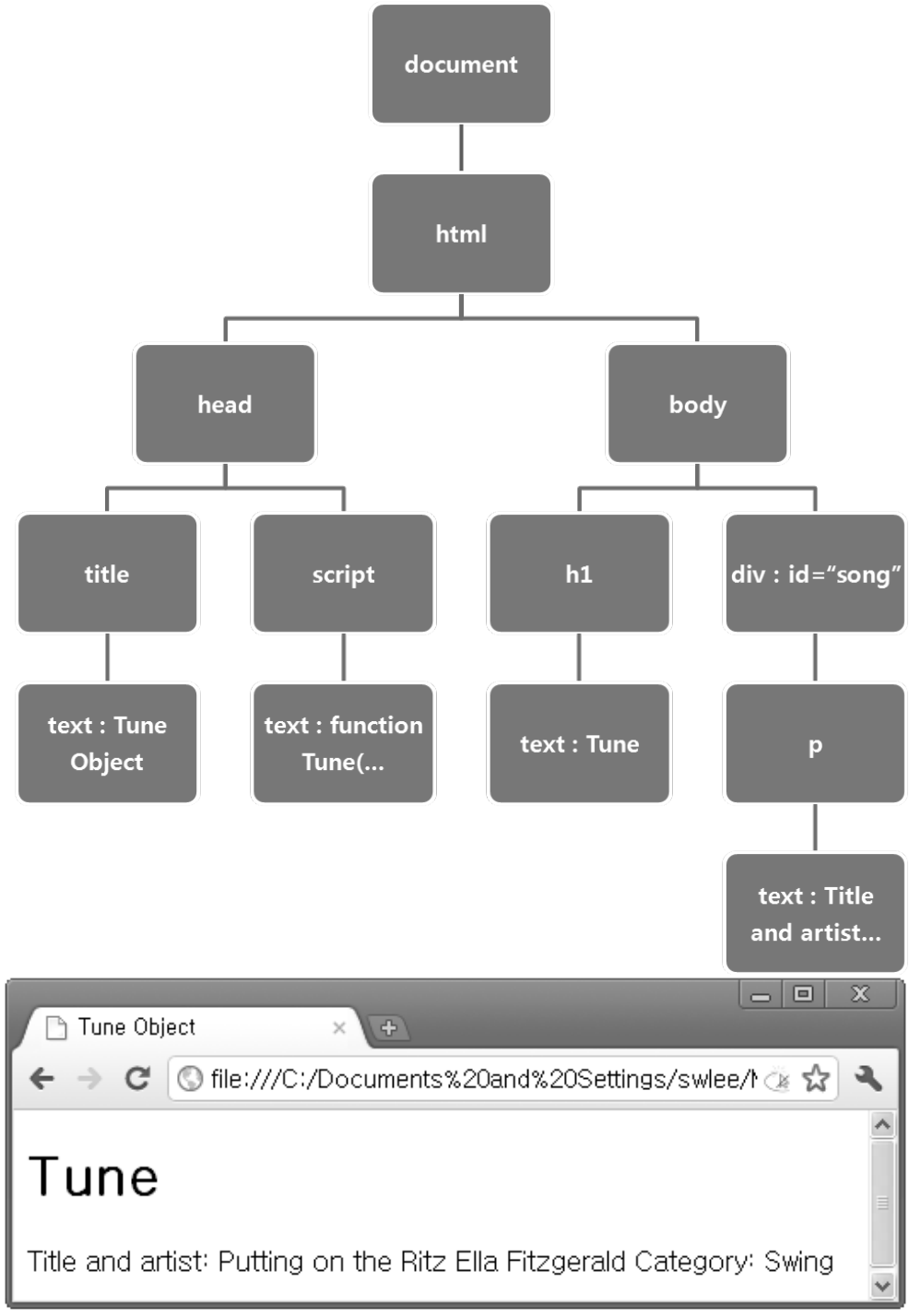

Figure 4-1. An example JavaScript and the window screen.

4.1.2 SFX JITC and Its Architecture

SquirrelFish Extreme (SFX) is a JavaScript engine included in the WebKit browser engine, which employs JITC. The previous version called SquirrelFish supported

interpretation only. The initial SFX on the x86 platform used a somewhat minimal JITC based on the context-threading interpreter, which is for improving the branch prediction hit ratio for indirect branches during interpretation [44]. The recent SFX includes a more substantial JITC which translates more of JavaScript code to machine code.

SFX first translates JavaScript code into an intermediate representation called the *bytecode*, and then translates the bytecode into machine code. The unit of translation is a function, such that when a function is called for the first time, it is translated to the machine code, which is then executed thereafter when called. This is different from the V8 JITC which has no bytecode. It also differs from the TraceMonkey JITC whose translation unit is a hot path, not a function [37].

There are around 109 bytecode instructions defined for the SFX. Compared to the Java bytecode, it is register-based instead of stack based. They include instructions for creating a new object, for reading/writing registers, for performing arithmetic, for accessing properties of an object, for branching and looping, for calling functions defined in JavaScript engine, for managing scopes involved with function calls, etc. Many of these bytecode instructions are involved with somewhat substantial execution semantics, so they are often handled by a call to a dedicated SFX function called *CTI functions* instead of being handled directly by the machine code generated by the JITC. That is, 51 bytecodes are always handled by the CTI functions. For 40 bytecodes, machine code is generated partially if the type of their operands allow, otherwise the CTI functions are called. The remaining 18 simple bytecodes are fully compiled to machine code. This means that the performance improvement opportunity using the SFX JITC would be somewhat lower than in the Java VM JITC, which generates machine code for most bytecodes (this is also true for the V8 JITC because it generates many calls to the functions of the V8 engine instead of fully generating machine code) [56].

Another feature of the SFX JITC that can lower the performance opportunity is the quality of the generated machine code. The WebKit [50] open source community led by Apple recommends developers to follow its JITC code generation architecture for platform portability. When the JITC translates a bytecode to the machine code, it must call a pre-defined function called the *macro assembler* corresponding to the bytecode. It is actually a hierarchy of macro assemblers, and

the JITC developer is allowed to modify only the body of the low-level macro assemblers for a given target architecture. For example, Figure 4-2 shows a macro assembler emit_op_bitxor() for the bytecode op_bitxor for ARM. The JITC must call this function to compile op_bitxor, which will call other macro assemblers for generating the machine code. For this example, we first generate machine code for loading the two arguments from the SFX register file to the virtual registers by calling emitGetVirtualRegisters(). Then, we call xorPtr() for the generation of the actual bitwise XOR instruction (EOR in ARM). Finally, we generate the code for storing the result virtual register to the register file by calling emitPutVirtualRegister(). The virtual registers are globally mapped to physical registers as shown in Figure 4-2. Consequently, if the bytecode is op_bitxor r-19, r-18, r-17, the following ARM code will be generated: ARM::r0=load@r-18; ARM::r1=load@r-17; ARM::r0= EOR ARM_r0 ARM_r1; store ARM_r0 @r-19;
Although this code generation based on a fixed hierarchy of macro assemblers would accelerate the retargeting of the JITC for a new CPU platform, it restricts code optimization seriously. That is, each bytecode is translated separately from other bytecode, which makes the execution of its machine code keep the same states for the SFX data structures exactly as when the bytecode is interpreted. So no optimized code generation beyond bytecode boundaries is allowed. Even for each bytecode, there is little leeway for optimizing the machine code since the assembler hierarchy decides the instruction sequence and we can modify only the body of some leaf assemblers. Also the virtual registers are globally mapped to fixed physical registers with no need for separate register allocation. Many complex bytecodes are translated to direct calls to CTI functions as mentioned above, so the code quality would be generally worse than in Java VM JITC.

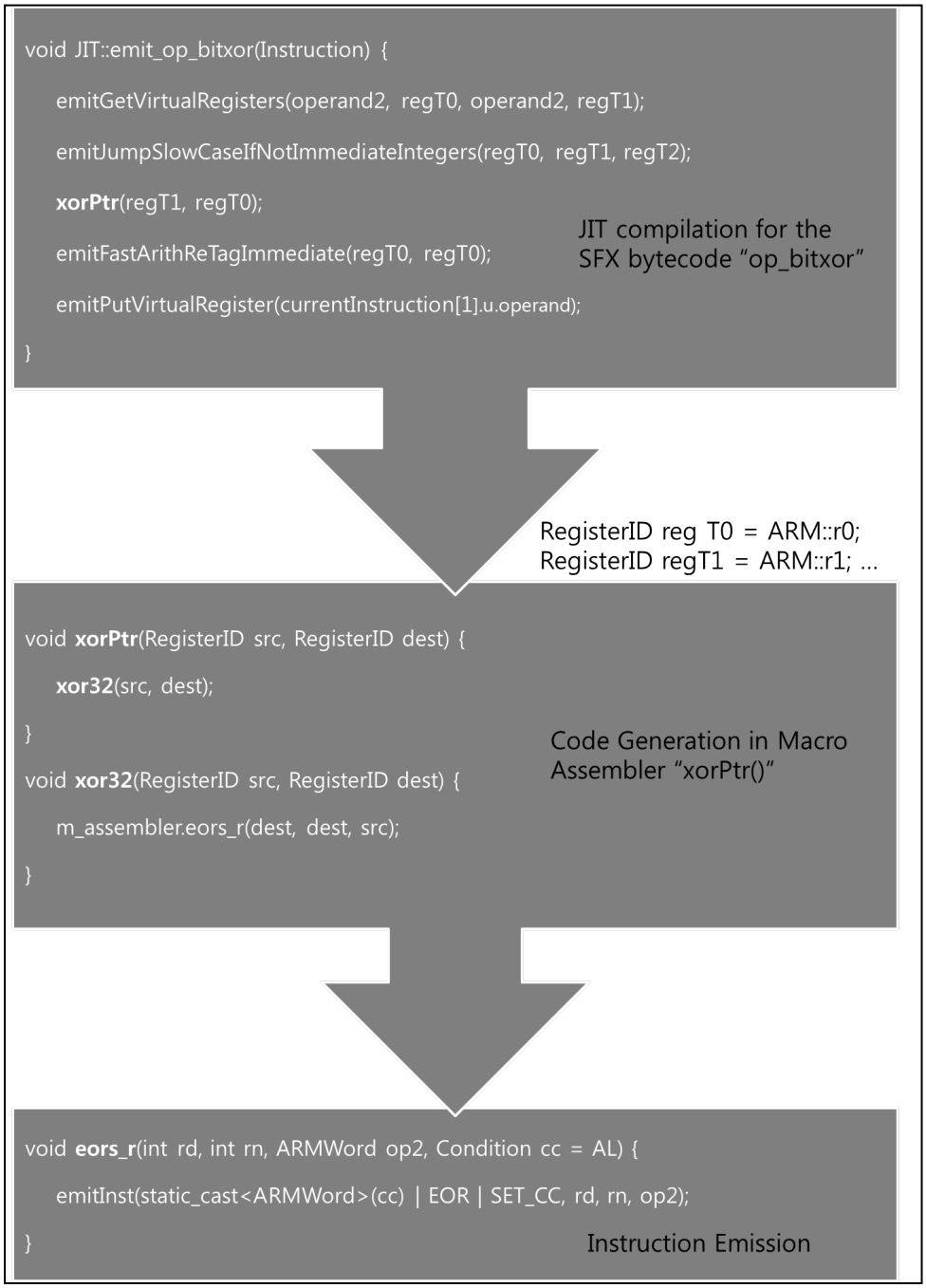

Figure 4-2. SFX macro assembler hierarchy for op_bitxor.

4.1.3 Benchmark JavaScript and Web Page JavaScript

While it is not easy for the SFX JITC to generate quality machine code, the web page JavaScript shows a behavior different from the benchmark JavaScript, making it even difficult for the JITC to improve performance. Generally, the benchmark JavaScript typically includes intensive loops or frequently-called functions, which

resembles those integer and floating-point benchmarks of C/C++/Fortran. Figure 4-3 shows sample JavaScript functions included in the SunSpider benchmark (bitops-nsieve-bits.js), where there are many loops which iterate many times.

On the other hand, the web page JavaScript typically interacts with the DOM tree as we saw in Figure 4-1 and reacts to user events. Previous study shows that the web page executes more than 10 times of JavaScript code than the benchmark [62], while 90% of the execution time is spent in 10% of the functions [58]. They indicate that this would be the reason why the performance benefit of JITC experienced in the benchmark could not be duplicated on the web pages. Actually, even the benchmark programs are reported to perform much worse, if many cold functions are added to them intentionally. Our own study on the number of function calls and loop iterations for web page JavaScript also confirm these behaviors (see Section 4.2).

```
function pad(n,width) {
  var s = n.toString();
  while (s.length < width) s = ' ' + s;
  return s;
}

function primes(isPrime, n) {
  var i, count = 0, m = 10000<<n, size = m+31>>5;

  for (i=0; i<size; i++) isPrime[i] = 0xffffffff;

  for (i=2; i<m; i++)
    if (isPrime[i>>5] & 1<<(i&31)) {
      for (var j=i+i; j<m; j+=i)
        isPrime[j>>5] &= ~(1<<(j&31));
      count++;
    }
}

function sieve() {
    for (var i = 4; i <= 4; i++) {
        var isPrime = new Array((10000<<i)+31>>5);
        primes(isPrime, i);
    }
}

function runBitopsNsieveBits() {
  var _sunSpiderStartDate = new Date();
  sieve();
  var _sunSpiderInterval = new Date() -
_sunSpiderStartDate;
  return _sunSpiderInterval;
}
```

Figure 4-3. Sample functions in the SunSpider benchmark.

These studies indicate that the compilation of all executed functions at their first invocations in the SFX JITC could be problematic. So, we want to introduce selective compilation to the SFX JITC such that only hot functions detected during interpretation are compiled. The problem is how to detect hot functions in JavaScript environment and if those hot functions would really behave as in other environments to justify JITC.

4.2 Selective JITC for the SFX

Previous section overviewed JavaScript embedded in the web pages and described the SFX JITC. It also raised the code quality issues of the SFX JITC and the web page JavaScript behavior, which motivate selective compilation. In this section, we propose the selective JITC for SFX with its heuristics and implementation.

4.2.1 Selective JITC

Adaptive or selective compilation has been popularly employed in Java VM JITC (e.g., Sun's HotSpot VM), 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 function 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 function has been running long enough, we need to estimate the running time of the function.

One simple estimation technique is based on software *counters* such that they count some interpreted bytecodes at runtime. The estimated running time of a function 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, 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] while others count loop iterations as well [64], and they estimate the runtime after multiplying some constants.

We took a similar approach to hot spot detection for the SFX JavaScript JITC, which is described in detail below.

4.2.2 Selective JITC Implementation for the SFX

We first made the interpreter used in the previous version (SquirrelFish) interoperable with the JITC of the SFX such that a function is executed by the interpreter initially and then is compiled by the JITC. Consequently, interpreted functions and JITCed functions can be executed interoperably, so we made the interface efficient so as to minimize the overhead when an interpreted function calls a JITCed function or vice versa.

Our hot function detection heuristic is based on three counters, following that of the HotSpot JVM: the function invocation count, the loop iteration count, and the transition count to JITCed functions. The transition count is measured because if there are many calls to the JITCed functions, the function would better be compiled earlier than other functions if other conditions are equivalent due to two reasons. One is the transition overhead from an interpreted function to a JITCed function, which can be reduced if the caller function is also compiled. The other is that the memory location of the interpreted function and that of the code cache which saves the JITCed function is often far away, which would affect the I-cache performance negatively.

Our heuristic is as follows:

(a) C1 * Invocations + C2 * Iterations + C3 * Transitions > Threshold

In the HotSpot JVM, the default value of C1, C2, C3, and the threshold is 20, 4, 30, and 20,000, respectively. In our case, we simply set C1, C2, and C3 the same value

of 1 for the following reasons. First, C2 needs to be worth more because loops do not iterate much in web page JavaScript (see Section 4.2). Also, the SFX JITC cannot support on-stack-replacement (OSR) unlike the HotSpot VM, which allows compiling a function in the middle of its interpretation and executing the translated machine code continuously in the same runtime context. Thus, under the same condition, the HotSpot JVM can compile a method as soon as its counter exceeds the threshold, yet SFX must wait until its next invocation no matter how many loop iterations are left in current invocation. So, we decided to give more weight to C2 so that a function with heavily-iterating loops is compiled earlier. Secondly, we reduced the value of C3 because the HotSpot VM includes a more complex process for transition between the interpreter and the machine code than our interoperable SFX due to the context migration for OSR. Meanwhile, the threshold value is inversely proportional to number of functions compiled and it depends on applications. We tuned the threshold value for JavaScript-heavy web pages, which is decided to be 23 in our implementation.

4.3 Experimental Result

Previous section proposed selective compilation for the SFX JITC, which attempts to compile only hot functions. In this section, we evaluate the proposed technique with some detailed study of the web page JavaScript on the mobile platform.

4.3.1 Experiment Environment

We experimented with a real smart phone. It has the ARM Cortex-A8 1 GHz CPU with 256MB memory. The browser is based on WebKit r47027, equipped with the SFX. We experimented with the Wi-Fi network.

4.3.2 Web Page JavaScript and SunSpider Benchmark

Our benchmark is composed of 9 JavaScript-heavy web sites. We also experimented with the SunSpider JavaScript benchmark for comparison. For these web sites we first measured JavaScript execution time during the loading of their first page and the whole loading time, whose ratio is shown in Figure 4-4 (this measurement was made with the original JITC of the SFX, and we took a geometric mean for 10 sets of execution).

For these web sites, JavaScript execution time takes more than 20% of the loading time, up to 50% (in Figure 4-4 the rest of the loading time other than JavaScript execution time had some fluctuations caused by the status of the Wi-Fi network

and the web server response time).

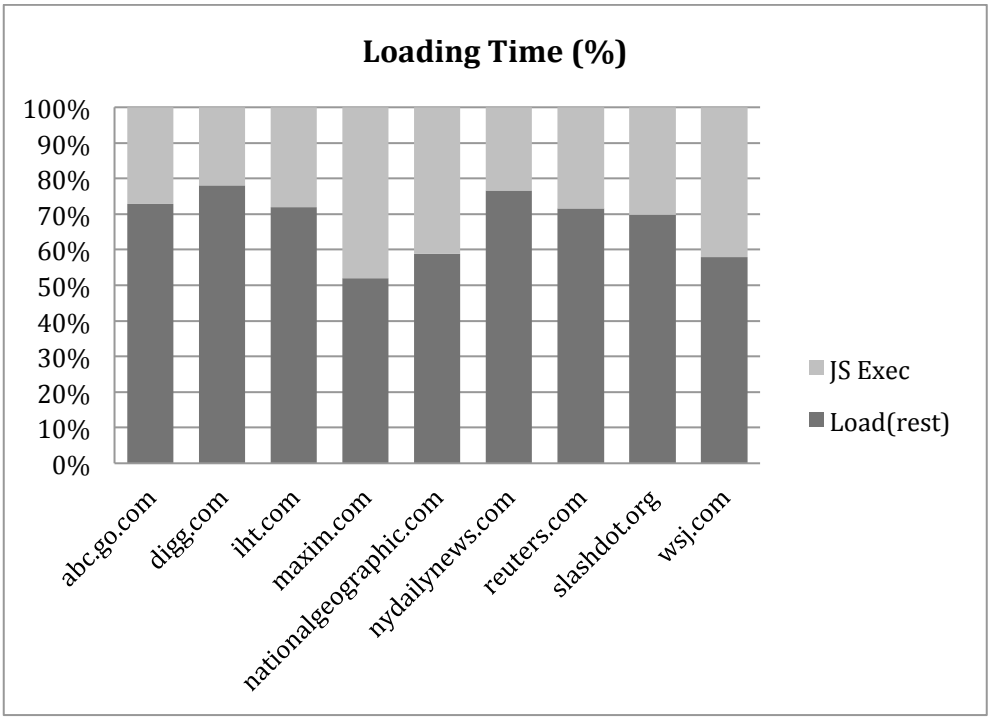

Figure 4-4. Ratio of JavaScript execution time to loading time.

Table 4-1 shows the number of JavaScript functions executed for the loading of the first page. Some web site requires the execution of more than 1000 JavaScript functions for the loading of its first page. It is not easy to understand what those functions actually do since their JavaScript source code is often obfuscated.

Table 4-1. Number of executed JavaScript functions.

Website	Number of Executed Function
abc.go.com	469
digg.com	215
iht.com	835
maxim.com	1429
nationalgeographic.com	872
nydailynews.com	517
reuters.com	622
slashdot.org	359
wsj.com	1012
SUM	6330

For each executed JavaScript function, we measured the number of times it is called, and Figure 4-5 shows its distribution. In sum, around 50% of functions are called only once and around 80% of functions are called less than 10 times. For comparison, Figure 4-5 also depicts the same distribution for the SunSpider benchmark, which shows a quite different profile where there are many frequently called functions.

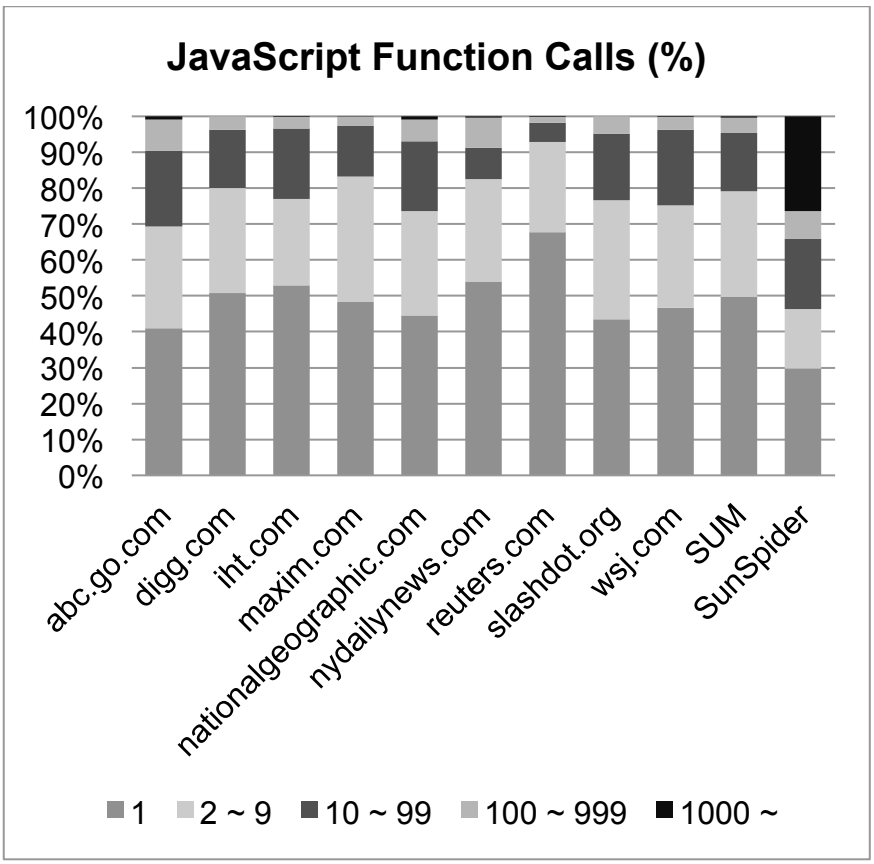

Figure 4-5. Distribution of the number of function calls.

Table 4-2 shows the number of loop iterations, the number of loops, and their ratio for all executed functions. In sum, a JavaScript loop iterates an average of 29.7 times. Table 4-2 also shows the ratio for the SunSpider benchmark, whose loop iterates more than 37,000 times. This means that JavaScript loops are not iterating much, as JavaScript functions are not called much.

Table 4-2. Number of iterations, loops, and the ratio.

	iterations	loops	ratio
abc.go.com	11517	129	89.3
digg.com	1408	77	18.3
iht.com	6456	121	53.4
maxim.com	9441	324	29.1
nationalgeographic.com	22885	957	23.9
nydailynews.com	15524	194	80.0
reuters.com	3334	213	15.7
slashdot.org	1711	65	26.3
wsj.com	12897	786	16.4
SUM	85173	2866	29.7
sunspider	10526132	282	37326.7

Table 4-3 shows the ratio of dynamic bytecode size (which is the sum of the bytecode sizes for all executed bytecode) to the static bytecode size for all executed functions. This will provide a more detailed reuse ratio of JavaScript code than the iteration count or the call count. The ratio for the web site JavaScript is 9.3, which is much smaller than that of the SunSpider benchmark, 3760.1.

These results indicate that in the web page JavaScript, repetitive execution of the same code is somewhat rare, meaning that compiling a function at its first invocation as in the SFX JITC is not likely to be beneficial since the compiled machine code would not be executed repetitively.

Table 4-3. Dynamic bytecode size to static bytecode size.

web site	Ratio
abc.go.com	19.6
digg.com	7.1
iht.com	4.9
maxim.com	6.8
nationalgeographic.com	15.4
nydailynews.com	18.1
reuters.com	3.5
slashdot.org	9.5
wsj.com	6.4
SUM	9.3
SunSpider	3760.1

4.3.3 Web page JavaScript Execution Time

We measured JavaScript execution time of each web page, when executed by the interpreter (Intrpr), the original JITC that compiles all functions at their first invocation (All-JIT), and the selective JITC that compiles only hot functions (Sel-JIT). We made 10 sets of measurements at 10 different times and took their geometric mean. Figure 4-6 shows three bars in each web site, which are the ratios of the execution time of Intrpr, All-JIT, and Sel-JIT, respectively, with Intrpr as a basis of 100%. We also include the ratio for the sum of each execution time at the end.

JavaScript execution time varies among the 10 sets depending on the time we access the web, mainly because the commercials on the web page are changed each time, which makes its JavaScript code execute slightly differently. Such differences of JavaScript execution time are depicted with the ranges at the top in Figure 4-6. However, the performance ordering of the three execution methods and their relative differences are mostly consistent among the 10 sets, and are also kept with the sum result.

The graph shows that the execution time of All-JIT consistently is longer than that of Intrpr, by 10% in sum. This is due to compiling all functions, while they are not

reused effectively. Also, the execution time of Sel-JIT is consistently shorter than that of All-JIT due to the compilation of only hot functions, yet still longer than Intrpr, by 4% in sum.

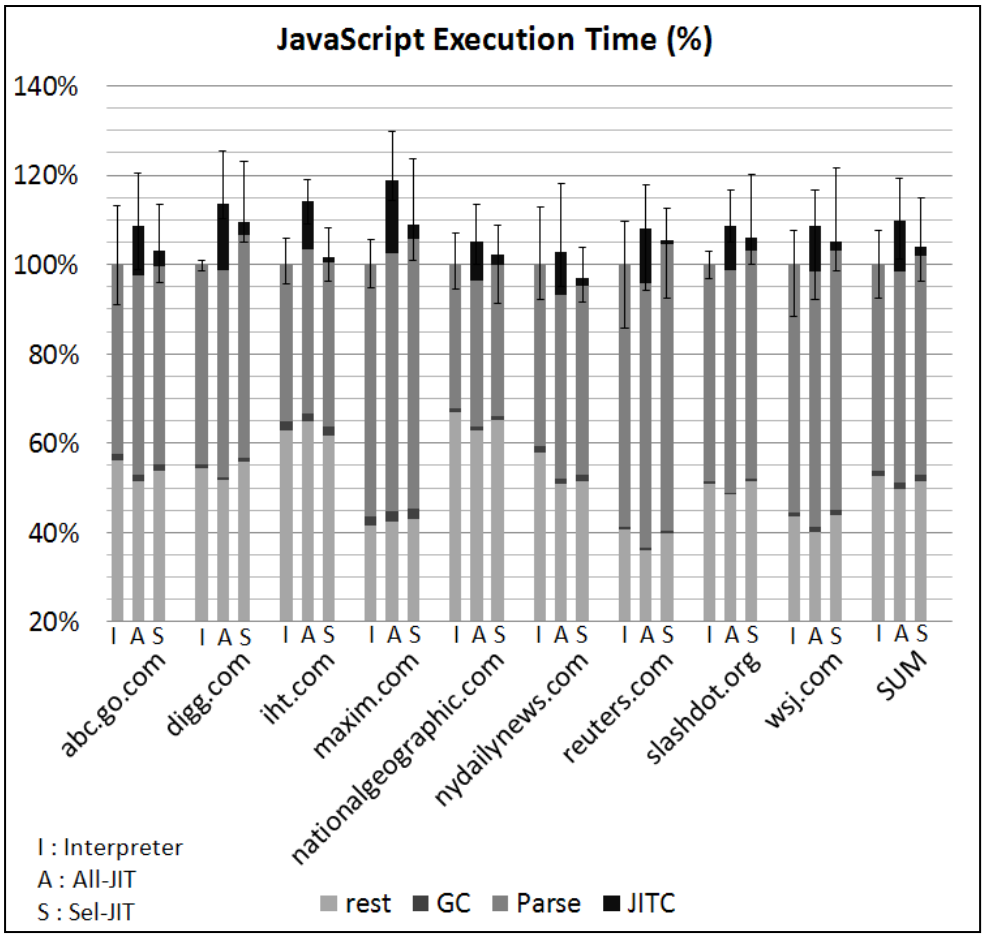

Figure 4-6. Web page JavaScript execution time.

Each bar includes the portion of GC time, parsing time, JITC time, and the rest. We can see that the increase of the JITC time is the main reason for the increased running time of All-JIT, while its decrease is also the reason for the reduced running time of Sel-JIT. In the SUM bar, the JITC time overhead of 11.3% in All-JIT is significantly reduced to 2.2% in Sel-JIT. So Sel-JIT could reduce the running time by 5.7%, even when the rest increases by 1.8% due to the overhead of interpretation/instrumentation. The parse time increases by 1.7% due to some differences in manipulating data structures related to object accesses which are

performed during the parsing phase.⑦

4.3.4 Comparison to Benchmark Execution Time

We now compare the web page JavaScript execution time with that of the SunSpider benchmark, which is depicted in Figure 4-7. The graph shows the three bars in Figure 4-6 for the SUM, compared to those of the SunSpider benchmarks. As we can see, the benchmark result is sharply different from the web page result such that the All-JIT execution time and Sel-JIT execution time is around 40% and 60% of the Intrpr execution time, respectively, showing a big improvement with JITC. Of course, this is due to the repetitive execution as we observed in Figure 4-5, Table 4-2, and Table 4-3. Sel-JIT is worse than All-JIT due to its additional interpretation and the instrumentation overhead.

One thing to note in Figure 4-7 for Intrpr is that parsing portion for the web page JavaScript is higher than for the SunSpider JavaScript. This is so because the relative amount of JavaScript code executed in web pages is much larger than in benchmarks, while the reuse ratio of JavaScript code is much smaller. This is also the reason why the JITC portion in web pages is larger than in benchmarks. Even for web pages, maxim.com or reuters.com where many functions are called infrequently have a larger portion of parsing time and JITC time than other web sites.

⑦ SFX has hash tables to store properties of objects. On each property access, its location in the hash table is cached to avoid repetitive accesses to the hash table. The locations are cached in the bytecode for Intrpr but in a list structure for All-JIT, so the parsing time for the space allocation is different.

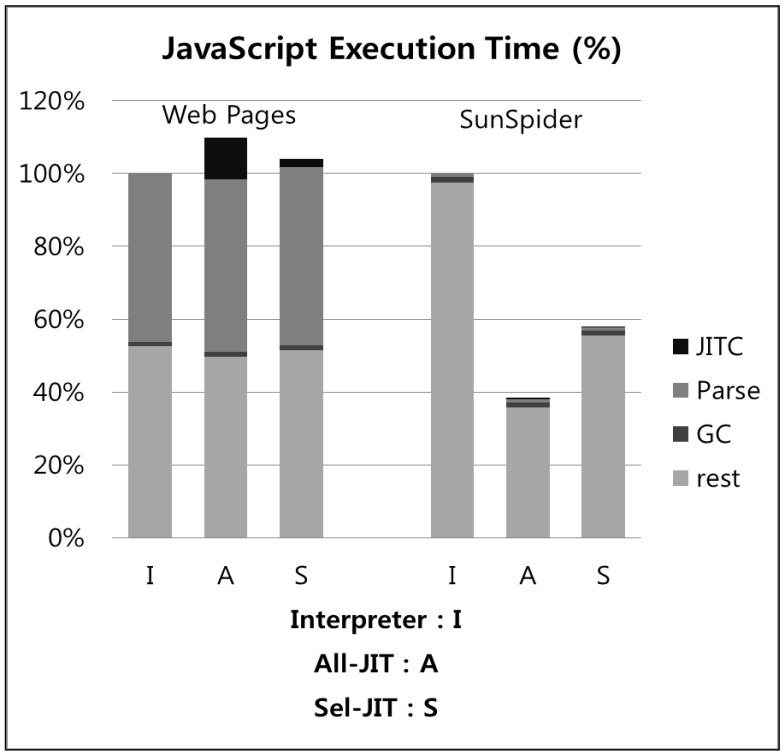

Figure 4-7 Comparing web page and benchmark execution time.

4.3.5 Evaluation of the Selective JITC Heuristic

We now evaluate how the proposed selective JITC works with its hot spot detection heuristic. Table 4-4 shows the ratio of the number of functions compiled to the number of all executed functions (which is 1 for All-JIT). In sum, the ratio is around 18%. For the HotSpot JVM, the ratio is less than 10% [59], yet previous study on JavaScript behavior indicates that around 23% of functions take 90% of the running time [58], so our compilation ratio seems to be reasonable according to the study.

We actually experimented with diverse values of the threshold for our heuristic. Increasing the threshold, which reduces the number of functions compiled, hence the compilation overhead, improves the performance, but we could not see any more improvement after the threshold reaches the current value of 23.

Table 4-4. Ratio of JITCed functions to all executed functions.

	JITCed	Total	ratio
abc.go.com	132	469	0.281
digg.com	39	215	0.181
iht.com	138	835	0.165
maxim.com	213	1429	0.149
nationalgeographic.com	211	872	0.242
nydailynews.com	98	517	0.190
reuters.com	44	622	0.071
slashdot.org	76	359	0.212
wsj.com	213	1012	0.210
SUM	1164	6330	0.184

Table 4-5 shows the average number of iterations per loop that we obtained in Table 4-2, yet for the compiled functions only this time. Compared to Table 4-2, the ratio is much higher (e.g., 128.7 iterations vs. 29.7 iterations per loop in sum), meaning that we compiled those functions which have loops with heavier iterations. This is also confirmed by comparing the ratio for the SunSpider benchmark (40534.4 iterations vs. 37326.7 iterations per loop).

Table 4-5. The data of Table 4-2 for JITCed functions only.

	iterations	loops	ratio
abc.go.com	7045	70	100.6
digg.com	401	20	20.1
iht.com	2251	20	112.6
maxim.com	6800	62	109.7
nationalgeographic.com	17645	84	210.1
nydailynews.com	13125	56	234.4
reuters.com	382	20	19.1
slashdot.org	1409	22	64.0
wsj.com	8842	96	92.1
SUM	57900	450	128.7
SunSpider	5958550	147	40534.4

4.3.6 Discussions

All-JIT suffers from the compilation overhead which increases the running time of Intrpr by around 10%. Unfortunately, All-JIT does not seem to achieve any significant benefit from machine code execution that can offset the compilation overhead. We can suspect two problems for the machine code execution: low reusability of the machine code or low code quality.

In Figure 4-7, the "rest" portion of the running time in All-JIT, which roughly corresponds to the execution time of the machine code, does not decrease much compared to the "rest" portion in Intrpr, which roughly corresponds to the interpretation time. This is in sharp contrast to the SunSpider benchmark, which shows a big (i.e., 60%) drop of the rest portion when moving from Intrpr to All-JIT. This means that the code quality is less of an issue than the reusability of machine code, unless the JITC generates lower-quality code for the web page JavaScript than the benchmark JavaScript. We need to investigate more to verify this.

What we can expect from Sel-JIT is the decrease of the JITC overhead. In fact, Sel-JIT reduces the JITC overhead from 10% to 2%, yet this still does not lead to a better performance than Intrpr. This is somewhat obvious since there is not much gain from the execution of the machine code, as we noted in comparing the "rest"

for All-JIT and Intrpr in the above. Sel-JIT can increase the “rest” portion of All-JIT by the interpretation overhead and the instrumentation overhead, yet the overhead appears to be less than 1.8% in web pages, if we compare the “rest” portion of All-JIT and Sel-JIT in Figure 4-7.

On the other hand, the JITC overhead of the benchmark in All-JIT is less than 1%, so Sel-JIT cannot improve the JITC overhead. Moreover, there are heavily-iterating outer loops or recursions in the SunSpider benchmark, so Sel-JIT which must interpret them initially for detecting hot methods with instrumentation overhead cannot outperform All-JIT which executes only in machine code. If we lower the threshold so as to compile more functions earlier, we would improve the performance of the SunSpider, yet then the performance of the web pages would be affected.

In order to increase the performance of Sel-JIT, we first need to investigate the quality of JITC-generated code for web page JavaScript and improve it. As we have mentioned in Section 2.2, the SFX JITC cannot generate quality code and requires more optimizations. We could hide the optimization overhead as dual-core CPUs are being introduced in the smart phones if we make the JITC module as a separate thread. If the SFX JITC can generate better code with its compilation hidden, we expect Sel-JIT (or All-JIT) could outperform Intrpr.

Another improvement idea is having an adaptive threshold value which can vary depending on the behavior of JavaScript programs. For example, if there are heavy iterations, the JITC reduces the threshold for earlier compilation, and if not, the JITC increases the threshold for more selective compilation. It is also needed to reduce the instrumentation for hot spot detection.

Chapter 5. Bytecode Level Optimizations

5.1 Analysis on Web Page JavaScript Execution

Web pages are programmed using HTML5, CSS, and JavaScript. During web page loading, the browser parses the HTML document and builds a document object model (DOM) tree, which is then displayed on the screen based on CSS by the rendering engine. When the JavaScript tag is met during the HTML parsing, the corresponding JavaScript code is executed by the JavaScript engine, mostly for initializing objects and registering event handlers. JavaScript code is first parsed to the bytecode, which is interpreted first. If one of the functions is repeatedly executed and becomes hot in runtime, the just-in-time compiler (JITC) translate it to machine code for faster execution. The JavaScript execution takes a significant (72%) portion in some web page loading time.

We found that JITC is not effective for web pages, especially for accelerating web page loading. For example, Figure 5-1 shows the performance of the JITC when we disabled the interpreter for the JavaScript execution time during the loading of some web pages, compared to that of the interpreter when we disabled the JITC. We experiment with a Safari browser based on the WebKit and the JavaScriptCore (JSC) engine [70]; they are (1) digg.com (digg), (2) maxim.com (maxim), (3) nationalgeographic.com (ng), (4) Slashdot.org (slashdot), (5) wsj.com (wsj), (6) foxnews.com (foxnews), (7) msnbc.com (msnbc), (8) nba.com (nba), (9) nhl.com (nhl), and (10) Sunspider benchmark (sunspider) for comparison, respectively. For all web pages, the JITC shows a worse performance than the interpreter. This is in sharp contrast for the Sunspider benchmark where the JITC is much better.

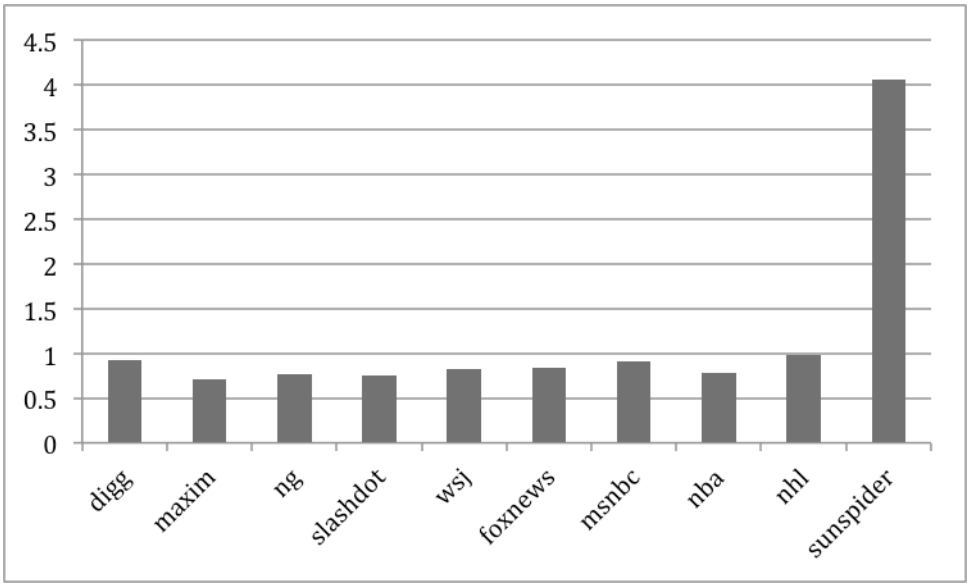

Figure 5-1. JavaScript JITC Speed-up over Interpretation

Since JavaScript execution tends to be rarely repeated during web pages loading, compiling every executed functions would be very inefficient[67]. Therefore, recent JavaScript engines applies adaptive JIT compilation to accelerate only hot spots while reducing JIT compilation overhead for the other functions in the working set.

However, one problem is that JavaScript execution during web page loading spends much of its time for executing *runtime services* of the JavaScript engine, which cannot be accelerated by the JITC. Runtime services are for executing some complex jobs such as object property accesses which read or write property values stored in objects, object creations which allocate memory spaces and construct data structures, or floating point operations which invoke floating-point system libraries, requested by the interpreter or the JITC. They are part of the functions in the JavaScript engine usually written in a low-level programming language such as C++. Figure 5-2 shows the distribution of JavaScript running time during the web page loading. Runtime services take 20% to 75% of the running time but 98% of runtime services are called by interpreter. Parsing and JITC overhead is also significant, leaving the somewhat marginal runtime portion of the interpretation (Inrpr), JITC-generated machine code (JITCed), and the other native code (Native) called from interpreter or JITC-generated machine code such as DOM APIs.

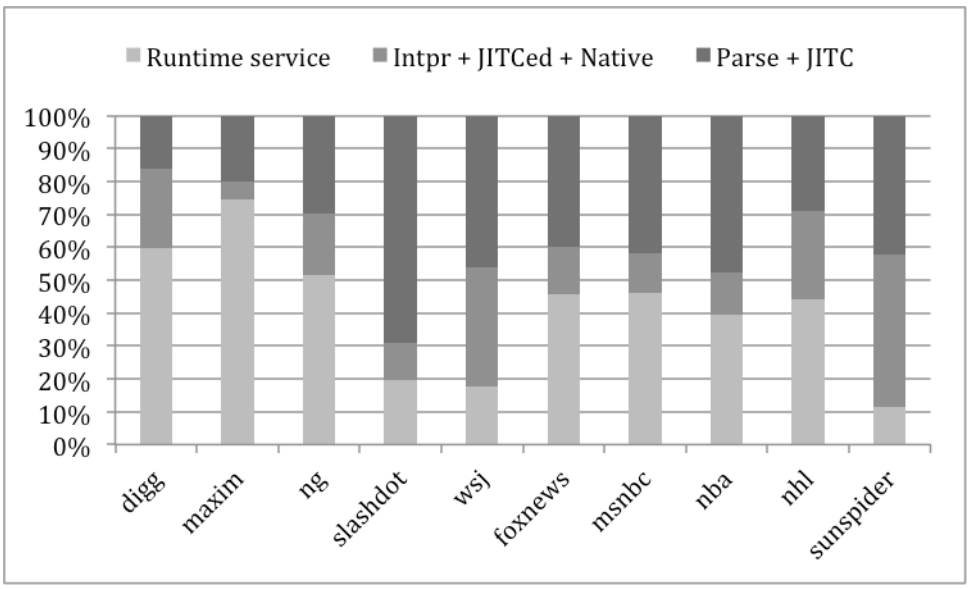

Figure 5-2. Distribution of JavaScript execution during loading.

The runtime portion of the Intpr, the JITCed, and the Native in the above graph indicates the JITC has reached the limit to speed up JavaScript further for web page loading because the JITC would compile only hot functions adaptively and the other functions are beneficial to be interpreted considering their compilation overhead. On the other hand, there is not much to accelerate the interpreter routine itself. And, the native functions are fixed, so there is nothing to accelerate them, either. The only thing left to optimize is the runtime services of the JavaScript engine. We found that the runtime services for object property accesses are dominant among the execution time of the runtime services as shown in Figure 5-3.

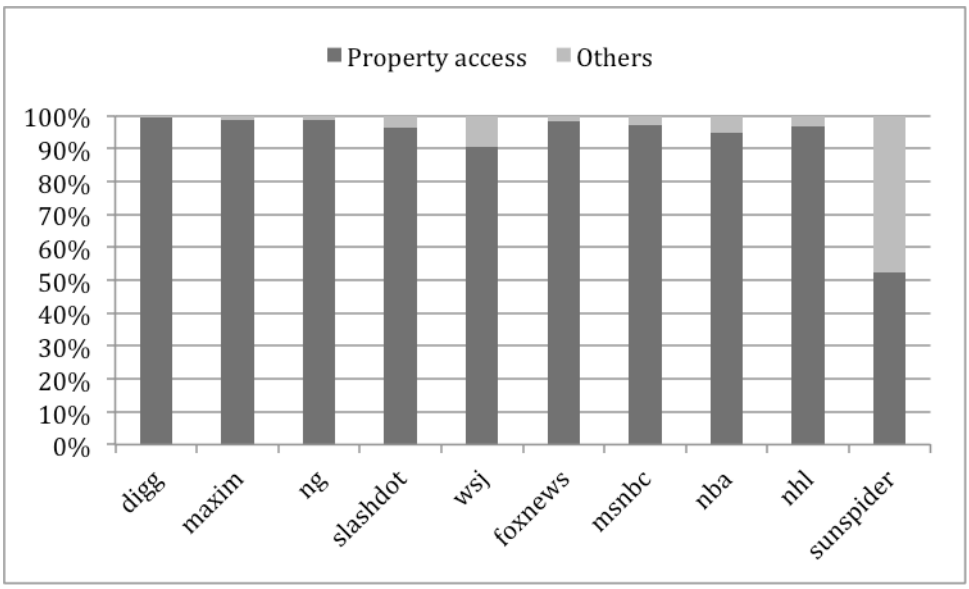

Figure 5-3. Portion of property access in runtime services.

We also found that 84% of the accesses are the one-time accesses, and repeated accesses are just 16%. By the way, the one-time accesses only have to be executed by the runtime services while the repeated accesses can be handled without the runtime services due to the optimization called inline caching. As a result, the runtime services consume majority of property access time in web pages loading unlike the benchmark as shown in Figure 5-4.

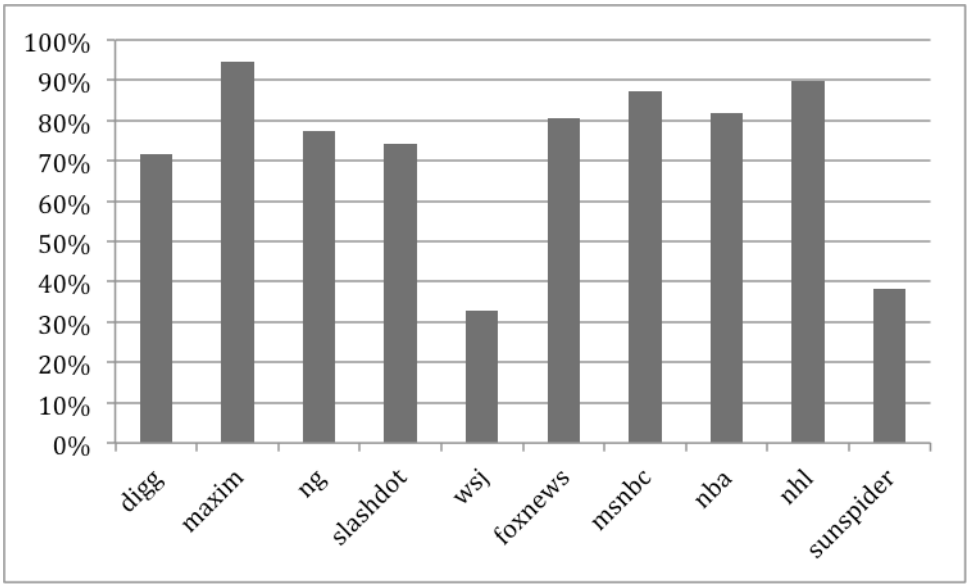

Figure 5-4. Portion of runtime services in property access time.

Based on these observation, we attempt to optimize the object property accesses during the web page loading, in the context of interpreter, especially for the first-time accesses, most of which are one-time accesses. We address bytecode level optimizations which can reduce the runtime service portion called by the interpreter. First, we suggest the super bytecode construction which supplies more optimized runtime services. Secondly, we propose the bytecode chaining which removes redundant runtime service requests.

5.2 Overhead in Property Accesses

We first explain the overhead involved with the object property access, especially the first-time access. A JavaScript object has a *storage table* which has the values of its properties and a *property map* which describes the offset of each property in the storage table [68]. For example, consider a function *point()* in Figure 5-5 (a), which is used as a constructor of an object to initialize its properties. Figure 5-5 (b) shows the bytecode generated by the parser of the JSC [70], where *put_by_id* will initialize each property. The runtime service for *put_by_id* will first check if the property exists in the object; if not, it will add the property in the storage table and create a new property map based on the old one, added with the offset of the new property. So, each *put_by_id* will generate a new property map as in Figure 5-5 (c). If *point()* is called again to create a new object, the previous property maps will be reused without creating them again, though [68]. However, web-page loading is involved with many one-time object property accesses, so the creation of property maps for each bytecode can be an overhead.

Another overhead of object accesses is related to *inline caching* [69]. If *point()* is called again repetitively as in Figure 5-5 (a) and the object structure (i.e., the constructor *point()*) does not change over iterations, it would be better to remember the offset and use it directly instead of accessing the property map. So, the bytecode *put_by_id* is replaced by a quicker version where the offset and the address of the property map are saved. When it is interpreted, the address is compared to the current address of the property map first, and if they are the same, the offset in the bytecode is used directly to access the property in the storage table without using the runtime services. This is called *inline caching*. The issue is that we need to compare the address three times for each access in Figure 5-5 (a), for

example, which we want to reduce.

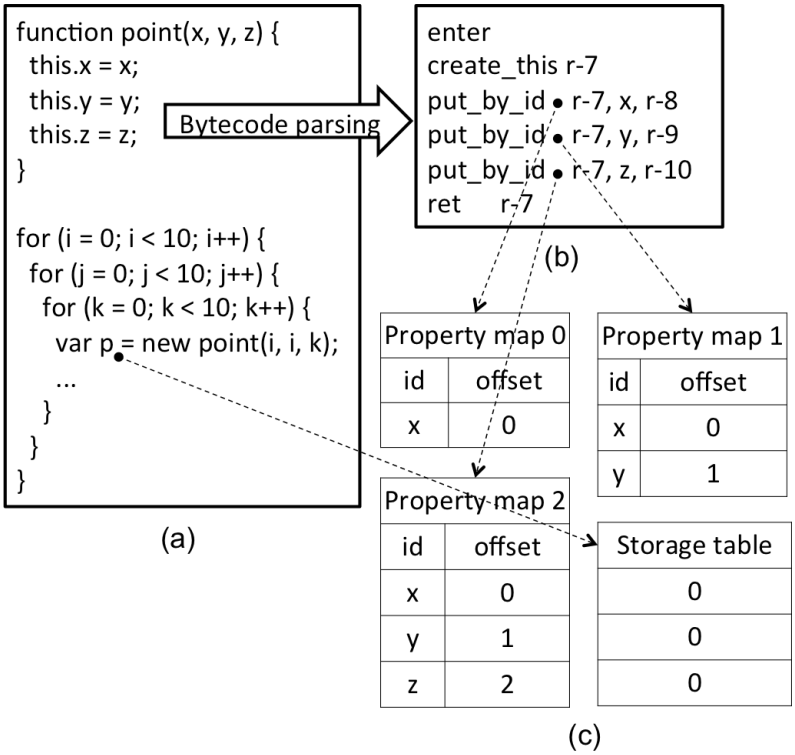

Figure 5-5. An example of the object property accesses.

Our optimization approach to reduce this overhead is using a *super-bytecode* instruction which merges multiple object-access bytecodes into one bytecode. This can reduce the overhead of the runtime services of individual bytecode, especially for the first-time object accesses. For inline caching, super-bytecode can also reduce the overhead of comparing multiple addresses of property maps by making a single comparison with the final property map. We will describe how to generate the super-bytecode in the next section.

Another issue is the redundant runtime service requests. The inline cached information such as the address of the property map or the offset in the storage table is obtained as a result of the runtime service request. But it can be used only when the cached bytecode instruction is executed again. If we utilize the information for another bytecode instructions, we could replace the corresponding runtime service requests to the fast inline cached accesses on their first executions.

For example, the Figure 5-6 illustrates a case which we can replace the runtime service request to the inline cached access. Object “p” and “q” are not the same

object but they share the same property map because they are created using the same constructor function “point”. After the creations of “p” and “q”, property “x”, “y”, and “z” are accessed for “p” followed by “q” respectively. Since “p” and “q” have the same property map, property accesses for the same identifier will be inline-cached with the same offset. For example, the bytecode *get_by_id r2, r-15, x* will be inline-cached with the property map “p0” and the offset “0” such like the bytecode *get_by_id r2, r-14, x, p0, 0* eventually. Therefore, we do not need to invoke the runtime service which will performs the same operation because we know that the property map p0 and the offsets for x, y, and z are already inline-cached after p.x, p.y, and p.z are accessed with the runtime service.

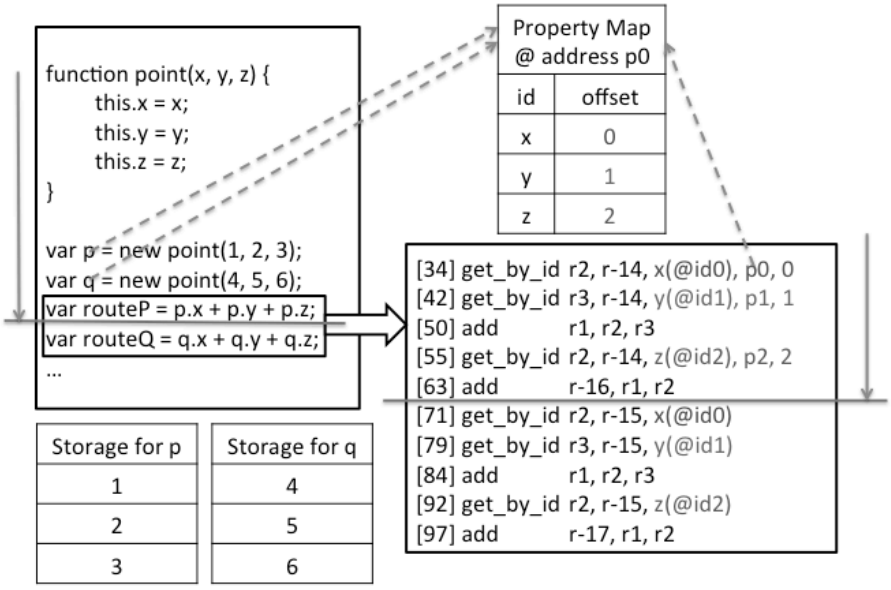

Figure 5-6. Redundant runtime service requests

We propose the bytecode chaining to avoid this redundant runtime service requests and replace them to fast direct accesses using the inline-cached information which is recorded in other bytecode instruction. The bytecode chaining links adjacent bytecode instructions which access properties of the same identifier recording the locations of the bytecode instructions as described in section 6.4. We can access properties directly using the inline-cached offsets in other bytecode instructions

which included in the bytecode chain and remove the corresponding redundant runtime service invocations contributing to the performance improvement.

5.3 Super-Bytecode Construction (SBC)

When a function is called for the first time, we perform the super-bytecode construction (SBC) for the method. To find the candidate object accesses for SBC, we first perform an analysis on the basic block (BB) boundary to reduce the analysis overhead. In the current implementation, we simply merge a sequence of *put_by_id*s for the same object in the BB into a single bytecode *m_put_by_id.* Similarly, a sequence of *get_by_id*s for the same object is replaced by a single bytecode *m_get_by_id.* In Figure 5-5 (b), we can replace the three *put_by_id*s which write the three properties for the same object by a new bytecode *m_put_by_id r-7, {x,y,z}, {r-8,r-9,r-10}* as described in Figure 5-7. This will write the three properties by a merged, optimized runtime service routine, instead of three runtime service routines.

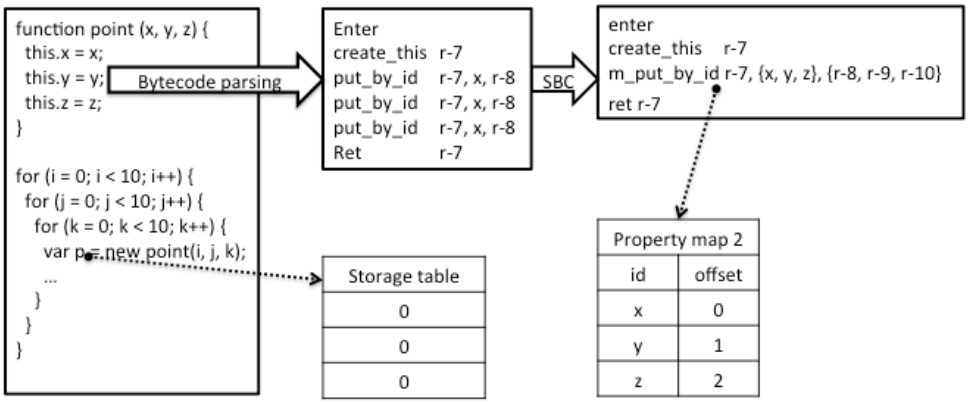

Figure 5-7. An example of the super-bytecode construction.

The benefit of the super-bytecode is two-folds. We can reduce the overhead of creating the property map for each individual bytecode for the first-time object property accesses. In Figure 5-5 (c), it is an overhead to create the property map0 and map1 since only the property map2 is enough to resolve the offset of x, y, and z. Even when the function *point()* is called repetitively, the property map2 is enough to resolve the offsets. Our super-bytecode will obviate the overhead of creating the extra property maps or accessing them, which will be more useful as more property accesses are merged, even if they are executed only once.

We can also have a benefit when the super-bytecode is executed repetitively. Inline caching requires the address comparison of the property map, which we need to do for each access. In Figure 5-5 (c), for example, we need to compare three times for the three accesses with the corresponding property maps. SBC requires saving only one address and comparing only once, which would accelerate the inline-cached, property accesses.

5.4 Bytecode Chaining (BC)

Bytecode chaining is performed along with the source code parsing when a JavaScript function is called for the first time. During the parsing, whenever a property access bytecode instruction is generated, the property identifier, which is specified as an operand in the instruction, is recorded in a table together with the location of the instruction. If another location of a bytecode instruction for the same identifier already exists in the table, the information is updated with the new location. Next we construct a bytecode chain linking the new location backward to the bytecode instruction of the old location or linking the old location forward to the bytecode instruction of the new location.

After the bytecode chaining is completed with the parsing, the interpreter tries to utilize the inline-cached information of the first executed instruction in a chain for the rest instructions. At this time, we can think of two ways to apply the inline-cached information of the first executed instruction to the following instructions in the bytecode chain. One is the speculative caching which caches the information unconditionally to all following instructions traversing the forward link right after the inline caching of the first executed instruction as illustrated in Figure 5-8 for the example in Figure 5-6. The other is the lazy caching which tries to searche the inline cached information of the first executed instruction traversing the backward link whenever each instruction in the chain is executed as illustrated in Figure 5-9 for the example in Figure 5-6.

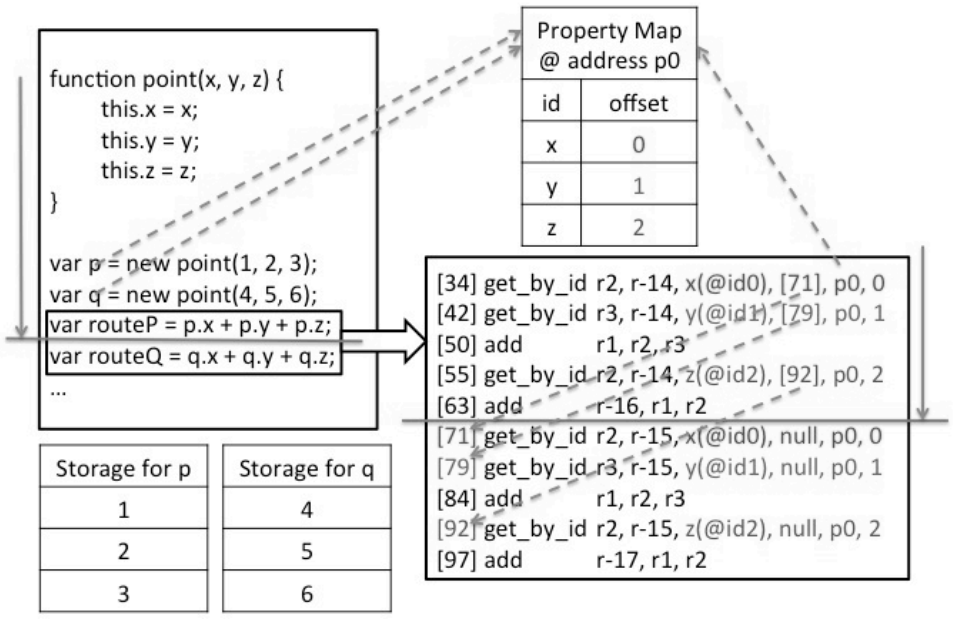

Figure 5-8. An example of the speculative caching.

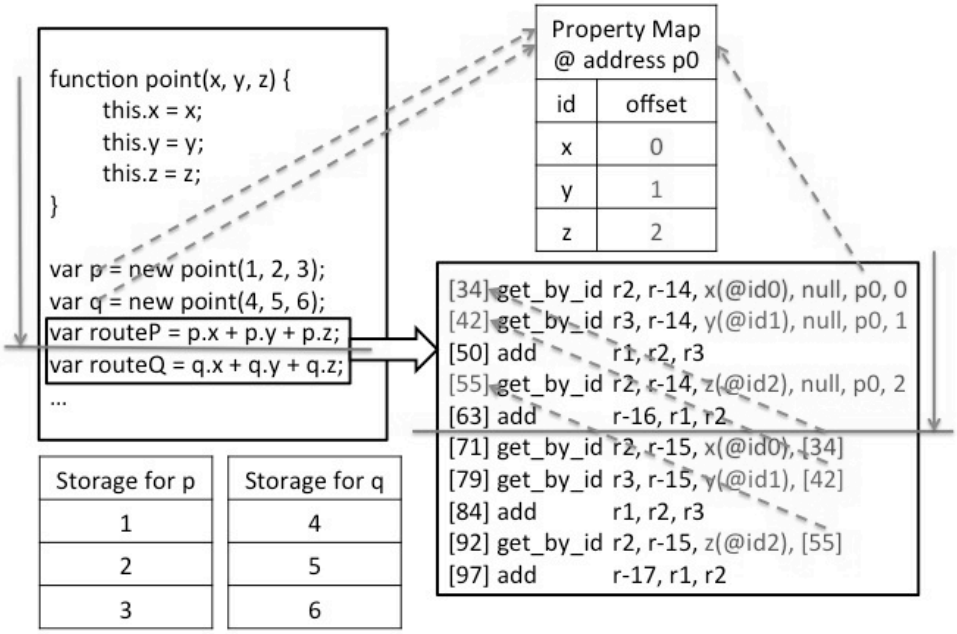

Figure 5-9. An example of the lazy caching.

5.5 Experimental Evaluation

We implemented the proposed SBC and BC for the JSC bytecode interpreter in a WebKit-based Safari browser (the adaptive JIT compilers are enabled). We measured the JavaScript execution time during the loading of web pages on an

x86-based environment in which the 2.3 GHz quad-core CPU runs with 8GB RAM.

5.5.1 Performance Result

Figure 5-10 displays the speed-up over the JavaScript execution time with JITC when the interpreter is disabled. Each data in the graph is obtained as the following.

(a) Intpr: the JavaScript execution only with the interpreter when the JITC is disabled

(b) Adaptive: the adaptive JavaScript execution both with the interpreter and the JITC

(c) SBC: The SBC is applied to the adaptive execution of (b).

(d) SBC+BC(SC): The bytecode chaining with the speculative caching is applied to the execution with SBC of (c).

(e) SBC+BC(LC): The bytecode chaining with the lazy caching is applied to the execution with SBC of (c).

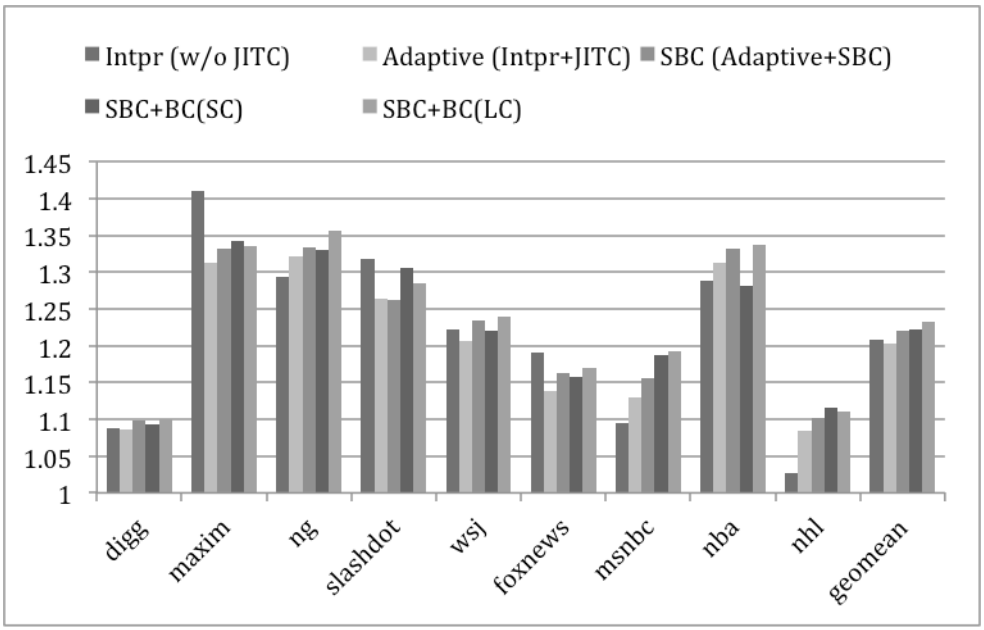

Figure 5-10. Speed-up over JITC only execution.

As we can see in the graph above, for the JavaScript execution during web pages loading, the Intpr outperforms the execution only with the JITC about 20%. On the other hand, the Adaptive shows better performance than the Intpr in some web pages such as msnbc or nhl due to the acceleration from the JITC. However, the JITC overhead of the Adaptive could make the performance degradation in the other web pages. As a result, the average performance of the Adaptive is slightly

worse than the Intpr.

If we applied the SBC to the Adaptive, we can see the improvement in all web pages except for the slashdot. Furthermore, the SBC makes better performance than the Intpr in the digg and the wsj which the Adaptive can not accelerate. After the bytecode chaining is added to the adaptive execution together with the SBC, we can obtain tangible enhancement over the Adaptive in all web pages including the 6.3 improvement in the msnbc. As a result, our proposed bytecode level optimizations makes the 3% JavaScript speed-up in web pages loading on average.

5.5.2 Performance Analysis

We measured the dynamic invocation count from the interpreter to the runtime service and figured out how many of them are optimized by our bytecode level optimizations in order to see the factor of the performance improvement.

5.5.2.1 Optimized Runtime Services with SBC

The Figure 5-11 shows the percentage of the runtime service invocations (Normal) which will be replaced by the optimized runtime service invocations (Optimized) for the super-bytecode execution when the SBC is applied. In this graph, we can see how much the JavaScript execution for the runtime service invocations is accelerated by the SBC. Also, the ratio of the Normal to the Optimized indicates the average number of runtime service invocations which is accelerated by a single optimized runtime service.

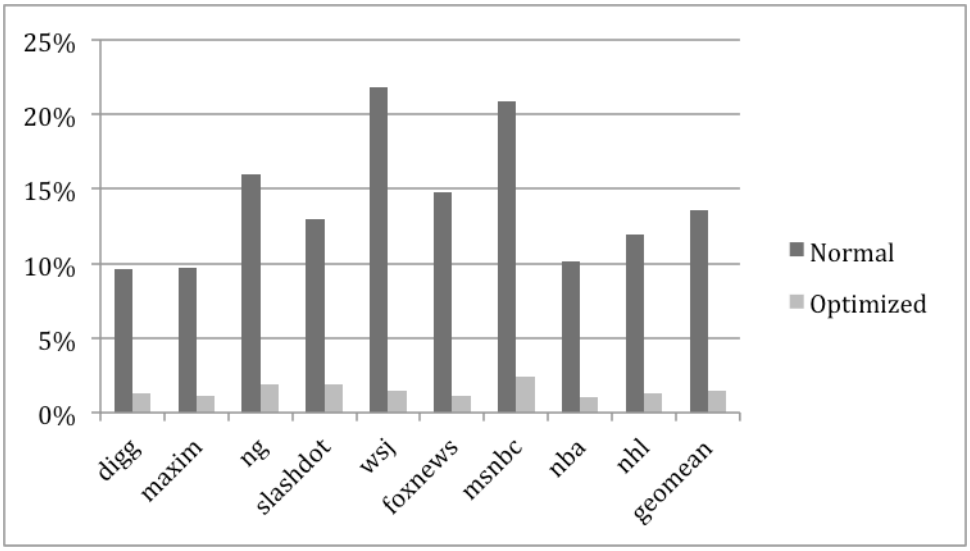

Figure 5-11. Runtime service invocations replaced by optimized runtime services for SBC.

More than 20% of the runtime service invocations is accelerated by the SBC in the wsj and the msnbc while 13% of the runtime service invocations is accelerated on average. As a result, the 2.3% performance improvement in the wsj and the msnbc is higher than the other web pages. Especially the wsj obtains high speed-up despite of its relatively low runtime service portion which is around 20% as displayed in Figure 5-2 in section 6.1. But we can find another clue that the ratio of the Normal to the Optimized for the wsj is 14.5 which is much higher than the average which is just 9.3. So, the wsj is accelerated more than the other web pages because SBC replace larger portion of the runtime service invocations to the optimized runtime service invocations as well as a single optimized runtime service invocation handles more property accesses on the execution of a super-bytecode instruction.

5.5.2.2 Removed Runtime Services with BC

Figure 5-12 shows the percentage of the redundant runtime service invocations which will be removed when the bytecode chaining is applied. Now we can see the portion of the runtime service invocations which is converted to direct accesses and accelerated by the bytecode chaining.

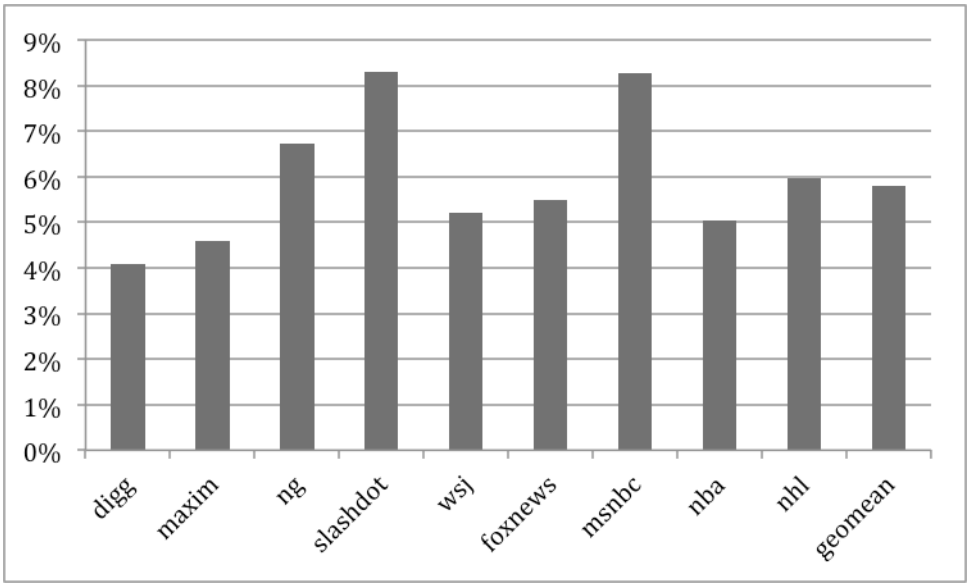

Figure 5-12. Runtime services invocations removed by BC.

More than 8% of the runtime service invocations is removed by the bytecode chaining in the slashdot and the msnbc while 6% of the runtime service invocations is replaced by direct accesses on average contributing to improve the performance.

Especially since the portion of the runtime service in the msnbc is much higher than the slashdot as displayed in Figure 5-2 in section 6.1, the wsj speed-up is 3.4% which is higher than the slashdot speed-up which is just 1.7%.

Chapter 6. Related Work

In this paper, we evaluated hot spot detection techniques which are relatively simple, thus appropriate for embedded systems. This section describes related work on hot spot detection, which would be more appropriate for servers or desktops due to the use of sampling, hardware units not supported in embedded systems, or complex adaptive compilation with a high overhead.

Jikes RVM is equipped with multiple JITCs with different optimization levels for full-fledged adaptive compilation [8]. When a method is called for the first time, it is compiled by the base-line JITC. There is a separate thread for sampling, which examines the call stack on a regular interval and counts the methods on the call stack. If the sampling count of a method exceeds a threshold, it is compiled by the next optimization-level JITC. Independently from the sampling period, the sampling count of each method is decremented by one on a different, regular interval, which keeps *old* hot spots from being compiled; some method which was frequently called previously might have a sampling count close to the threshold, but if it is not called enough recently, the method should not be compiled since it is not a current hot spot.

Hardware performance monitors (HPM) can be exploited to improve hot spot detection using the precise runtime information. Gu et. al define the execution phase of a program using the L1 cache miss rate and propose a JITC heuristic for each phase to replace the existing sampling heuristics of the Jikes RVM [20]. More precisely, they monitor the L1 cache miss rate continuously and judge the right phase of the current program at the process scheduling time to apply the corresponding JITC heuristic. On the other hand, Buytaert et. al attempt to refine the sampling heuristics of the Jikes RVM using the HPM [21]. Sampling in the Jikes RVM occurs only at the *yield* points of the executing program such as the backward branch or the method prolog where the sampling timer event can be handled, but this can make the sampling interval inconsistent. HPM can control the whole system when its cycle counter overflows, so HPM-based sampling can achieve a consistent interval.

Open runtime platform (ORP) uses counter-based hot spot detection but does not include an interpreter as in the Jikes RVM [22]. Instead, ORP's baseline JITC

generates instrumentation code for counting method calls and loop iterations, whose overhead is higher than the interpreter's. To reduce the overhead, two versions of the native code are generated: one with detailed instrumentation code and the other with no instrumentation code [24]. Initially, the fast, non-instrumented code is executed where a single counter is maintained to count the total number of method calls, but if this counter overflows, the instrumented version is executed to get the detailed counts. Now, the counter is reset, which makes the non-instrumented code execute again. This *sampled instrumentation* approach is used for hot spot detection with a reduced instrumentation overhead, which is employed in the IBM J9 JVM [23].

Oracle's HotSpot JITC provides additional heuristics for its server version. When a hot method is detected and compiled, the caller method in the call stack is also compiled even if its estimated time does not reach the threshold [25]. This is to reduce the transition overhead from the interpreted caller to JIT compiled callee earlier (see Section 2.2), for better performance.

There is also an approach to mix the different hot spot detection techniques. For example, interpreted methods use counters, compiled methods by the baseline JITC use sampling, and those methods to be compiled by the highest-level JITC use instrumentation [26]. This can balance between the preciseness and the overhead of hot spot detection in adaptive compilation with multi-level JITCs.

Offline profiling can also help hot spot detection. Bytecode trace obtained from a profile run is analyzed to obtain the loop iteration counts, which are annotated to the bytecode so that faster hot spot detection is achieved using this information [27]. Offline profiling based on instrumented bytecode is also proposed to allow portable profiling without changing the VM internals [32]. Precise offline profiling is shown to be possible for the Pharo language VM, where all arithmetic operations and loop iterations are implemented by library calls, so counting such library calls lead to precise estimation of the running time [33]. Utilizing these offline techniques for online profiling is shown to be inappropriate due to their overhead, though.

Chapter 7. Conclusion

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 their compilation at the first invocation. 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 the performance of the benchmarks tangibly. They also work for embedded applications such that the user response time of the DTV Xlet application can be improved. Although hot spot detection is a difficult and somewhat arbitrary problem, FSRE appears to be a promising approach to detect hot spots. We need to optimize and tune further, and one thing to tune is the constant values such as C3 or C8. Deciding these constant values depending on the method structure or on the actual method execution time would be left as a future work.

JITC has been introduced to mobile JavaScript engine for accelerating full web browsing, mobile RIA, and widgets. The code size improvement is an important issue for mobile JITC since JITC-generated code can be increasingly huge as the web browsing proceeds, causing high memory pressure. We developed a JITC for the Thumb2 architecture in order to exploit its code size advantage, for the Webkit's SFX JavaScript engine. One constraint is that we use their macro assemblers to follow their open-source guidelines, which restricts optimizations for reducing code size and for avoiding performance degradation. We tried best to generate as many 16-bit Thumb2 instructions as possible and to reduce the number and the size of CPs. Our experimental results on the Sunspider benchmarks show that we could reduce the code size by 29%, with a performance penalty of 3.5%. This is much better than when exploiting Thumb2 for Java VM JITC, and is comparable to when generating Thumb2 code with static compiler.

Full web page loading on the smart phones affect users' response seriously due to slow JavaScript execution based on interpretation. In order to accelerate JavaScript execution, JITC has been introduced to JavaScript engine, with an expectation of

performance improvement as in other VMs. Unfortunately, our study indicates that JITC is not highly effective for web page JavaScript due to low reusability or code quality, especially when all executed functions are compiled as in the SFX.

In order to meet the web page JavaScript behavior, we introduce selective compilation to the SFX JITC, which could reduce the compilation overhead by compiling only hot functions for the web page JavaScript, while keeping the advantage of JITC for the benchmark. Unfortunately, we could not achieve a highly satisfactory result for neither of them, and our investigation shows the low reusability of compiled code is the primary reason. It would be a challenging problem to design an efficient JavaScript JITC that can work well for both web pages and benchmarks.

We believe that other JavaScript JITCs suffer from the same problems especially in the smart phone area. In fact, adaptive JITC or hybrid JITC is being introduced in other engines recently. Google's V8 introduced Crankshaft [66], which compiles all executed functions using a baseline compiler at their first invocations to generate unoptimized machine code fast. Then it recompiles some of them later with more optimizations such as loop invariant code motion or inlining, if they are found to be hot spots via profiling. Mozilla's Firefox employed a new JavaScript engine called JaegerMonkey [65]. The existing TraceMonkey interprets first to detect hot traces of loops and then compile them. Since the unit of compilation is a trace, it has been less effective than other function-based JITC engines. JaegerMonkey added the function-based JITC so that it works with the trace-based JITC and the interpreter in order to achieve a better performance.

The competition among JavaScript engines to address the challenge proposed in this paper will be more intense in the future, as we need to accelerate a new form of JavaScript such as mobile RIA or web OS, as well as existing web page JavaScript. And, we think that a more efficient, adaptive compilation would be a solution where hot spots are compiled to highly efficient code even if the compilation overhead is high, while cold spots cause little compilation overhead even if they are executed slowly. We hope our selective compilation explored in this paper would provide a clue.

In summary, the JITC tends to bring the JavaScript speed-down in web pages loading and the adaptive execution with the interpreter can not help to outperform

the interpretation-only execution. Under this circumstance, we applied the SBC and the bytecode chaining to the adaptive execution environment and achieved definite better performance than the interpretation. Therefore, we can conclude that our bytecode level optimizations are much more effective than the JITC for JavaScript speed-up in web pages loading considering even the adaptive execution just makes the 0.6% performance degradation.

Bibliography

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

[2] Oracle. CDC Runtime Guide for the Sun Java Connected Device Configuration Application Management System version 1.0, Nov. 2005.

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

[4] Oracle. CDC: An Application Framework for Personal Mobile Devices. White Paper, June 2003.

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

[6] J. Aycock. A Brief History of Just-in-Time, ACM Computing Surveys (CSUR), 35(2):97-113, June 2003.

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

[8] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive Optimization in the Jalapeño JVM. In Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 47-65, Oct. 2000.

[9] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization, In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 32-43, June 1992.

[10] Oracle. CDC Reference Implementation, http://www.oracle.com/technetwork/java/javame/tech/overview-jsp-141092.html.

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

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

[13] M. Arnold, S.J. Fink, D. Grove, M. Hind and P. Sweeney. A Survey of Adaptive Optimization in Virtual Machine. In Proceedings of the IEEE, 93(2):449-466, Feb. 2005.

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

[15] A. Nartovich, A. Smye-Rumsby, P. Stimets, and G. Weaver. IBM Technology for Java Virtual Machine in IBM i5/OS, IBM Redbooks, Feb. 2007.

[16] S. Fink and F. Qian. Design, Implementation, and Evaluation of Adaptive Recompilation with On-Stack Replacement. In Proceedings of International Symposium on Code Generation and Optimization (CGO), pages 241-252, Mar. 2003.

[17] J. Smith and R. Nair. Virtual Machines, Morgan-Kaufmann, June 2005.

[18] S. Morris and A. Smith-Chaigneau, Interactive TV Standards: A Guide to MHP, OCAP, and JavaTV. Focal Press, Apr. 2005.

[19] D. Tsafrir. The Context-Switch Overhead Inflicted by Hardware Interrupts (and the Enigma of Do-Nothing Loops), In Proceedings of ACM Workshop on Experimental Computer Science (ExpCS), June 2007.

[20] D. Gu and C. Verbrugge. Phase-Based Adaptive Recompilation in a JVM. In Proceedings of IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pages 24-34, Apr. 2008.

[21] D. Buytaert, A. Georges, M. Hind, M. Arnold, L. Eeckhout, and K. Bosschere. Using HPM-Sampling to Drive Dynamic Compilation. In Proceedings of ACM SIGPLAN conference on Object-Oriented Programming Systems and Applications (OOPSLA), pages 553-568, Oct. 2007.

[22] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. The Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment. Concurrency and Computation: Practice and Experience, 17(5-6):617-637, Apr. 2005.

[23] V. Sundaresan, D. Maier, P. Ramarao, and M. Stoodley. Experiences with Multi-threading and Dynamic Class Loading in a Java Just-In-Time Compiler.

In Proceedings of IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pages 87-97, Mar. 2006.

[24] M. Arnold and B. Ryder. A Framework for Reducing the Cost of Instrumented Code. In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 168-179, June 2001.

[25] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot™ client compiler for Java 6. ACM Transactions on Architecture and Code Optimization (TACO), 5(1), Article 7, May 2008.

[26] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. Design and Evaluation of Dynamic Optimizations for a Java Just-in-Time Compiler. ACM Transactions on Programming Languages and Systems (TOPLAS), 27(4):732-785, July 2005.

[27] M. Namjoshi and P. Kulkarni. Novel Online Profiling for Virtual Machines. In Proceedings of ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), pages 133-144, July 2010.

[28] D. Jung, S. Moon, and H. Oh. Hybrid Compilation and Optimization for Java-based Digital TV Platform, ACM Transactions on Embedded Computing Systems (TECS), 13(2), Article 62, Jan. 2014.

[29] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney. Evaluating the Accuracy of Java Profilers, In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 187-197, June 2010.

[30] JProfiler. http://www.ej-technologies.com/products/jprofiler/overview.html.

[31] Java ME Phone Platform Development Project. https://java.net/projects/phoneme.

[32] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, M. schoeberl, and M. Mezini. Portable and Accurate Collection of Calling-Context-Sensitive Bytecode Metrics for the Java Virtual Machine, In Proceedings of the International Conference on Principles and Practice of Programming in Java (PPPJ), pages 11-20, Aug. 2011

[33] A. Bergel. Counting Messages as a Proxy for Average Execution Time in Pharo, In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), pages 533-557, July 2011

[34] G. Buttazzo. Research Trends in Real-Time Computing for Embedded Systems, ACM SIGBED Review - Special Issue on Major International Initiatives on Real-Time and Embedded Systems, 3(3):1-10, July 2006.

[35] Demonstration of the DTV Xlet Execution. http://altair.snu.ac.kr/DTV-demo.html.

[36] B. Cheng and B. Buzbee. A JIT Compiler for Android's Dalvik VM. http://www.google.com/events/io/2010/sessions/jit-compiler-androids-dalvik-vm.html

[37] A. Gal et. al, Trace-based just-in-time type specialization for dynamic languages. In Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI'09), Dublin, Ireland, June 2009.

[38] ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition

[39] ARM STRENGTHENS JAVA COMPILERS, http://www.arm.com/miscPDFs/10069.pdf

[40] Cortex-A8 Technical Reference Manual, revision: r3p1

[41] ECMAScript, http://www.ecmascript.org

[42] Improving ARM Code Density and Performance, http://www.arm.com/pdfs/Thumb2CoreTechnologyWhitepaper-Final4.pdf

[43] J. Jeon and S. Lee., Technical Trends of Mobile Web 2.0: What Next?, http://www.research.att.com/~rjana/MobEA2008/final/mobea2008_submission_6-1.pdf.

[44] M. Berndl et. al, Context Threading: A Flexible and Efficient Dispatch Technique for Virtual Machine Interpreters, Proceedings of the international symposium on Code generation and optimization, p.15-26, March 20-23, 2005

[45] Palm Inc., Palm webOS Platform Architecture,

http://developer.palm.com/index.php?option=com_content&view=article&id=1570, 2009.

[46] R. Kalden et. al, Wireless service usage and traffic characteristics in GPRS networks. In Proceedings of the 18th International Teletraffic Congress (ITC-18), Berlin, 2003.

[47] Rich Internet Applications,

http://www.adobe.com/platform/whitepapers/idc_impact_of_rias.pdf

[48] S. Pichai, Introducing the Google Chrome OS, The Official Google Blog, http://googleblog.blogspot.com/2009/07/introducing-google-chrome-os.html, Jul 2007

[49] SquirrelFishExtreme,

http://webkit.org/blog/214/introducing-squirrelfish-extreme

[50] SunSpider JavaScript Benchmark,

http://www2.webkit.org/perf/sunspider-0.9/sunspider.html

[51] TraceMonkey, https://wiki.mozilla.org/JavaScript:TraceMonkey

[52] V8, http://code.google.com/p/v8

[53] Widgets: What are Mobile Widgets?,

http://www.feedzilla.com/articles/widgets/what-are-mobile-widgets-/

[54] ECMAScript Language Specification, 5th edition, 2009.

[55] David Flanagan, “JavaScript: The Definitive Guide, 5th Edition”, O’Reilly Media, 2006.

[56] Seong-Won Lee, Soo-Mook Moon, Won-Ki Jung, Jin-Seok Oh, and Hyeong-Seok Oh, “Code Size and Performance Optimization for Mobile JavaScript Just-in-Time Compiler”, 14th Workshop on Interaction between Compilers and Computer Architectures (INTERACT-14), 2010.

[57] MG Siegler, “iPhone 3GS JavaScript Performance Blows Away Rivals, Approaches MacBook Speed”, http://techcrunch.com/2009/06/24/iphone-3gs-javascript-performance-blows-away-rivals-approaches-macbook-speed/, 2009.

[58] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek, “An analysis of the dynamic behavior of JavaScript programs”, ACM SIGPLAN conference on Programming language design and implementation (PLDI), 2010.

[59] Seong-Won Lee, Soo-Mook Moon, Seong-Moo Kim, “Enhanced Hot Spot Detection Heuristics for Embedded Java Just-in-Time Compilers”, ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES), 2008.

[60] Maciej Stachowiak, “Introducing SquirrelFish Extreme”, http://webkit.org/blog/214/introducing-squirrelfish-extreme, 2008.

[61] The WebKit Open Source Project, http://webkit.org/

[62] P. Ratanaworabhan et. al., “JSMeter: comparing the behavior of JavaScript benchmarks with real web applications”, WebApps'10 Proceedings of the 2010 USENIX conference on Web application development, 2010.

[63] Shelly Powers, “JavaScript Cookbook”, O’reilly Media, 2010.

[64] Oracle. CDC Runtime Guide for the Sun Java Connected Device Configuration Application Management System version 1.0. (Page 58), http://download.oracle.com/javame/config/cdc/cdc-opt-impl/cdc_runtime_guide.pdf

[65] JaegerMonkey, https://wiki.mozilla.org/JaegerMonkey

[66] Kevin Millikin et. al., “A New Crankshaft for V8”, http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html, December 2010.

[67] S. Lee and S. Moon. Selective Just-in-Time Compilation for Client-side Mobile JavaScript Engine. In CASES, 2011

[68] C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based On Prototypes. In OOPSLA, 1989.

[69] U. Holzle, C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In ECOOP, 1991.

[70] JavaScriptCore. http://trac.webkit.org/wiki/JavaScriptCore

Abstract

Some Optimizations for Accelerating Application Startups on Language Runtimes

Seong-Won Lee

School of Electrical Engineering and Computer Science

The Graduate School

Seoul National University

Java just-in-time compilers (JITC) often compile only hot methods since the compilation overhead is a part of the running time. This requires precise and efficient *hot spot detection*, which includes distinguishing hot methods from cold ones, detecting them as early as possible, and paying a small detection overhead. Hot spot detection is especially important in embedded applications since they show more of a start-up phase behavior of a regular application where methods are not executed heavily, so the hot methods are not definite. Since a long-running method is likely to be a hot method, we can detect a hot method by measuring its running time during interpretation. 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 such as Oracle's HotSpot heuristic. One problem is that 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 called *flow-sensitive runtime estimation* (FSRE) which can estimate the running time more precisely than others with a relatively low overhead. It only counts important bytecode instructions dynamically, but it can obtain the precise count of *all* interpreted bytecode instructions with a simple arithmetic calculation. We also propose a static analysis

technique to predict those hot methods which spends a huge execution time once invoked, so as to compile them at their first-invocation. Our experimental results show that these techniques can improve the performance by as much as an average of 7.4% compared to the HotSpot heuristic for the benchmarks when they run once, which is often regarded as showing the start-up phase behavior. Even for real embedded Java applications such as the digital TV (DTV) Java Xlet applications, our techniques can improve the user response time by an average of 7.1%.

Smart phone's full web browsing requires a high-performance JavaScript engine because JavaScript execution takes a non-trivial portion of the loading time for many web sites. The current wisdom of speeding up JavaScript engine is simply turning on its just-in-time compilation (JITC), which compiles JavaScript code to machine code on the fly and executes it instead of interpretation. One issue is that since mobile phones suffer from tight memory constraints, the JITC needs to keep a low memory footprint by generating small-sized machine code. In fact, many mobile CPUs support half-sized encoding for small code size with small performance degradation, as in the ARM Thumb2. This paper describes our code generation and optimization for a mobile JavaScript JITC in the Webkit's SquirrelFish Extreme (SFX) for the ARM Thumb2. We try to generate as many 16-bit instructions as possible and reduce the data area, while strictly following the code generation guidelines of the SFX, which actually leaves little room for code optimization. Our experimental results show that we could reduce the code size by 29% with a performance degradation of 3.5%, compared to the ARM version of the SFX.

On the other hand, we found that JITC actually increases the loading time tangibly for some JavaScript-heavy web pages compared to interpretation, while it can still reduce the running time for JavaScript benchmarks. We observed that the web page JavaScript behaves differently from the benchmark JavaScript in the sense that hot spots rarely exist. This would lower the reuse ratio of the compiled machine code, making the compilation overhead higher than its benefit. This is especially true for a JavaScript engine which compiles all executed functions at their first invocation, as the SFX engine in the WebKit. In order to overcome this problem, we introduce *selective compilation* to the SFX engine so as to compile only hot functions detected during interpretation. This reduces the slowdown of the SFX for web page

JavaScript, while accelerating JavaScript benchmarks. However, selective compilation for web page JavaScript shows a different behavior from other environment, and we discuss it.

JavaScript execution during web page loading spends much of its time for executing runtime services of the JavaScript engine, especially for accessing properties of objects. One problem is that many of these object properties are accessed for the first time during web page loading. This makes the bytecode for these first-time accesses be executed without optimizations such as just-in-time compilation or inline caching, while suffering from the overhead of creating property maps for each access or calling redundant runtime services. Therefore, we propose *super-bytecode* for merging a sequence of property accesses for the same object, with optimized runtime services to accelerate them. It also improves inline caching. In addition, we suggest bytecode chaining for replacing some redundant runtime services to direct accesses with previously cached information. Our preliminary experimental results show that the super-bytecode and bytecode chaining accelerate the loading of some web pages.

Keywords : Just-in-Time Compiler, Hot Spot Detection, Code Size Optimization, Selective Compilation, Object Property Access, Bytecode Level Optimization

Student Number : 2007-21042