The Impact of Java Applications at Microarchitectural Level from Branch Prediction Perspective

Adrian Florea, Arpad Gellert, Lucian Vinţan, Marius Velţan

Adrian Florea, Arpad Gellert, Lucian N. Vinţan
"Lucian Blaga" University of Sibiu, Computer Science Department
Emil Cioran Street, No. 4, Sibiu 550025, Romania
E-mail: {adrian.florea,arpad.gellert,lucian.vintan}@ulbsibiu.ro

Marius N. Velţan
SC IMC Information Multimedia Communications SRL Sibiu, Product Development Department
Cristian Street, No. 21, Sibiu 550073, Romania
E-mail: marius.veltan@im-c.de

Abstract: The portability, the object-oriented and distributed programming models, multithreading support and automatic garbage collection are features that make Java very attractive for application developers. The main goal of this paper consists in pointing out the impact of Java applications at microarchitectural level from two perspectives: unbiased branches and indirect jumps/calls, such branches limiting the ceiling of dynamic branch prediction and causing significant performance degradation. Therefore, accurately predicting this kind of branches remains an open problem. The simulation part of the paper mainly refers to determining the context length influence on the percentage of unbiased branches from Java applications, the prediction accuracy and the usage degree obtained using a *Fast Path-Based Perceptron* predictor. We realize a comparison with C/C++ application behavior from unbiased branches perspective. We also analyze some Java testing programs, built using *design patterns* or including inheritance, polymorphism, backtracking and recursivity, in order to determine the features of indirect branches, the arity of each indirect jump and the prediction accuracy using the Target Cache predictor.

Keywords: object-oriented programs, branch prediction, indirect jumps/calls, unbiased branches, benchmarking

1 Introduction

As the Internet evolves and becomes increasingly popular, the need for a portable programming language becomes increasingly important. Providing a common language interface Java technology, that includes Java language, Java Virtual Machine (JVM) and Java API, allows programs to be used across a wide range of machine platforms from browsers, e-commerce, financial and bioinformatics applications on desktop computing systems to software for real-time embedded systems (intelligent mobile phones, palms, PDA, etc.). The portability, the object-oriented and distributed programming models, multithreading support and *automatic garbage* collection are features that make Java very attractive for application developers. In addition to portability, security and ease of application development has made it very popular with the software community. According to Sun Microsystems, in 2007 there were more than 5 billion java enabled devices (desktops, phones, cards, settop boxes, etc), from among 2.1 billions are phone handsets or PDA. Also, more than 6 million professional java developers are programming these devices [1].

Besides, the last decade is characterized by the advance of visual or object-oriented applications and the portability trend of many of them, as well as the usage of Dynamically-Linked Libraries. To support polymorphism the object-oriented languages such as Java, C#, and C++ include dynamically-dispatched function calls (i.e. virtual functions) whose targets are not known until run-time because they depend on the dynamic type of the object on which the function is called. Virtual function calls are implemented using indirect jump/call instructions in the instruction set architecture (ISA). From microarchitectural point of view object-oriented programming techniques exercise different aspects of computer architecture to support the object-oriented programming style [2], [3]. The object-oriented languages have significantly more indirect jumps than procedural languages. In addition to virtual function calls, indirect jumps are commonly used in the implementation of programming language constructs such as switch-case statements (with more than five options [3]), jump tables, indirect calls through function pointers, and interface calls [4].

In current pipelined processor designs a particularly difficult challenge consists in target prediction for indirect jumps and calls. Because the target of an indirect jump (call) can change with every dynamic instance of that jump, predicting the target of such an instruction is really difficult. Such hard-to-predict jumps not only limit processor performance and cause wasted energy consumption but also contribute significantly to the performance difference between procedural and object-oriented languages. Simulations made on Intel Core2 Duo processor analyzing suggestive applications such Matlab, Cygwin, Excel, Firefox, Winamp and InternetExplorer show that about 41% of all jump mispredictions are due to indirect jumps [5]. Besides indirect jumps/calls, another class of hard-to-predict branches consists in unbiased branches. In two of our previous papers [6], [7] we found a minority of dynamic conditional branches showing a low degree of polarization towards a specific prediction context since they tend to shuffle between taken and not-taken. We called them unbiased branches and we have demonstrated that, irrespective of the prediction information length and type, used in the state of the art branch predictors, these branches are characterized by low prediction accuracies (at average about 70%). We have demonstrated that actual programs have a significant fraction of such difficult to predict branches that severely affect prediction accuracy leading to performance degradation and additional power consumption, which are very important especially in the embedded systems.

The main goal of this paper consists in pointing out the impact of Java applications at microarchitectural level from two perspectives: unbiased branches and indirect jumps/calls, such branches limiting the ceiling of dynamic branch prediction and causing significant performance degradation. Therefore, accurately predicting this kind of branches remains an open problem. Following our aim, first we extend our tool Advanced Branch Prediction trace driven Simulator (ABPS) [8] for analyzing (detecting and predicting) unbiased branches from SPEC JVM98 benchmarks [9], a suite of eight standardized Java applications proposed to better understand where performance bottleneck exist in the Java Virtual Machines and what optimizations are possible. We determine the context length influence on the percentage of unbiased branches from Java applications, the prediction accuracy and the usage degree obtained using a *Fast Path-Based Perceptron* (FPBP) predictor [10]. Also, we realize a comparison with the behavior of C/C++ applications from unbiased branches perspective.

The differences between object-oriented and procedural applications from execution viewpoint on ILP processors guided us to analyze some Java testing programs built using *design patterns*. Based on these Java applications we illustrated that the two computer science components (software and hardware) are just apparently *"disjointed"*. Thus, in section 4 we describe six well-known design patterns [11], [12] and applications developed based on them. Additionally, we created four simple applications that include inheritance, polymorphism, backtracking and recursivity. These programs were *execution-driven* simulated on "Dynamic SimpleScalar" environment (DSS) [13] in order to determine the features of indirect branches, the arity of each indirect jump and the prediction accuracy using the Target Cache predictor.

The remainder of this paper is organized as follows. In section 2 we illustrated comparatively the

C/C++ and Java languages, emphasizing especially the differences between them from the execution viewpoint on host architectures. Section 3 describes the implemented predictors used for studying unbiased branches and indirect jumps/calls. Section 4 includes simulation methodology and presents the benchmarks used for simulation. In section 5 we illustrate the experimental results obtained using our developed simulator. Finally, section 6 suggests further work directions and concludes the paper.

2   Some differences between C/C++ and Java languages

C++ programming language was designed and implemented as an extension of C language that supports data abstraction, object-oriented concepts and generic programming, dedicated for desktop programming. Unfortunately, it inherited also all problems from C. Java was created as a new object-oriented and distributed language, without compatibility constraints, mainly dedicated for small electronic devices that embeds networking capabilities and has cross platform portability and also for Internet applications programming. In Java were removed all the procedural programming concepts that proved to be dangerous or unsecured.

Unfortunately, the portability leads to decreasing the execution speed of Java programs. Thus, it was observed that the interpreted Java bytecode is 30 times slower than an optimized C++ code. However, just in time (JIT) compilers could produce considerable performance improvements over interpretation (20 times) by removing dispatch overhead and applying some optimizations to the generated code. The classes used by server applications are executed using JIT compilers in order to obtain high processing performance. Though, the memory required by JIT compilers (additional space for the generated native code, for profiling and other data structures) is very expensive especially in the case of embedded systems. For these systems are preferred native Java processors that use small memory and have a small power consumption, and last but not least, have small costs, easy to support by any consumer. From this reason, the modern researches try to optimize the Java virtual machines. Using server applications, when Sun JVM was updated from the 3rd version to the 6th version the processing performance was improved with 344%. Similarly, using desktop applications, the performance increases but not very spectacular [1]. Nowadays Java virtual machines use even *dynamic metrics* that optimize the JVM taking into account not only the host processor but also the applications' behavior. There are applications that run better if they are written in Java than in C++. Simulation results on the SciMark benchmark have shown that Java runs with 4% quicker than C++, and LINPACK benchmarks are slower with only 2% in Java against C++. Also, the Garbage Collector is much quicker than dynamic memory allocation / de-allocation realized with the *malloc* and *free C++* instructions [1]. Thus, after releasing the last JVM version (JRE 6.0), Sun Microsystems engineers claim that they destroyed the myth *interpreted language means not performing*.

3   Predictors implemented for studying unbiased branches and indirect jumps/calls

For indirect branch prediction we considered a tagged Target Cache (TC) predictor first introduced by Chang [15]. The Target Cache improves the prediction accuracy for indirect branches by choosing its prediction from the last *N* targets of the indirect branch that have already been encountered. When an indirect jump is fetched, both the PC and the *globalHR* (a history register that retains the behavior of last *k* conditional branches) are used to access the TC for predicting the target address. As the program executes, the TC records the target for each indirect jump target encountered. Our proposed scheme uses for set selection the least significant bits of the word obtained by hashing (XOR) the indirect jump's address (PC) and the global history (*globalHR*). The most significant bits of this hashing form the *Tag*. In the case of a hit in the TC the predicted target consists in the corresponding address belonging to that TC set (field *Adr* from Figure 1). In the case of a misprediction, (the Tags coincide but the target addresses

differ) after the indirect branch is resolved, the Target Cache entry is updated with its real target address. We have implemented and simulated a *P*-way set associative TC, where P=1, 2, 4 like that presented in Figure 1. In the case of a miss in the TC the prediction is considered wrong, it does not provide any value and a new entry updated with the proper *tag* and *target* is added to the respective set, according with the implemented LRU replacement algorithm.
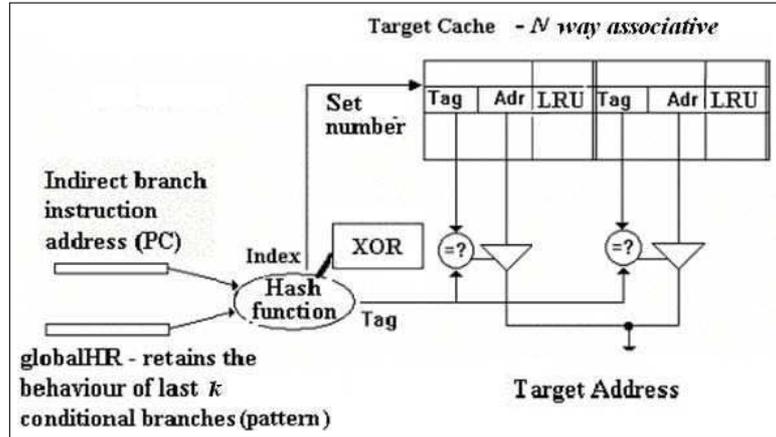
Figure 1: The structure of Target Cache predictor

The most accurate single-component branch predictors in the literature are neural branch predictors [10], [16] and [17]. Their main advantages consist in the possibility of using longer correlation information at linear cost. The *Perceptron* predictor - the simplest neural branch predictor - keeps a table of *weight vectors* (small integers that are learned through the *perceptron* learning rule) [16]. As in global two-level adaptive branch prediction, a shift register records a global history of conditional branche outcomes, recording *true* for *taken*, or *false* for *not taken*. To predict a branch outcome, a weight vector is selected by indexing the table with the branch address modulo the number of weight vectors. The dot product of the selected vector and the global history register is computed, where *true* in the history represents 1 and *false* represents -1. If the dot product is at least 0, then the branch is predicted taken, otherwise it is predicted not taken. Once the perceptron output has been computed, the training algorithm starts: it increments the *i*-th correlation weight when the branch outcome agrees with the *i*-th bit from the global branch history shift register and decrements the weight otherwise. Unfortunately, the high latency of the perceptron predictor and its impossibility to predict the linearly inseparable branches makes it impractical yet for hardware implementation. In order to reduce the prediction latency, the *Fast Path-based Perceptron* [10] chooses its weights for generating a prediction according to the current branch's path, rather than according to the branch's PC and history register. The prediction latency is hidden due to the speculative calculation of the perceptron's output. Let PC_0 be the current branch address, and PC_i be the ith most recent branch address in the *path history*. For the perceptron, each weight is chosen with the same index based on PC_0. For the FPBP, each weight w_i is chosen based on an index derived from PC_i. This provides path history information that can improve prediction accuracy, and spreading out the weights in different entries also helps to reduce the impact of inter-branch aliasing. To implement the FPBP, the lookup phase is actually pipelined over many stages based on the overall branch path / global history length. Undertaken from [18], Figure 2 exposes a very intuitive scheme of *Fast Path-based Perceptron*. For a branch at cycle *t*, the FPBP starts the prediction at cycle *t - h* using PC_h. For each cycle after *t - h*, the FPBP computes partial sums of the dot-product of weights vector and global history register. Pipeline stage *i* contains the partial sum for the branch prediction that will be needed in *i* cycles. At the very end of the pipeline, the critical lookup latency consists of looking up the final weight and performing the final addition.
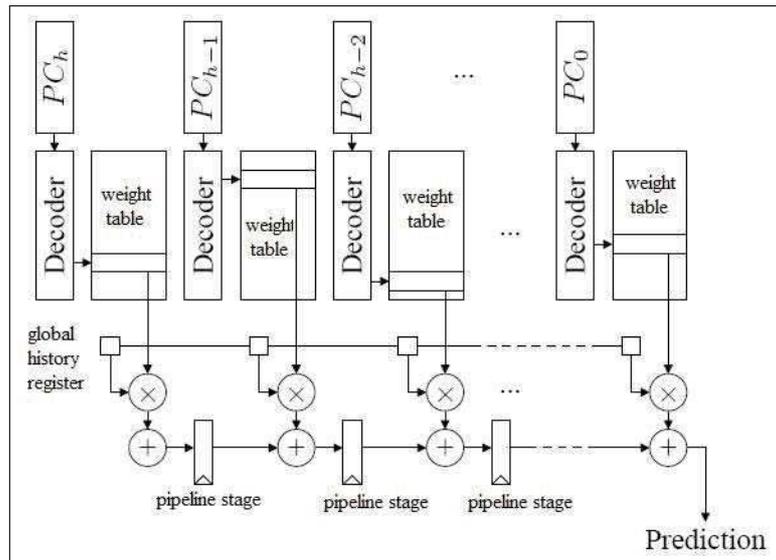
Figure 2: The structure of Fast Path-based Perceptron predictor

4 Simulation Methodology and Benchmarking

Related to our evaluations on procedural programs, we collected results from different versions of SPEC benchmarks: 3 integer (*li, go, cc1*) and 4 floating point (*applu, apsi, fpppp, hydro*) SPEC'95 benchmarks. From the integer SPEC2000 suite, we simulated 8 benchmarks (*gzip, b2zip, parser, crafty, gap, gcc, twolf* and *mcf*). We also simulated some SPEC'95 benchmarks in order to compare their behavior with that of the more recent SPEC2000. All these benchmarks cover a lot of applications ranging from compression (text/image) to word processing, from compilers and architectures to games enhanced with artificial intelligence, etc. These programs were selected based on their characteristics: a high number of indirect branches and high target entropy.

Recall that the majority of indirect branches are generated by object oriented programs. In software engineering, after structured programming and object orientation, one of the most important innovation with impact on the art of constructing software systems were *design patterns* [11]. It is hard to understand and practice effective object orientation without being conscious of design patterns and their suitability for the problem at hand. According to [19] a design pattern represents a formal way of documenting a solution to a design problem in a particular field of expertise. In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design. Using real-life scenarios helps in understanding programming abstraction to some extent but do not provide the deep insight that one gets by actually seeing these abstractions. A design pattern is not a finished design that can be transformed directly into code. Patterns show how to build systems with good object oriented design qualities. Most patterns allow some part of a system to vary independently of all other parts. Patterns provide a share language that can maximize the value of communication between developers.

For indirect branch analysis of Java applications we simulated six design pattern applications [11], [12] and also four other simple object-oriented programs: *Jbytemark* that sorts arrays of values through many sorting methods, *Queens_5* that uses inheritance and polymorphism for solving different problems through recursive backtracking, *Switch_06* that illustrates the indirect jump generation from switch/case statements and *Simple_Inheritance* that uses polymorphism and treats uniformly the heterogenic array of objects. Table 1 illustrates the characteristics of the design pattern applications [11], [12] used for indirect branch analysis.

The results related to the indirect branch analysis illustrated in chapter V were obtained after simula-

Table 1: Design Pattern characteristics

Design Pattern	Application	Characteristics
Strategy	DuckSimulator	It is a *behavioral* pattern that defines a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently on the clients that are using it. Example: *Adventure Game*
Observer	WeatherStation	It is a *behavioral* pattern that defines a one-to-many dependency between objects so that when one object's state changes, all its dependents are notified and updated automatically. Example: *Newspaper subscription*
Decorator	StarbuzzCoffee	It is a *structural* pattern that dynamically attaches additional responsibilities to an object. Decorators provide a flexible alternative to sub-classing for extending functionality. Example: *Dressing a character in a social game*
Factory Method	PreparingPizza	It is a *creational* pattern that defines an interface for creating an object, but let subclasses decide which class to instantiate. Example: *Factory methods*
Template Method	ServeingDrinks	It is a *behavioral* pattern that defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. Example: *Searching in a database*
State	Gumball Machine Test	It is a *behavioral* pattern that allows an object to alter its behavior when its internal state changes. The object will appear to change its class. Example: *Automated Teller Machine*.

tion on 2.4 GHz Pentium IV microprocessor under Microsoft WindowsXP operating system, disposing of Cygwin emulator. Taking into account that the simulation time of over 600.000.000 dynamic instructions from aforementioned Java benchmarks vary between 35 minutes (Queens_5) and more than 24 hours (jBytemark) we chose to limit the simulation to maximum 10 billion instructions. From the SPEC'95 suite we simulated 100 millions instructions, from the SPEC2000 set we simulated 500 millions instructions, whilst all Java applications were completely simulated (except JBytemark benchmark), the total number of instructions ranging between 600 millions and 4290 millions instructions, respectively.

For indirect jump/call analysis we developed a cycle-accurate execution driven simulator (*sim-jindir* [12]), derived from the *sim-cache* simulator of the Dynamic SimpleScalar toolset [13]. We modified it to incorporate the indirect jump predictor proposed in Section 3 in order to measure the number of indirect (static / dynamic) branches, the arity of each indirect jump and to predict targets of indirect jumps and calls, respectively. DSS represents a standardized software tool used to simulate Java programs executed on a Java Virtual Machine (in our case IBM Jikes Research Virtual Machine) and was created to extend the SimpleScalar toolset [20]. Jikes RVM is a virtual machine that runs Java programs by compiling them to native code at runtime. It comes with two Java bytecode to native code compilers. The fast baseline compiler does not perform any optimizations. The optimizing compiler performs a complete set of optimizations most importantly *inlining* and *register allocation*. The *sim-jindir* provides a wider variety of configuration options. We can vary the TC associativity degree, the number of sets of TC, the maximum number of instructions executed, the simulated benchmark, and the platform used for simulation.

Table 2 illustrates the characteristics of standardized SPECJVM98 benchmarks [21] used for unbiased branch analyze.

ABPS is a trace driven simulator used to analyze (detect and predict) unbiased branches. It is written in Java and includes state of the art branch predictors such as Fast Path-based Perceptron predictor. This request as inputs parameters: the number of entries in prediction table, the global history length, the threshold value used by the learning algorithm, number of bits for storing the weights. ABPS can detect unbiased branches or predict all branches or only those that are unbiased. The ABPS simulation results consist in four important metrics. The prediction accuracy is the number of correct predictions divided to total number of dynamic branches. We compute also a confidence metric that represents the total cases when the prediction was correct and the perceptron did not need to be trained (the magnitude of perceptron output was greater than the threshold) divided to the total number of correct predictions. While the first two have impact on processor's performance, the next two metrics have direct influence on transistors' budget and integration area (the number of perceptrons used in the prediction process and the saturation degree of perceptrons). The saturation degree represents the percentage of cases when the weights of perceptrons cannot be increased / decreased because they are saturated. If the last two metrics are quite low means that the perceptrons are underused.

5 Experimental Results

Tables 3 and 4 illustrate comparatively the percentage of static / dynamic (indirect) branches within the tested programs.

Figures 3 and 4 illustrate the arity of indirect branches from static and dynamic points of view, respectively. The arity means the number of distinct dynamic targets of the same indirect jump/call. This can be determined either after simulations or knowing profile information (identifying branch class) either estimated through source code level analysis. After this process, monomorphic jumps/calls (having a single target) can be successful predicted by a predictor without history (BTB), yet, for polymorphic branch prediction (with two or more distinct targets) additional information are necessary.

Analyzing Tables 3 and 4 and Figures 3 and 4, we observe some significant differences related to the behavior of Java applications against C/C++ applications at microarchitectural level, from indirect

Table 2: SPEC JVM98 characteristics

Benchmark	Characteristics
227_mtrt	It represents a modification of 205_raytrace, a ray tracer that creates a pictorial scene portraying a dinosaur. It uses a multi-threaded driver, where the threads render the scene from an input file.
202_jess	This benchmark is an expert system shell written entirely in Java. The intent of Jess is to give Java applets the ability to "reason" by continuously applying a set of rules. The workload used in the benchmark solves a set of puzzles.
201_compress	Represents Java version of the 129.compress benchmark from the SPEC'95 benchmark suite, but improves upon that benchmark in that it compresses real data from files using a modified Lempel-Ziv method (LZW). It is highly recursive.
209_db	It performs multiple database operations (insertion, removing, finding, sorting) on a 1 MBytes memory resident database.
222_mpegaudio	This benchmark is an application that decompresses audio files defined by the ISO MPEG Layer-3 audio specification. The MP3 encoding technique allows data compression of digital audio signals up to a factor of 12, without losing sound quality as perceived by the human ear. The workload consists of about 4MB of audio data.
228_jack	It represents a Java parser that is based on the Purdue Compiler Construction Tool Set, in fact an earlier version of JavaCC compiler. The workload consists of a file named jack.jack, which contains instructions used for the generation of jack itself. This is fed to jack so that the parser generates itself multiple times (highly recursive).
213_javac	It represents the Java compiler from the JDK 1.0.2.

Table 3: Statistics about *static* (indirect) branches from the analyzed benchmarks

Testing programs (1)	Average of indirect static branches (2)	Average of static branches (3)	Average percentage of indirect static branches (4) = (2)/(3)*100%
SPEC 95	59	4759	2.15%
SPEC 2000	57	8187	0.86%
Design Patterns	6765	16619	40.69%

Table 4: Statistics about *dynamic* (indirect) branches from the analyzed benchmarks

Testing programs (1)	Average of indirect dynamic branches (2)	Average of dynamic branches (3)	Average percentage of indirect dynamic branches (4) = (2)/(3)*100%	Maximum percentage of indirect dynamic branches
SPEC 95	432041	13807188	2.47	5.63%
SPEC 2000	790826	95218337	0.78	2.03%
Design Patterns	17904385	93676540	18.82	20.41%
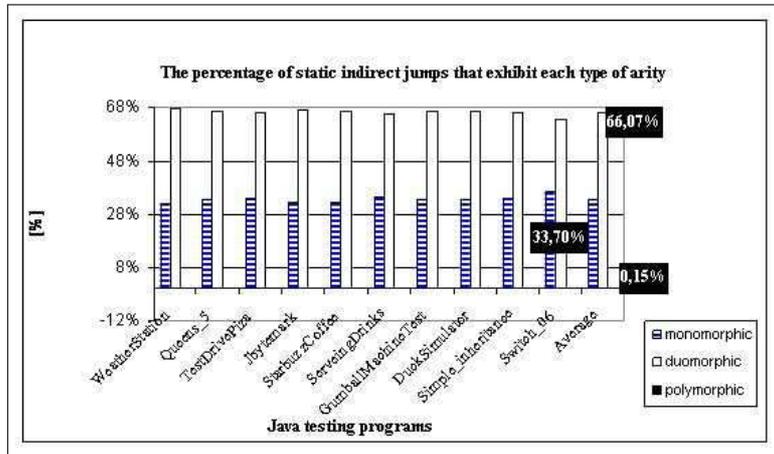
Figure 3: The arity of indirect branches from static point of view
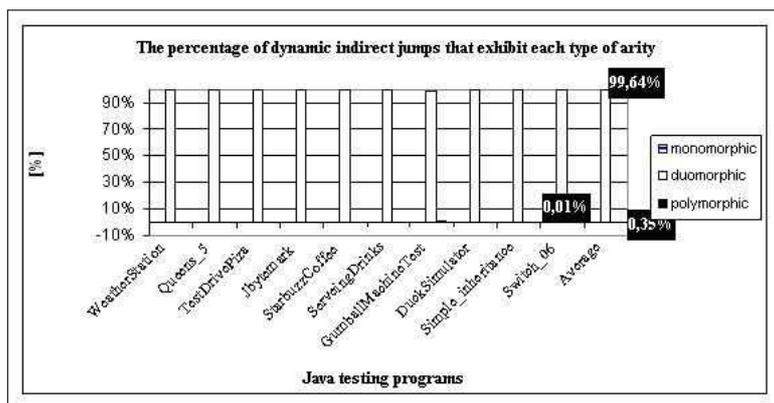
Figure 4: The arity of indirect branches from dynamic point of view

branches perspective: first, the percentage of dynamic indirect branches from Java testing programs (maximum 20.41%) is much higher than that from procedural or object oriented C/ C++ programs (maximum 5.63%, see also [22]). The results are similar with those reported in [23]. According to Tao, the performance results are different depending on the JVM mode of execution. In interpreter mode, 19.5% of all dynamic branches are indirect branches and 11.8% of all branches are polymorphic indirect branches. In JIT compiler mode, 12.4% of dynamic branches are indirect branches and only 5% are polymorphic branches. Second, the significant percentage of duomorphic branches from Java programs, recommends using a minimum 2-way associative Target Cache structure for indirect branch prediction. The very small percentage of dynamic polymorphic indirect branches from our proposed Java programs could be due to the fact that these programs do not represent standardized benchmarks but only simple testing programs. However, we consider that there are two major reasons for the high number of indirect branches targets within Java applications: 1) as an object-oriented programming language, Java promotes a polymorphic programming style in which late binding of subroutine invocations is the main instrument for modular code design. With the help of virtual method table, the implementation chosen for Java interpreters and compilers, it executes an indirect branch for every polymorphic call. More than that, in Java, instance methods are virtually declared by default. If they are not explicitly declared final, they can be overridden in subclasses. 2) Switch/case statements (in the bytecode translation routine of the interpreter) and indirect function calls through pointers (calls to the dynamically shared native interface libraries) from runtime interpretation and JIT compilation of bytecodes performed by the Java Virtual Machine are subject to high indirect branch frequency.
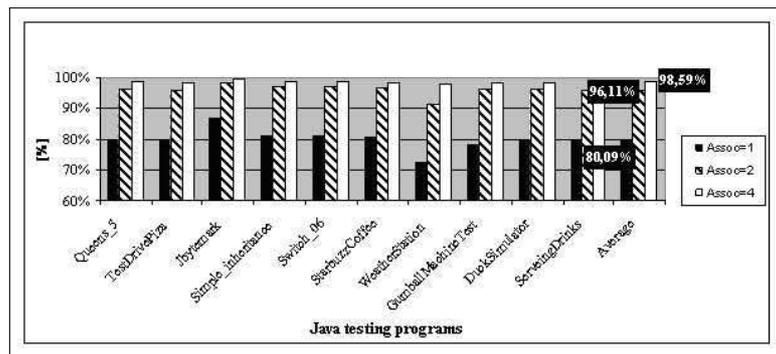
Figure 5: The indirect branch prediction accuracy using a Target Cache Predictor with 512 entries

From Figure 5 we observe that indirect branch prediction accuracy increases as the associativity degree of TC structure increases. The improvement is significant for a 2-way associative TC against a direct mapped one (20%) whilst the improvements diminish to 2.58% for a 4-way associative TC compared with a 2-way associative one. The results from Figure 5 are correlated with those from Figure 4 which show a high percentage of duomorphic and an insignificant percentage of polymorphic indirect branches. The improvement of prediction accuracy from 96.11% to 98.59% is due to the miss conflict and capacity miss accesses at a 2-way associative TC (that are solved by increasing the associativity degree), but also to the smaller percentage of polymorphic indirect branches.

Analyzing procedural programs (SPEC'95 and SPEC2000), we have shown in [24] that the best prediction accuracy of indirect branches obtained using a 8-way associative Target Cache with 128 entries was 88.97%, for Target Cache capacities greater or equal with 256 entries the prediction accuracy becoming saturated.

Figure 6 exhibit comparatively the behavior of procedural benchmarks (SPEC2000) versus object-oriented benchmarks (SPEC JVM98) from unbiased branches perspectives. Repeating the detection methodology for a length-ordered set of contexts used by us in [25] it could be observed how the number of unbiased branches decreases within both procedural and object-oriented applications. On
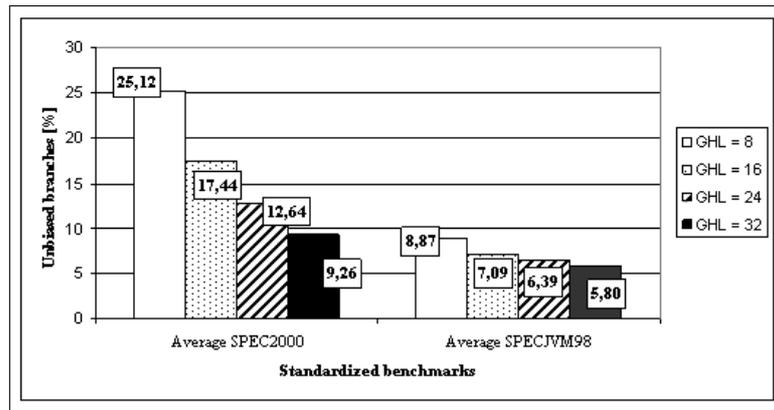
Figure 6: Reducing the number of unbiased branches by increasing global history length (GHL)

the SPEC2000 benchmarks the percentage of unbiased branches decreases in average from **25.12%** to **9.26%**. We consider that this value is still too high and further investigations are required. For the SPEC JVM98 benchmarks the percentage of unbiased branches decreases from **8.87%** to **5.80%** (almost half compared to the SPEC2000 integer benchmarks).
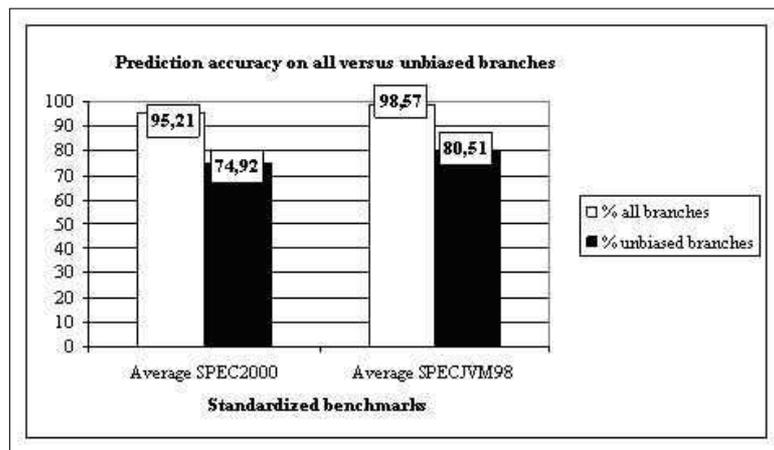
Figure 7: Fast Path-Based Perceptron prediction accuracies on SPEC2000 versus SPEC JVM98 benchmarks using a table of 1024 entries

In Figure 7 we considered, comparatively on SPEC2000 versus SPEC JVM98 benchmarks, the impact of unbiased branches on a FPBP predictor with a table of 1024 entries considering a global history register of 32. We achieved on the SPEC2000 suite an average prediction accuracy of 95.21% on all branches but the impact of unbiased branches still remained significant at 74.92%. However, in Java benchmarks, the smaller percentage of unbiased branches has a lower impact on prediction accuracy. The prediction accuracy obtained on all branches is very high - 98.57% at average and even unbiased branches are predicted more accurately than those from the SPEC2000 integer benchmarks (where 11 of 12 are procedural applications).

Based on the number of perceptrons used, we determined the usage degree of perceptron table. This metric will directly affect the prediction accuracy and indirectly the processing performance lost. The reduced usage degree (39.77% in average on the SPEC2000 benchmarks) proves a high interference degree in the perceptrons table, further diminishing the prediction accuracy, the only advantage being the reduced integration area cost. However, this aspect has been not observed on the SPEC JVM98 benchmarks where the perceptron table is almost entirely exploited (99.65% in average).

6   Conclusions and Further Work

Java technology from software for embedded systems to commercial and research applications and also, visual, interactive and desktop applications are characterized by high percentage of indirect branches. Unfortunately, the prediction accuracy of indirect branches is still very low because many indirect branches have multiple targets that are difficult to predict even with specialized hardware. This paper shows that besides indirect jumps, the unbiased branches represent another class of hard to predict branches in Java applications that limits the ceiling of dynamic branch prediction and causes performance degradation and additional power consumption.

Studying the impact of Java applications at microarchitectural level from branch prediction perspectives we concluded:

- The percentage of dynamic indirect branches from Java testing programs (maximum 20.41%) is much higher than that from procedural or object oriented C/ C++ programs (maximum 5.63%).

- The significant percentage of duomorphic indirect branches ($\geqslant$90%) from Java programs recommends using a minimum 2-way associative Target Cache structure for indirect branch prediction.

- The prediction accuracy improvement is significant for a 2-way associative TC against a direct mapped one (20%) whilst the improvements diminish to 2.58% for a 4-way associative TC compared with a 2-way associative one.

- For Java benchmarks the percentage of unbiased branches decreases from 8.87% to 5.80% (almost half compared to the C/C++ integer benchmarks).

- For Java benchmarks, the smaller percentage of unbiased branches has a lower impact on prediction accuracy. The prediction accuracy obtained on all benchmarks is very high (98.57% in average) and even unbiased branches are predicted more accurately than those from C/C++ integer benchmarks. The lower percentage of unbiased branches occurred in Java benchmarks could be determined by the reduced number of conditional branches from object oriented benchmarks compared with procedural applications. It is well known that object-oriented programs contain many methods with few instructions, and fewer conditional branches implicitly.

Hardware-software interface optimizations are not possible without a deeper understanding process that requires an integrated vision about multiple stages of information processing, coded at different semantic levels. A solution could consist in developing some "*semantic predictors*", based on High Level Language (HLL) information whose importance related to the generation of indirect jumps (polymorphism, indirect function calls, etc.) is proved by us. This might be a completely new approach in branch prediction domain, where HLL semantics are often hidden. In order to efficiently use such information we consider it will be necessary to have a significant amount of compiler support.

Bibliography

[1] Gosling J., *Java: a Tour of the Landscape*, Sun Technology Days at a Glance, Frankfurt, Germany, 3-5 December, 2007.

[2] Calder B., Grunwald D., Zorn B., *Quantifying behavioral differences between C and C++ programs*, Journal of Programming Languages, Volume 2, Issue 4, 1994, pp. 313-351.

[3] Florea A., Vinţan L., Mihu I.Z., *Understanding and Predicting Indirect Branch Behavior, Studies in Informatics and Control*, Volume13, Issue 1, March 2004, pp. 61-82, National Institute for Research and Development in Informatics, Bucharest.

[4] Alpern B., Cocchi A., Fink S., Grove D., Lieber D., *Efficient implementation of Java interfaces: Invoke interface considered harmless*, In Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, pp. 108-124, October 14-18, 2001, Tampa, Florida, USA.

[5] Joao J.A., Mutlu O., Kim H., Agarwal R., Patt Y., *Improving the Performance of Object-Oriented Languages with Dynamic Predication of Indirect Jumps*, ACM SIGOPS Operating Systems Review, Volume 42, Issue 2, March 2008, pp. 80-90.

[6] Vinţan L., Gellert A., Florea A., Oancea M., Egan C. *Understanding Prediction Limits through Unbiased Branches*, Lecture Notes in Computer Science, Advances in Computer Systems Architecture, vol. 4186, pp. 480-487, 2006, Springer-Verlag Berlin.

[7] Gellert A., Florea A., Vinţan M., Egan C., Vinţan L. *Unbiased Branches: An Open Problem*, Lecture Notes in Computer Science, Advances in Computer Systems Architecture, vol. 4697, pp. 16-27, 2007, Springer-Verlag Berlin / Heidelberg.

[8] Radu C., Calborean H., Crapciu A., Gellert A., Florea A., *An Interactive Graphical Trace-Driven Simulator for Teaching Branch Prediction in Computer Architecture*, The 6th EUROSIM Congress on Modeling and Simulation, September 9-13, 2007, pp. 58 (6 pg.), Ljubljana, Slovenia.

[9] *SPEC JVM98 benchmarks*, http://www.spec.org/jvm98/

[10] Jiménez D. *Fast Path-Based Neural Branch Prediction*, Proceedings of the 36th Annual International Symposium on Microarchitecture, December 3-5, 2003, pp. 243-252, San Diego, CA, USA.

[11] Freeman E., Freeman E., Sierra K., Bates B. *Head First Design Patterns*, O'Reilly Publishing House, First Edition, October 2004.

[12] Veltan N.M. *The Interaction of Java Programs at Microarchitectural Level from Branch Prediction Viewpoint (in Romanian)*, MSc Thesis, 'Lucian Blaga' University of Sibiu, Computer Science Department, Romania, 2008.

[13] *The University of Massachusetts Amherst and the University of Texas - Dynamic SimpleSscalar*, 2004. http://www-ali.cs.umass.edu/DSS/index.html.

[14] Lindholm T., Yellin F., *The JavaTM Virtual Machine Specification*, Sun Press Publishing Hall, 1999.

[15] Chang P.Y., E. Hao, Y.N. Patt., *Target Prediction for Indirect Jumps*, Proceedings of the International Symposium on Computer Architecture, 1997.

[16] Jiménez D., Lin C. *Neural Methods for Dynamic Branch Prediction*, ACM Transactions on Computer Systems, Vol. 20, Issue 4, November 2002, pp. 369-397, New York, USA.

[17] Jiménez D., *Idealized Piecewise Linear Branch Prediction*, Journal of Instruction-Level Parallelism, Vol. 7, April, 2005, pp. 1-11.

[18] Loh G. H., Jiménez D., *Reducing the Power and Complexity of Path-Based Neural Branch Prediction*, Proceedings of the 5th Workshop on Complexity Effective Design (WCED5), pp. 1-8, June 5, 2005, Madison, WI, USA.

[19] *http://en.wikipedia.org/wiki/Design_pattern*

[20] Burger D., Austin T., *The SimpleScalar Tool Set*, Version 2.0, University of Wisconsin Madison, USA, Computer Science Department, Technical Report no. 1342, June, 1997.

[21] Bowers K. R., Kaeli D., *Characterizing the SPEC JVM98 benchmarks on the Java virtual machine*, Technical Report, Northeastern University, ECE Department, Computer Architecture Group, 1998, pp. 1-20, Boston, Massachusetts, USA.

[22] Florea A., *The dynamic values prediction in the next generation microprocessors (in Romanian)*, MatrixRom Publishing House, 2005, Bucharest.

[23] Tao Li, Lizy K. John, *Adapting Branch-Target Buffer to Improve the Target Predictability of Java Code*, ACM Transactions on Architecture and Code Optimization, Volume 2, Issue 2, June 2005, pp. 109-130.

[24] Florea A., Vinţan L., *Advanced techniques for improving indirect branch prediction accuracy*, Proceedings of 19th European Conference on Modelling and Simulation, June 2005, pp. 750-759, Riga, Latvia.

[25] Florea A., Radu C., Calborean H., Crapciu A., Gellert A., Vinţan L., *Designing an Advanced Simulator for Unbiased Branches Prediction*, Proceedings of 9th International Symposium on Automatic Control and Computer Science, ISSN 1843-665X, Iasi, 2007.