Understanding and Predicting Indirect Branch Behavior

Adrian FLOREA, Lucian N. VINTAN

“Lucian Blaga” University of Sibiu, Computer Science Department, Str. E. Cioran, No. 4, Sibiu-550025, ROMANIA, Tel./Fax: +40-269-212716, E-mail: adrian.florea@ulbsibiu.ro, lucian.vintan@ulbsibiu.ro, ioan.z.mihu@ulbsibiu.ro

Abstract:
Understanding program behavior is at the foundation of computer architecture and program optimization. Machine code usually has much less semantic information than source code, which makes it much more difficult to discover control flow or data flow information. The main aim of this paper consists in extracting some corpus program and features from procedural and object-oriented applications or execution characteristics of desktop applications that generate indirect function call (unavoidable breaks in control) at machine level. We also made a comparative analysis of C and C++ languages behavior from the execution point of view on instruction level parallelism architectures. The simulation part of the paper mainly refers to measuring the locality degree of target addresses on series of architectural resources but specially to adapting some knowing prediction schemes (Two Context Predictors named Target Cache and, respectively, the complete PPM Predictor) to predict indirect jump instructions. PPM predictor seems to be an almost ultimate limit of target prediction, and, thus, a good frame for deriving practical prediction schemes.

Keywords
Procedural and object-oriented programs, indirect branches / function calls, virtual functions, dynamic branch prediction, instruction level parallel architectures, SPEC benchmarks, dynamic link libraries (DLL).

1. Introduction
As the average instruction issue rate and depth of the pipeline in multiple-instruction-issue (MII) processors increase, accurate dynamic branch prediction becomes more and more essential. Very high prediction accuracy is required because an increasing number of instructions are lost before a branch misprediction can be corrected. As a result even a misprediction rate of a few percent involves a substantial performance loss [Vin02].

If branch prediction is to improve performance, branches must be detected within the dynamic instruction stream, and both the direction taken by each branch and the branch target address must be correctly predicted. Furthermore, all of the above must be completed in time to fetch instructions from the branch target address without interrupting the flow of new instructions to the processor pipeline. A classic Branch Target Cache (BTC) [Hen96] achieves these objectives by holding the following information for previously executed branches: the address of the branch instruction, the branch target address and information on the previous outcomes of the branch. Branches are then predicted by using the Program Counter (PC) address to access the BTC in parallel with the normal instruction fetch process. As a result each branch is predicted while the branch instruction itself is being fetched from the instruction cache. Whenever a branch is detected and predicted as taken, the appropriate branch target is then available at the end of the instruction fetch cycle, and instructions can be fetched from the branch target in the cycle immediately after the branch itself is fetched. Straightforward prediction mechanisms based on the previous history of each branch give a prediction accuracy of around 80 to 95% [Hen96]. This success rate proved adequate for scalar processors, but is generally regarded as inadequate for MII architectures.
The requirement for higher branch prediction accuracy in MII systems and the availability of additional silicon area led to a dramatic breakthrough in the early 90s with branch prediction success rates as high as 97% [Yeh92] being reported. These high success rates were obtained using a new set of prediction techniques known collectively as Two-Level Adaptive Branch Prediction that were developed independently by Professor Yale Patt’s group at the University of Michigan [Yeh91] and by Pan, So and Rahmeh from IBM and the University of Texas [Pan92]. Two-Level Adaptive Branch Prediction uses two levels of branch history information to make a branch prediction. The first level consists of a History Register (HR) that records the outcome of the last $K$ branches encountered. The HR may be a single global register, HRg, that records the outcome of last $K$ branches executed in the dynamic instruction stream or one of multiple local history registers, HRl, that record the last $K$ outcomes of each branch. The second level of the predictor, known as the Pattern History Table (PHT), records the behavior of a branch during previous occurrences of the first level predictor. It consists of an array of two-bit saturating counters, one for each possible entry in the HR. $2^K$ entries are therefore required if a global PHT is provided, or many times this number if a separate HR and therefore PHT is provided for each branch.

Although a single term is usually applied to the new predictors, this is misleading. Since the first level predictor can record either global or local branch history information, two distinct prediction techniques have in fact been developed. The global method exploits correlation between the outcome of a branch and the outcome of neighboring branches that are executed immediately prior to the branch. In contrast, the local method depends on the observation that the outcome of a specific instance of a branch is determined not simply by the past history of the branch, but also by the previous outcomes of the branch when a particular branch history was observed.

We have emphasized earlier that most branch prediction research is based on two closely related correlation mechanisms. Yet branch prediction is a specific example of a far more general time series prediction problem that occurs in many diverse fields of science. It is therefore surprising that there has not been more cross-fertilization of ideas between different application areas. A notable exception is a paper by Mudge et al [Mud96] that demonstrates that all two-level adaptive predictors implement special cases of the Prediction by Partial Matching (PPM) markovian algorithm that is widely used in data compression. Mudge uses the PPM algorithm to compute a theoretical upper bound on the accuracy of branch prediction, while Steven et al [Ste99] demonstrate how a two-level predictor can be extended to implement the PPM algorithm with a resultant reduction in the misprediction rate. Other researchers developed some more sophisticated predictors based on neural networks algorithms [Ega03].

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. Using conventional prediction mechanisms (BTB), the misprediction rates for indirect branches measured on some SPEC benchmarks were about 50% [Cha97]. The authors proposed a new prediction mechanism, named the Target Cache, for predicting the targets of indirect jumps. As the program executes, the target cache records the target address for each indirect jump target encountered. Therefore, the prediction mechanism consists in selecting its prediction from all the targets of the indirect jump that have already been encountered. Just as an example, for Perl and Gcc belonging to SPECint95 benchmarks, the authors reported quite high misprediction rates varying from 30% to 48%. In [Sta98] the authors are developing a prediction scheme for indirect jumps by constructing the index into the prediction table using the last $N$ target addresses, and using profiling information to select the proper value of $N$ for each branch. In this manner, good accurate branch prediction is achieved. In fact, both research groups developed some context predictors that are not derived from the general complete PPM prediction method, because in their schemes there are not encountered frequencies associated with the searched pattern.

The indirect jumps prediction challenge is particularly gained through object-oriented programming techniques that exercise different aspects of computer architecture to support the object-oriented programming style. These languages tend to use indirect function calls, where the
address of the call target is loaded from memory [Cal94]. Therefore, the necessity of understanding and accurate predicting these indirect jumps grow more and more. Accordingly, the main aim of this paper consists in understanding what C and C++ constructions and characteristics generate indirect jumps and calls and, based on these investigations, trying to better predict indirect branches. In Section 2 we develop some analysis of C versus C++ languages from execution viewpoint on Instruction Level Parallel Architectures. In Section 3 we extract some typical corpus of procedural and object-oriented languages that generates, after compilation, indirect jumps and calls. In Section 4 we measure target localities associated with these indirect branches, in order to estimate how predictable they are. In Section 5 we predict these indirect jumps and calls using some contextual value predictors, derived from the complete PPM predictor respectively the Target Cache predictor. Finally, Section 6 provides some concluding remarks and further work.

2. The Analysis of C versus C++ Languages from Execution Viewpoint on Instruction Level Parallelism Architectures

The history of processors marks out two paradigms for improving the performance based on software respectively on hardware. Despite their common goal – exploiting and increasing the Instruction Level Parallelism – the research community is split in two “almost separated” entities for accomplishing it. Whereas the computer architects sewer their efforts for exploiting / optimizing the existing processing techniques through laborious simulations on representatives benchmarks in machine code format, without taking account of source code semantic, the compiler writers’ issue is to optimize the source (or object) code. Sustaining the last idea, Knuth himself, after analyzing the static and dynamic behavior of a large collection of Fortran programs, concluded that programmers had poor intuition about what parts of their programs were most time-consuming, and that execution profiles would significantly help programmers improve the performance of their programs [Cal94]. The idea that processor architecture interacts only accidentally with software domain is completely wrong, between hardware and software existing strong interdependences, unexploited appropriately yet. There are at least two reasons, which prove that the processors and the compilers design processes are made in the same time:

- The simulated benchmarks are compiled for some certain architectures (for example GNUC Compiler from Linux can generate machine code for Intel or SimpleScalar architecture).
- The compiler should generate code for exploiting the architectural characteristics otherwise this code will be inefficient from execution time point of view.

It’s extremely well-known that the computer systems technology develops extremely fast (remember Moore’s low), making necessary the emergence of new benchmarks to reflect the architectural and technological improvements from computer architecture domain, new compilers, languages interpreters, the latest multimedia applications, web servers, compressing / decompressing / transmitting text or audio / video / GSM signals. A majority of the papers published in leading computer architecture conferences use SPEC CPU2000, or its predecessor SPEC CPU95, which has become the de facto standard for measuring processor and memory-hierarchy performance [Cit03]. Taking into account that SPEC CPU95 were entirely procedural and additionally, due to the latest years manifested trends to transit towards object-oriented languages based on advanced programming concepts like encapsulation, inheritance, polymorphism it was expected for the new suite (SPEC2000) to contain more object-oriented benchmarks (C++ programs). Unfortunately, only two C++ applications were submitted to SPEC consortium for voting: whilst one of them worked with gnu g++ but proved impractical to port to ANSI C++, the other one - 252.eon – was far more portable, working well on all 17 C++ compilers tested [Henn00]. Thus, there are only some few reasons for which C++ applications are considered at the moment the biggest challenge both for compilers writers’ community and for microarchitectures designers. More than that, since 1994 the researchers [Cal94] have proved, based on simulation, that
there are significant differences between procedural (C) and object-oriented (C++) programs having impact on performance (processor speed, memory requirement / usage).

The main deficiency of structured programming is the separate treatment applied to data structures and algorithms that process these structures. Also, in procedural programs written in languages like C it is very difficult to reuse programs, scale, maintain, debug and extend some program modules. By contrast, based on fundamental concepts the object-oriented programming goal is to build a class hierarchy to model complex systems, deriving from advantages like economy and code reuse, extendibility. Dividing the applications in many modules, so that one who develops a module is not needed to know the implementing details of the others, has some very useful consequences:

- Decreasing the developing time of applications;
- Simplifying the maintenance of modules;
- Increasing the program quality.

Next, we comparatively present some of the C versus C++ programs execution characteristics.

- Statistically, many object-oriented programs contain a small number of C++ functions, and quite a large percentage of C routines because these programs rely heavily on predefined C libraries.
- C++ functions and methods are much shorter than C functions. More than that, C++ methods contain fewer instructions than C++ functions, consequence of inheritance. Function size is important from two aspects: first, small functions have proportionally greater fixed function call overhead (saving registers, setting up arguments) and so will benefit more from optimizations like inlining; second, function size could have a significant effect on instruction cache performance. Instead of executing large monolithic functions to perform a task, as is often the case in C programs, C++ programs tend to perform many calls to small functions, so that the object-oriented programs benefit less from the spatial locality of larger blocks, and suffer more from function call overhead. Also, C++ programs require instruction caches that are approximately two times larger than C programs to achieve similar miss rates [Cal94].
- There is a small difference in the basic block size between C and C++ programs, with C programs having slightly larger blocks (but under 6 instructions). Larger basic blocks offer more opportunity for architecture-specific optimizations, such as instruction scheduling and, as a consequence it might involve a more aggressive fetch rate.
- Whilst the C programs execute more conditional branches (bnez $t0), object-oriented programs execute more direct or indirect function calls (jal <proc_name> / jal $t1) and returns in the caller routine (jr $ra). These results imply that different branch prediction architectures are needed for C and C++ programs in order to achieve high prediction accuracy for the different languages. Simulation results show that virtual function calls are currently used in object-oriented programs with percentages between 0.1% and 20% (4 to 20 times fewer than direct calls and 30 to 150 times fewer than conditional branches) [Roth98, Flo03].
- To better understand the behavior of programs, it is very helpful to know, statistically, how many of the conditional branches present in a program are actually significant during the program’s execution. So, a relatively small fraction of the conditional branches and indirect jumps present in each program source actually contributes significantly to the dynamic execution of the programs because many branches are only encountered during error conditions, or may reside in unreachable or unused code, or they could appear only in certain context unreachable always (correlated branches $\text{if cond1} \text{ if cond2}$). For both C and C++ the mean of the 95\textsuperscript{th} quantiles requires less than 10% of the total branches present and the 50\textsuperscript{th} quantile requires less than 1% of all branches [Cal94, Flo03]. Also, there are many more “significant” indirect calls in C++ programs (25%) than in C programs (10%). An implication of this result is that for many programs, branch prediction techniques need not
predict all the program branches effectively, but instead can be very effective if only a small fraction of important branches are correctly predicted.

- There are some behavior differences between C and C++ programs due to the compiler used. Most of them are explained by the different runtime systems and standard libraries provided with each compiler. Concluding:
  - The code generated by GNU C++ compiler (g++) is slightly less efficient than that compiled with DEC C++ [Cal94].
  - The G++ program executed 2.5% more instructions and called 8% more functions [Cal94].
  - The G++ program generated significantly fewer conditional branches (almost 28%) although both programs executed roughly the same number of branches [Cal94].

In our research we have used the GNU G++ compiler, 2.6.3 version under Linux Red Hat 7.3 operating system and Intel Pentium III processor.

- Link-time optimization represents an important factor in analyzing the performance of those two kinds of programs (C vs. C++). Object-oriented programs are harder to optimize than programs written in languages like C or Fortran because they use dynamic dispatch: the procedure invoked by the call is not known until run-time since it depends on the dynamic type of the receiver. Therefore, a compiler usually cannot apply standard optimizations such as inline substitution or inter-procedural analysis to these calls.
  - Type feedback – using profile information gathered at run-time reduces the number of indirect function calls with 31% by conditional in-lining of these functions [Cal94b, Aig96]. The optimization is possible only if the execution frequency of a call site is high enough. However, the benefit of substituting conditional branches for indirect function calls is very dependent on the underlying architecture; other disadvantages could be the code expansion and the increasing of the instruction cache miss ratio.
  - Optimizing compilers for object-oriented languages apply static class analysis and other techniques to try to deduce precise information about the possible classes of the receivers of messages; if successful, dynamically-dispatched messages can be replaced with direct procedure calls and potentially further optimized through inline-expansion. By examining the complete inheritance graph of a program (class hierarchy analysis) - identifying the virtual declared methods, inherited but not overridden in the inherited class, the binding being made statically, the compiler can improve the quality of static class information and thereby improve run-time performance [Dean94, Aig96]. In our work we propose an example that can mark out the proper usage of class hierarchy analysis [Flo03] – see figure 2.

- The call stack depth is pretty small for both kinds of programs (9.9 – C vs. 12.1 – C++) being strongly influenced by application domain and the programming techniques (recursion, backtracking, dynamic programming). Thus, it is justified the very high degree of value locality obtained for MIPS processors SP (stack pointer) register (over 80% when the history depth is 4 or 8 values respectively over 90% while extending the checking of instruction results in last 16 or 32 values [Flo02]).

- A significant contribution to the high rate of memory operations in C++ is probably due to register saves and restores across function calls. Due to the object-oriented programs tendency to create reusable components and also to the relative small size of allocated objects there is much more heap allocation. Another important difference between C and C++ is the support the languages provide for heap-allocation. Whereas in C, there is a simple library interface, C++ supports allocation and de-allocation with both syntactic and semantic conveniences such as constructors, destructors, new, and delete.

Although, the SPEC benchmarks represents the proper environment for testing and simulation of the novel ideas from computer architecture, the most computing systems have as platform the Microsoft Windows (9x, NT, 2000, XP) with Intel x86 processors. Many of these execute personal applications (multimedia, text editors, graphical and interactively presentations, image processing,
email, Internet browsers) and not based on scientifically research, industrial or military purpose. Results gathered from laborious simulations have shown that beside some similar characteristics of desktop applications running on Windows NT operating systems and Intel x86 processors, respectively of integer SPEC95 benchmarks, there are differences in using the hardware resources and in microarchitectural level programs behavior (indirect and conditional jumps, usage of library functions) [Lee98].

- A significant difference between desktop application and integer SPEC95 benchmarks consists in the very large binary (object code) size of the formers (from 3 to 10 times bigger). This is to be expected, as the desktop applications are feature rich and most of them support many functions in addition to their main task (drawing graphs in PowerPoint). Also, program execution tends to be dispersed across more functions compared to the SPEC95 applications (at least one order of magnitude more unique functions) increasing the probability of conflict and capacity misses in the instruction cache.

- From branch prediction performance point of view, simulation results on five well-known desktop applications (acrord32, netscape, photoshp, powerpnt, winword) showed that - despite the interactive nature of those application – these applications don’t translate into unpredictability at the microarchitecture level; an explanation could be that user interaction happens at a course level relative to the decisions that are made in the microarchitectures – the user tells a program “what to do”, rather “how to do it” [Lee98].

- Another major difference is that desktop applications make extensive use of dynamic linked libraries (DLLs), many of these (graphics library, access to operating system services, networking, C runtime library, user interface functions) shared with other applications, while the SPEC95 applications do not. There are some reasons for that DLL calls are considered more expensive than calling statically linked functions:

  - Unlike regular function calls, DLL calls are implemented as indirect function calls. Simulation results on the above five desktop applications exhibit a higher proportion of indirect calls: averaging 25% (pure indirect calls + DLL) versus almost null (under 4%) for SPEC95, as we will show further [Lee98, Flo03]. Taking into account Calder’s remark saying that C++ programs tend to have more indirect calls because C++ programs tend to use dynamic dispatch [Cal94], the readers could be tempted to believe that these five desktop applications are entirely object-oriented programs. Actually, just netscape, photoshp, powerpnt are written in C++ [Lee98].

  - DLLs are shared between applications, so improving instruction locality through traditional reordering algorithms, statically or dynamically is more difficult. Frequent crossing of DLL boundaries will require pages from both DLLs to be resident in the address space even though only a small portion of these pages are actually used by the application otherwise arising an internal fragmentation.

  - Other metrics like data cache and TLB buffer, function and basic block sizes, percentage of different types of instructions in programs, are quite comparable in both cases of applications (desktop and SPEC).

3. Investigations about Indirect Jump Generating through Procedural and Object-Oriented Applications

At compile time the code involving calls to library routines, to procedures defined in separately compiled modules, and to dynamically dispatched “virtual functions” in object oriented languages (in the case where the virtual function is never overridden), cannot be effectively optimized. Machine code usually has much less semantic information than HLL source code, which makes it much more difficult to discover control flow or data flow information. Control flow analysis of executable files can be difficult because determining the extent of jump tables, and hence the
possible targets of the code derived from switch/case statements, can be difficult. In this work, we have analyzed SPEC’95 benchmarks (entirely procedural) as well as some own test programs, with small number of dynamic instructions (under 5 millions), two of them being object-oriented. Based on simulation results there are detached some obvious conclusions:

- Indirect jumps occur more frequent in object-oriented programs rather than in procedural programs.
- Late (dynamic) binding realized through polymorphism (which in his turn is based on inheritance) generates indirect function calls in object-oriented applications (C++, Java, Smalltalk). These languages promote a polymorphic programming style in which late binding of subroutine invocations is the main instrument for modular code design. Virtual function tables, the implementation chosen for most C++ and Java compilers, execute an indirect branch for every polymorphic call. More than that, in Java, instance methods are virtually declared by default. If they aren’t explicitly declared final they can be overridden in subclasses.
- The presence of indirect jumps in procedural (C) programs is mainly due to the following two aspects:
  - Indirect calls through function pointers (address).
  - The possible targets of the code derived from switch/case statements (with more than 5 options - using GCC compiler for SimpleScalar architecture).
  - One of the reasons that favor the indirect jumps existing in programs is the presence of statically or dynamically library function calls (e.g. the qsort function from BorlandC Help – see the file ...\BorlandC\Crt\Clib\qsort.cas [Flo03] and DLLs usage from desktop applications [Lee98]).

Late binding appears in the situation in which the compiler and link-editor make the correspondence between a method’s name and its address from code segment through a table of possible jump targets. Choosing the effective call address is made only during run-time, depending on the object for which it is called (too late from compiler’s point of view). For every class the compiler builds a virtual methods table – VMT and includes in every class instance (object) a pointer (VMT_ptr) towards the VMT table of each appropriate class (see figure 1). VMT is a record with a slot for every virtual method (private, protected, and public) of the class. Each slot holds the address of the code segment of the appropriate method body. In this way, if a message is sent to an object, it will be made indirect through the virtual method table to find the appropriate code to be executed. When a subclass is defined, a new VMT is constructed by starting with a copy of the old VMT, adding new virtual methods to the end, and replacing code pointers of overridden methods.

![VMT tables belonging to application mostenire_simpla1.cpp](image)
In the next picture (figure 2), we present comparatively two screenshots of the same programs (in the left side – C source code and in the right side – assembler MIPS code). The application mostenire_simpla1.cpp (s – after compiling) [Flo03] solves through recurrent backtracking two well-known problems. The first one generates permutation of \( n \) elements and the second tries to put \( n \) queens on the chess table without attack each other. It is implemented a class back having as member properties the solutions stack (int s[10]) and the size of stack (int n). Among the member methods compl (the main routine for executing the backtracking), show (the method for printing the solutions), valid (for validating the partial solution) and final (for testing if the top of the stack is reached) only the last three are declared virtual. From the base class it is derived the class dame that inherits members (properties and methods) of back class and override the two methods (valid and show) with code of the owner application.

Figure 2. Dynamically dispatched of “virtual functions” in object oriented languages

The statistical results, dynamically generated after test programs execution on SimpleScalar architecture show 14 statically indirect branches (occurred after link-edit stage and execution for \( n=4 \), three of them due to virtual function call (valid, final and show) existing after compiling stage, the rest (11) due to the library function call. Although function final is virtually declared in the base class (back), it has just inherited and not overridden in subclass dame. This is reflected in dames’ virtual table methods (see figure 1), allowing to an optimized compiler that includes class hierarchy analysis to replace the indirect branches generated by this method (final) with a direct procedure call. The three indirect function calls (valid, final, show) occur just in compl method from back class, because this is inherited and early bind (non-virtual) in derived class (dame). If compl is virtual declared in base class and overridden in subclass dame then the three component methods (valid, final, show) would generate a greater (double) number of statically indirect branches depending upon the number of objects (or pointers) from these two classes. Eliminating the virtual declaration of function final will determine, after compiling, changing the indirect function call (jal $31, $2) in a direct procedure call (jal final__4backi). After compiling, there were generated only two indirect branches from a total of 13 statically indirect branches obtained after link-edit stage. The explanation consists in link-edit of some used library functions.
Another graphic application, simple and object oriented is based on inheritance, polymorphism, type casting between derived classes, dynamical allocation memory for objects, constructor and destructor usage in inheritance context.

Since a subclass could be seen as an extension of C++ base class, pointers to base class can keep the addresses of the derived class instances. Also, destroying a subclass instance, first it executes the subclass destructor and after that the base class destructor. The program called *mostenire_simpla3.cpp[.s]* applies a uniform treatment for an array of objects, dynamically allocated, through polymorphism (see figure 3). The base class contains an explicit simple constructor, a method and a destructor, the last two virtually declared. The derived class, besides an additional property member, overrides the two virtual methods. The main application function supposes the existence of a table with 6 dynamic objects some of them belonging to base class and the rest to derived class. The array size may be parameterized, directly influencing the number of dynamic indirect function calls (72 dynamic indirect branches in this case). There are two repetitive instructions (*for* - in C++, see below). First of them calls the method *met* of the corresponding object. The second one is used for destroying the objects array. After the compilation phase, two indirect branches result (seen in *mostenire_simpla3.s*) and 11 indirect branches (after link-edit phase and execution on Simple Scalar architecture – probably because of the library function calls). First indirect branch *jal $31, $2* is due to virtual method *met*. The indirect function call is justified since it doesn’t know exactly in the compilation phase which method will be executed from base class or from the derived one (it doesn’t know exactly to what object the method *met* (*table[i] ->met*) belongs to). The second indirect function call (*jal $31, $2*) is due to the virtual destructor and to the same reason that the object that made the call is unknown.

```c
class Base{
    public:
    ...
    virtual ~Base() {...} // destructor.
    virtual met() {...} // method.
};

class Derived public Base {...
    public:
    ...
    ~Derived() {...}
    met() {...} // destructor.
};

void main(){
    Base *table[6];
    addu $2, $3, 16 // virtual method is at 16 offset "met()".
    addu $2, $3, 8 // virtual destructor is at 8 offset "delete table[k];"
    ...
    lw $2, 0($3) // register $2 will contain the object base address - O_b
    lw $3, 4($2) // register $3 will contain the VMT table base address - V_m
    ...
    move $4, $3 // parameter for function
    ...
    for(int k=0; k<6; k++)
        ...
        lw $4, 0($3) // register $4 will specify the object destructor which is used by
        ...
        move $4, $3 // parameter for function
        ...
        for(k=0; k<6; k++)
            ...
            ...
            ...
} // function end
```

**Figure 3.** Through dynamic bind that incorporate, the polymorphism allows an uniform treatment of heterogeneous massive.

As we can see in figure 3, it could successfully apply virtual function call target prediction via dependence-based pre-computation [Roth99]. Using information fetched from run-time execution of the program, the mechanism for pre-computing the indirect branch targets is based on instruction that computes them rather than on previous targets (see the Target Cache case [Cha97]). Pre-
computation is a general technique that can be applied to any type of instruction, which dynamically identifies the sequence of operations that computes a virtual call target. When the first instruction in such a sequence is encountered, a small execution engine speculatively and aggressively pre-executes the rest. The pre-computed target is stored and subsequently used when a prediction needs to be made. Pre-computation has a shorter learning period and works even in the absence of statistical correlation. The polymorphism is translated in accessing the Virtual Method Table using a three dependent load instructions sequence followed by an indirect function call. The virtual function call (v-call named by [Roth99]) mechanism allows the programmer to treat an array of objects of mixed types Base and Derived uniformly and use a v-call to invoke the correct function implementations. A Virtual Method Table contains the addresses (at predetermined offsets) of all the methods accessible by a given object class. Every object is initialized with a pointer to the VMT corresponding to its class and accesses its functions via this pointer as shown in figure 3 (right side). The initial load accesses the base of the Object (Omet, Od), the second uses the object’s base address to access its Virtual Method Table (Vmet, VD), and the third retrieves the Function address (Fmet, Fd), which is used by the indirect Call (Cmet, Cd). There is also one disadvantage of this technique: pre-computations are not always available due to either insufficient time or an inability to correctly predict the next object’s address.

The following examples will exhibit some procedural program characteristics that imply indirect branches generation at machine code level. First application (sort.cpp – see figure 4) illustrates an indirect call through function pointers (address). The application sort.cpp [Flo03] randomly sorts an integer array, based on a counter variable generated by a function (GenMetoda()), using one from 5 classical sorting methods implemented Bubblesort, Quicksort, Selectionsort and Insertionsort (iterative and binary). The sorting method is chosen by a switch/case statement. This construct will be analyzed later. The result after compilation phase consists of 5 static indirect branches and, respectively 14 indirect branches (after link-edit phase and execution on Simple Scalar architecture – probably because of the library function calls usage – rand, scanf, printf, etc). From a dynamical point of view the results show 1035 indirect branches.

```c
int GenMetoda(){
    return rand() % 5;
}

int AflisareMetoda(int (*p)()){  
    int x=p();
    ...
}

void main(){  
    ...
    for (k=0; k<nr_sort; k++)
    {
        ...
        AflisareMetoda(GenMetoda);
        ...
    }
}
```

Figure 4. Example of indirect call through function pointer in a procedural application (sort.[cpp;s]

Since at this moment, the real-time and embedded systems know a very large dispersion and the embedded software often runs on processors with limited computation power, optimizing the code becomes a necessity. Some of these optimization techniques refer to switch/case statements. In this article we will show how the same switch/case statement behaves differently – on the same
platform (operating system and processor), from generated machine code point of view, depending on the number of possible options (number of sub-cases that must be treated). If the switch/case instruction offers more than 5 possible branches (sub-cases) – our tested programs do not contain a default case - in the resulting assembler code will occur an indirect jump; otherwise, this indirect jump will be replaced by a sequence of conditional branches. In our tests (see figure 5), sources of programs were compiled with gnuC++ compiler 2.6.3 with \(-O3\) optimization flag, on a platform having an Intel 80x86 processor and Linux RedHat 7.3 as operating system. Using the \(-O3\) optimization flag favors nearly all supported optimizations that do not involve performing a space-speed tradeoff. As compared to \(-O\), this option increases both compilation time and performance of the generated code turning on the \(--finline-functions\) and \(--fdelayed_branch\) options.

<table>
<thead>
<tr>
<th>switch 04.cpp</th>
<th>switch 05.cpp</th>
</tr>
</thead>
<tbody>
<tr>
<td>#include &lt;stdio.h&gt;</td>
<td>#include &lt;stdio.h&gt;</td>
</tr>
<tr>
<td>void main()</td>
<td>void main()</td>
</tr>
<tr>
<td>*n = random();</td>
<td>*n = random();</td>
</tr>
<tr>
<td>switch (*n)</td>
<td>switch (*n)</td>
</tr>
<tr>
<td>*n = random();</td>
<td>*n = random();</td>
</tr>
<tr>
<td>switch (*n)</td>
<td>switch (*n)</td>
</tr>
<tr>
<td>case 0:</td>
<td>case 0:</td>
</tr>
<tr>
<td>case 1:</td>
<td>case 1:</td>
</tr>
<tr>
<td>case 2:</td>
<td>case 2:</td>
</tr>
<tr>
<td>case 3:</td>
<td>case 3:</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>#include &lt;stdio.h&gt;</td>
<td>#include &lt;stdio.h&gt;</td>
</tr>
<tr>
<td>void main()</td>
<td>void main()</td>
</tr>
<tr>
<td>*n = random();</td>
<td>*n = random();</td>
</tr>
<tr>
<td>switch (*n)</td>
<td>switch (*n)</td>
</tr>
<tr>
<td>*n = random();</td>
<td>*n = random();</td>
</tr>
<tr>
<td>switch (*n)</td>
<td>switch (*n)</td>
</tr>
<tr>
<td>case 0:</td>
<td>case 0:</td>
</tr>
<tr>
<td>case 1:</td>
<td>case 1:</td>
</tr>
<tr>
<td>case 2:</td>
<td>case 2:</td>
</tr>
<tr>
<td>case 3:</td>
<td>case 3:</td>
</tr>
<tr>
<td>case 4:</td>
<td>case 4:</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

\[
\text{Figura 5. The different behavior of switch/case statement from indirect branches generation point of view (a \(- n 4\) subcases; \(b \(- n 5\) subcases)}
\]

The translation mechanism is realized in assembly language (MIPS) adding at least two instructions at every additional sub-case: an immediate load $l<3,0x3$ (see figure 5) respectively a conditional branch (for comparing if the chosen sub-case is that specified by the \$3 register - beq \$2, \$3, \$L31 - see figure 5). It is easy to understand that, using a great number of options, the number of instructions to be executed will grow very much. The problem could be solved writing much more generically the code sequence, replacing the conditional branches with an indirect jump and an additional sub-cases address table stored in the application data segment. The threshold of 5 sub-cases for switch/case instruction has been chosen because until 4 sub-cases, the number of used
instructions to translate the construction in assembly code were less or equal to the number from
generic sequence (with indirect jump) – see figure 5. The price for generic character of high-level
application is paid at low level because the indirect branches are very difficult to predict and more
than that, a single static instance could determine 6000 dynamic indirect branches (see the case of
go.ss benchmark from SPEC’95 suite [Flo03]).

Looking from the perspective of switch/case statement having at most 4 options, where the
compiler generates a cascade of if-else-if instructions for comparing the selection parameter with
every case address, it is justified trying to place the most frequently executed case on the first
position in the option list. In this situation, the number of comparisons decreases automatically.
Typically, this means that cases corresponding to the success of an operation should be placed
before cases of failure handling. In the SPEC’95 benchmark li.ss, a very frequently called function,
livecar, contains a switch statement where one of the case labels, corresponding to the type LIST,
occurs over 80% of the time. Knowledge of this fact allows the code to be restructured so that this
common case can be tested separately first, and so does not have to go through the jump table,
which is relatively expensive. As these examples suggest, if we know that certain values occur very
frequently at certain program points, we may be able to take advantage of this information to
improve the performance of the program. This information is generated by a value profile, which is
a (partial) probability distribution on the values taken on by a variable when control reaches the
program point under consideration at runtime [Watt01].

Another method to reduce the number of comparisons performed judiciously breaks big
switch statements into nested switches. It puts frequently occurring case labels into one switch and
keeps the rest of case labels into another switch, which is the default leg of the first switch.

The next program called qsort[c,.s] represents an additional example supporting the
affirmation that library function (printf, qsort) determines indirect branches. The assembly code
does not present any indirect calls through function pointers (the only calls that could be seen in
figure 6 are direct: jal qsort, jal strcmp). Analyzing the object code (after linking the source code
with library function used) resulted 11 static indirect branches that generate 61 dynamic indirect
jumps. The C program, offered by BorlandC 3.1 environment as a sample application, sorts a string
array through library function qsort that receive as parameters (the array address, number of
elements, size of every element and the name of a function that compare two strings). A source for
one of those 11 static indirect branches consists in the indirect call through comparing function
pointer, step realized within qsort-precompiled function. Further, we described the Qsort function
characteristics: parameters, usage, call as it occurs in ...\BorlandC\Crtl\Clib\qsort.cas.

<table>
<thead>
<tr>
<th>Name</th>
<th>qsort – sorts using the quick sort routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage</td>
<td>void qsort(void *base, int nelem, int width, int (*fcmp)());</td>
</tr>
<tr>
<td>Prototype</td>
<td>stdlib.h</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qsort is an implementation of the &quot;median of three&quot; variant of the quicksort algorithm. Qsort sorts the entries in a table into order by repeatedly calling the user-defined comparison function pointed to by fcmp. base points to the base (0-th element) of the table to be sorted. nelem is the number of entries in the table. width is the size of each entry in the table, in bytes. *fcmp, the comparison function, accepts two arguments, elem1 and elem2, each a pointer to an entry in the table. The comparison function compares each of the pointed-to items (*elem1 and *elem2), and returns an integer based on the result of the comparison.</td>
</tr>
<tr>
<td>If the items</td>
</tr>
<tr>
<td>• *elem1 &lt; *elem2</td>
</tr>
<tr>
<td>• *elem1 == *elem2</td>
</tr>
<tr>
<td>• *elem1 &gt; *elem2</td>
</tr>
</tbody>
</table>

In the comparison, the less than symbol (<) means that the left element should appear
before the right element in the final, sorted sequence. Similarly, the greater than (>) symbol means that the left element should appear after the right element in the final,
Our research goes further and we propose a trivial procedural test example which contains just the main function and does nothing (main(){ } – see void.cpp [Flo03]). At first sight, after compiling and simulating the object code on SimpleScalar architecture, the obtained results are very surprising. The assembly code saves in stack segment the frame pointer and return address (the context from the application occur – similar to every application [Flo03]) and makes a direct call (jal __main) to an operating system routine. Then the return address and the frame pointer are restored from stack, following a return instruction (j $31) to come back into the operating system. Apparently, there are no indirect branches. However, after simulating the object code 6 static indirect branches occur that generate 8 dynamic indirect branches.

4. Indirect Jumps Target Locality

In order to have an ultimate predictability metric of indirect branches we measured their target address value locality. The value locality concept was first introduced by Lipasti et al [Lip96] and it represents the likelihood of the recurrence of a previously seen value within a storage location. Accordingly, in our case, we’ll say that an indirect jump (call) target value is local if it belongs to the previous K dynamic target instances of that certain jump (call). Obviously, a great target locality degree involves great prediction accuracy, too. In other words, the value locality degree obtained for K dynamic target instances represents the maximum achievable prediction accuracy using a context predictor of order K. Therefore, our approach establishes an analogy between value prediction, and, respectively, indirect jumps target prediction. Using SimpleScalar tool set we developed some simulators in order to measure target locality, and, respectively to predict targets for indirect jumps and calls. We have used some of the SPECint95 benchmarks and simulated 5,000,000 dynamic instructions for each one. Also, as we detailed in the previous paragraph, we developed some C and C++ test programs in order to better understand how indirect branches are generated through the compilation process (unfortunately, SPECint95 doesn’t contain object-
oriented benchmarks; SPECint2000 contains only one!). Table 2 presents some static and dynamic characteristics of these developed C/C++ test programs. The test programs were described in details in the previous paragraph. Interesting, in Back_ test program, 22 indirect static branches involve over 145,000 indirect dynamic branches representing about 3.13% of the total number of dynamic branches.

<table>
<thead>
<tr>
<th></th>
<th>Static Indirect Branches</th>
<th>Total Static Branches</th>
<th>Dynamic Indirect Branches</th>
<th>Total Dynamic Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mostenire_simpla1</td>
<td>14</td>
<td>585</td>
<td>512</td>
<td>22,226</td>
</tr>
<tr>
<td>Back_</td>
<td>22</td>
<td>639</td>
<td>145,236</td>
<td>4,636,747</td>
</tr>
<tr>
<td>Hanoi</td>
<td>12</td>
<td>591</td>
<td>1,291</td>
<td>211,390</td>
</tr>
<tr>
<td>Mostenire_simpla3</td>
<td>11</td>
<td>475</td>
<td>72</td>
<td>9,307</td>
</tr>
<tr>
<td>Qsort</td>
<td>11</td>
<td>376</td>
<td>61</td>
<td>3,966</td>
</tr>
<tr>
<td>Sort</td>
<td>14</td>
<td>632</td>
<td>1,035</td>
<td>1,272,867</td>
</tr>
</tbody>
</table>

Table 2. Counting the number of static / dynamic indirect jumps and calls from total number of static / dynamic branch instructions executed belonging to our test programs

In this paragraph we mainly present the obtained target locality degrees. In the next paragraph we present some prediction accuracies obtained using some contextual predictors dedicated for indirect branches. First, we measured the percentage of dynamic pure indirect branches, calls and returns, belonging to some SPEC benchmarks, and, respectively, to our special developed test programs (see table 3 and figures 7 and 8). Even if the percentage of indirect dynamic branches is up to 4.01% in the considered SPEC benchmarks, their processing action is quite important. According to our experience in developing the previously described object-oriented toy benchmarks, we are expecting that object-oriented programs considerably increase the percentage of indirect branches, due to their intrinsic programming characteristics. Figures 9 and 10 present the obtained target localities for some SPEC benchmarks respectively for our toy test programs, considering different context orders, varying from 1 to 32. In both cases, for a context order greater than 1, the corresponding average value locality is over 90% that is remarkable, suggesting us that indirect branch targets are very predictable. Also important, the optimal target locality is obtained considering the last $K$ instances of a certain indirect branch. These results suggest us that the optimal context predictor should be obtained for $K=4$, too (see the next paragraph). Surprisingly good results for Back_ test program could be justified by the small number of declared individual objects and the corresponding recurrently called methods (see figure 10).

<table>
<thead>
<tr>
<th></th>
<th>Static Indirect Branches</th>
<th>Total Static Branches</th>
<th>Dynamic Indirect Branches</th>
<th>Total Dynamic Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>applu</td>
<td>30</td>
<td>782</td>
<td>986</td>
<td>93529</td>
</tr>
<tr>
<td>apsi</td>
<td>52</td>
<td>2127</td>
<td>29877</td>
<td>745243</td>
</tr>
<tr>
<td>cc1</td>
<td>37</td>
<td>4294</td>
<td>27294</td>
<td>1037705</td>
</tr>
<tr>
<td>fpppp</td>
<td>36</td>
<td>1103</td>
<td>2205</td>
<td>133251</td>
</tr>
<tr>
<td>go</td>
<td>1</td>
<td>289</td>
<td>6272</td>
<td>1004731</td>
</tr>
<tr>
<td>li</td>
<td>16</td>
<td>1293</td>
<td>34591</td>
<td>1168222</td>
</tr>
</tbody>
</table>

Table 3. Counting the number of static / dynamic indirect jumps and calls from 5,000,000 dynamic instructions executed on suite of SPEC’95 benchmarks
Percent of branch instructions difficult to predict
(direct call, return, "pure" indirect jumps)

SPEC’95 benchmarks that present indirect jumps and calls

Figure 7. Statistic about the percentage of difficult to be predicted dynamic branch instructions

Percent of static branch instructions (direct call, return, "pure" indirect jumps and calls)

Our test programs

Figure 8. Statistic about the percentage of difficult to be predicted static branch instructions
5. Predicting Indirect Jumps

We have developed two distinct types of context predictors dedicated for indirect jumps. The first one represents a set of $K$ Markov predictors, named complete PPM predictor [Mud96]. The prediction is based here on the PPM (Prediction by Partial Matching) algorithm that represents a universal compression/prediction algorithm. PPM has been theoretically proven optimal in data compression and prefetching and also in some speech recognition problems. The bases of the complete PPM algorithm of order $K$ are a set of $K$ Markov predictors. A Markov predictor of order $J$, $J \in [1, K]$, predicts the next targets (values) based upon the $J$ immediately preceding target patterns (a simple Markov chain of order $J$). More precisely, the prediction process counts every time when that searched pattern on $J$ targets was found in the last $I$ stored targets, $I \gg J$. The prediction is according to the most frequent target (certain value) that follows the searched founded pattern in the stored string of $I$ values. The prediction algorithm for a Markov predictor of order $K$ is illustrated in figure 11. PPM uses the $K$ immediately preceding targets to search a pattern in the highest order.
Markov model, in this case $K$. If the search succeeds, which means the pattern appears in the input sequence seen so far ($I$ values), PPM predicts the next value using this $K$-th order Markov predictor. However, if the pattern is not found, PPM uses the $(K-1)$ immediately preceding targets to search the next lower order $(K-1)$-th order Markov predictor. Whenever a search misses, PPM reduces the pattern by one value and uses it to search in the next lower order Markov predictor. This process continues until a match is found and the corresponding prediction can be made. If even a 1 order Markov predictor doesn’t generate any target prediction (search miss), no prediction is made.

![Figure 11. Markovian predictor of order K](image-url)
A $K$ order Markov context predictor algorithm, as it was implemented in our developed simulator, is presented in figure 11. If the $K$ values search pattern is not found in the stored $I$ values string an $(K-1)$ order Markov predictor will be used. Figure 12 represents the classification automata associated with the $K$ order Markov context predictor that gives the prediction confidence. There are $K$ confidence automata, associated with each Markov predictor. A prediction is generated only in the two predictable states. For each indirect branch, the Jump Value Prediction Table structure (JVPT) has an entry containing the last $I$ target values. After the indirect branch is resolved, if it produces a new target value, this value is stored in the corresponding JVPT’s line, using a FIFO (First in First Out) replacement algorithm. Obviously, JVPT structure has different associative degrees, from direct mapped to full associative, established as an input parameter into the developed simulator.

The second considered predictor was a tagged Target Cache predictor inspired from that presented in [Cha97]. The Target Cache (TC) 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 fetching an indirect jump, the TC is accessed with the PC to produce the predicted target address. As the program executes, the TC records the target for each indirect jump target encountered. Our simplified proposed scheme uses the least significant bits of PC for set selection. The most significant bits of the instruction address (PC) form the tag. In the case of hit in TC (the Tag emitted by PC is equal to one Tag from the respective set) we propose as new target the corresponding address (field $Adr$ from figure 13). In our implementation in a certain set we have only different Tags determined by different PCs (indirect branch instructions), and every location retains the last target of every jump. In 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. It is implemented a FIFO (First In First Out) replacement algorithm. We have implemented and simulated a $P$-way set associative TC, where $P=1,2,4$, like that presented in figure 13. In the case of miss in TC (the tag emitted by PC doesn’t appear in none location of the set) the prediction is considered wrong, it doesn’t propose any value and it is added a new entry in the respective set updating with the proper tag and the proper target accordingly with the specified replacement algorithm.

Figure 12. Classification automata of a certain entry in JVPT table
Figures 14 and 15 present the obtained prediction accuracies for a PPM predictor implemented with a complete associative JVPT structure having 256 entries. Each entry in the JVPT stores 32 respectively 256 targets values of the corresponding indirect branch. The search pattern length is varied from 1 to 4 (figure 14) respectively from 1 to 12 (figure 15). Analyzing the harmonic means obtained results we observed an optimal prediction accuracy of about 88.60% (for a search pattern length of 3 values) respectively of about 90.09% (for a search pattern length of 4 values). These results are better than that reported by other researchers that used more simplified context predictors [Cha97, Dri98].
Ap=f(pattern) - using the PPM complete predictor

**Figure 15.** Indirect branch prediction accuracy using a complete PPM, varying the pattern size

The next two figures are analogous with the previous two, obtained in the same conditions; the specific difference consists that they present the accuracies for simulating our own developed test programs. The poor results obtained for programs named Mostenire_simpla3 and Qsort is probably due to the utilization of massive heterogeneous objects and other data structures. As we expected, the complete PPM predictor exploits very well the recursion used in Back_ and Hanoi test programs. The optimal search pattern length is 4, in perfect concordance with the previous results obtained on SPEC, and, also with the results obtained in Section 4.

**Figure 16.** Indirect branch prediction accuracy using a complete PPM predictor, varying the pattern size, having an associative JVPT table, checking our own test programs
Ap=f(pattern) using PPM complete predictor
History=256, JVPT=256 entries

97,87
63,93
55,56
99,97
76,70
97,83
78,02
77,99
78,01
78,01

Figure 17. Indirect branch prediction accuracy using a complete PPM predictor, varying the pattern size, having an associative JVPT table, checking our own test programs.

Table 4 presents the percentages of correct classified indirect branches using the automata showed in figure 12 (how many times the PPM complete predictor guessed the correct value divided to the number of times when the automata decided that the value was predictable and respectively how many times the PPM mispredicted the value divided to the number of times when the automata decided that the value was unpredictable). As it can be observed, the predictable indirect branches are accurately classified but the unpredictable branches are very poorly classified. In this second case the confidence automata is too conservative. Fortunately, fewer than 4% of indirect branches are really unpredictable, but the misclassification problem still exists. We’ll try to find other automata in order to have a better classification, especially for unpredictable indirect branches, and, therefore, hopefully a better prediction accuracy.

<table>
<thead>
<tr>
<th>SPEC’95 benches</th>
<th>pattern=1 Pred</th>
<th>Unpred</th>
<th>pattern=2 Pred</th>
<th>Unpred</th>
<th>pattern=3 Pred</th>
<th>Unpred</th>
<th>pattern=4 Pred</th>
<th>Unpred</th>
</tr>
</thead>
<tbody>
<tr>
<td>applu</td>
<td>99,67%</td>
<td>8,33%</td>
<td>99,56%</td>
<td>6,25%</td>
<td>99,67%</td>
<td>4,35%</td>
<td>99,67%</td>
<td>4,35%</td>
</tr>
<tr>
<td>apsi</td>
<td>93,72%</td>
<td>42,72%</td>
<td>95,77%</td>
<td>36,92%</td>
<td>96,40%</td>
<td>41,24%</td>
<td>97,11%</td>
<td>30,20%</td>
</tr>
<tr>
<td>cc1</td>
<td>76,71%</td>
<td>52,01%</td>
<td>86,92%</td>
<td>57,55%</td>
<td>92,54%</td>
<td>63,45%</td>
<td>94,53%</td>
<td>66,53%</td>
</tr>
<tr>
<td>fpppp</td>
<td>98,86%</td>
<td>19,64%</td>
<td>98,86%</td>
<td>19,64%</td>
<td>98,86%</td>
<td>19,64%</td>
<td>98,86%</td>
<td>19,64%</td>
</tr>
<tr>
<td>go</td>
<td>92,85%</td>
<td>0,00%</td>
<td>92,84%</td>
<td>0,00%</td>
<td>92,85%</td>
<td>0,00%</td>
<td>92,82%</td>
<td>0,00%</td>
</tr>
<tr>
<td>li</td>
<td>80,62%</td>
<td>19,47%</td>
<td>83,49%</td>
<td>27,89%</td>
<td>84,69%</td>
<td>29,18%</td>
<td>85,53%</td>
<td>30,04%</td>
</tr>
<tr>
<td>Average</td>
<td>90,40%</td>
<td>23,70%</td>
<td>92,91%</td>
<td>24,71%</td>
<td>94,17%</td>
<td>26,31%</td>
<td>94,75%</td>
<td>25,13%</td>
</tr>
</tbody>
</table>

Table 4. Percentages show fractions of unpredictable and predictable indirect branches identified as such by the classification automata afferent to PPM complete predictor using an associative JVPT table.

The next two figures (figure 18 and 19) present how two TC having 64 respectively 256 $P$-way associative sets, where $P=1,2,4$ work. For a 256 set TC the prediction accuracy is practically saturated. Anyway, the obtained results are smaller than those obtained using a complete PPM predictor. According to the target locality results obtained in Section 4, a 4-way set associative TC seems to be the best solution.
6. Conclusions and Further Work

This paper tries to investigate the two hemispheres, hardware and software, just apparently separated (disjointed) in which the computer science researchers develop their activity in order to better understand indirect branch behavior and prediction processes. The programmers’ tasks should not be limited only to the application interface that attract or to the several different tricks that turn the user into a simple robot but must take care of the application implications on the microarchitecture.

Through this paper we developed some analysis of C versus C++ languages from execution viewpoint on Instruction Level Parallel Architectures. Also we extracted some typical corpus of procedural and object-oriented languages that generate, after compilation, indirect jumps and calls. Polymorphism, indirect function calls and some case/switch structures are the main sources responsible for generating these indirect branches. Next, we measured target localities associated with these indirect branches, in order to estimate how predictable they are. As we expected, the
conclusion was very optimistic. Accordingly, we predicted these indirect jumps and calls using some contextual value predictors, derived from the complete PPM predictor respectively the Target Cache predictor. The obtained results were better than that reported by other researchers that used more simplified context predictors. PPM predictor seems to be an almost ultimate limit of context target prediction, and, thus, a good frame for further deriving new practical prediction schemes. In this sense, we’ll try to extend the prediction information adding new correlations and finding some efficient hash functions.

The application goal should be proper usage both of software resources (libraries, interfaces components, etc.) and the algorithms, and the well-known programming concepts (virtual function declaration, function call through pointers) even when it is not the case. In contrast “the sickness” (to read indirect jumps, huge object code, additionally hardware resources) will reflect on the architecture performance. As far as the architecture designers are concerned, their proposed schemes could be more efficient if not only the object code from benchmarks (“wear off by any semantic information”) is analyzed but they will also look “higher” towards high level sources of simulated programs.

References


[Lip96] Lipasti, M. H., Wilkerson, C. B., Shen J. P. Value Locality and Load Value Prediction,


