Adding 2D-Profiling Support in ORC

Linda Bigelow
bigelow@ece.utexas.edu

Aater Suleman
suleman@ece.utexas.edu

1 Introduction

Profiling is commonly used by static compilers to predict run-time program behavior, enabling more compiler optimizations. These profile-guided optimizations, however, are only beneficial if the profile-time behavior of the program accurately reflects the run-time behavior of the program. Capturing a wide range of program behaviors often requires profiling with multiple input sets, which is expensive in terms of resources.

One example of an input-dependent program characteristic is branch predictability. Prior research [3] has shown that, although some branches behave similarly across different input sets, others do not. It has also been observed that branches that are very easy to predict with one input set may behave very differently with a different input set [8]. Since the optimization decisions made by the compiler (for example, whether to predicate a branch or not) can significantly affect overall performance, it would be beneficial for the compiler to be able to efficiently identify input-dependent branches.

2D-Profiling [8] is an effective profiling mechanism for identifying input-dependent branches without using multiple input sets. Unlike other branch profiling techniques, 2D-Profiling uses the time-varying phase behavior of a branch to predict whether the branch is input-dependent or not. This is accomplished by using a single input set and realizing that a branch whose prediction accuracy varies over the profile-run is more likely to be input-dependent.

The goal of our project was to add support for 2D-Profiling to the Open Research Compiler (ORC). We successfully completed this task, and we were able to use the 2D-Profiling information to annotate the Control Flow Graph of a program to make decisions regarding predicated execution.

The rest of this report is organized as follows: Section 2 describes in more detail the background and motivation for this project; Section 3 provides an overview of ORC, including support for profiling and predication; Section 4 details the process of implementing 2D-Profiling in ORC; Sections 5 and 6 explain our experimental methodology and results, respectively; Section 7 discusses future work we would like to perform related to this project; and, finally, Section 8 concludes the report. We have also provided Appendices containing a description of problems we faced throughout the project (Appendix I), a sample feedback file generated by ORC with 2D-Profiling added (Appendix II), and the C code for the synthetic benchmark we created for the project (Appendix III).

2 Background and Motivation

One of the classic problems in computer architecture is how to handle branches in pipelined processors. Branch instructions have the ability to redirect the instruction stream, which means the front end must either stall and wait for the branch to resolve or predict the direction of the branch and provide a way of recovering to the correct path if the prediction turns out to be wrong. A third option is to have the compiler eliminate branch instructions from the instruction stream by using predicated execution [1]. In the following subsections, we briefly discuss the trade-offs between branch prediction and predication, as well as a technique for determining when to predicate.

2.1 Predication vs. Prediction

As the processor pipeline depth increases, the branch misprediction penalty also increases. This means that the performance gained by branch prediction is very dependent on how hard it is to predict the direction of a branch. If a branch is easy to predict, then performance can be increased significantly by having a branch predictor; however, if a branch is hard to predict, then the cost of mispredicting the branch may outweigh the benefit of having the branch predictor. One technique commonly used to eliminate hard-to-predict branches is predicated execution [1], which converts control dependencies into data dependencies. Since predication adds additional instruction overhead and data dependencies, it should only be used for branches that are actually hard to predict.

Determining which branches to predicate is a non-trivial problem, and several techniques have been proposed. The IMPACT compiler [9] uses path execution frequencies and generates predicated code to create larger basic blocks, providing a larger scope for compiler optimizations and eliminating branch mispredictions. Other algorithms for generating predicated code only convert highly mispredicted branches [2] or short forward branches [14][18]. ORC, Intel's

Open Research Compiler [13], uses edge-profiling information and equations that approximate the execution cost of normal branch code versus predicated code to decide when to predicate.

Profile information can be used by compilers to help decide whether to predicate a branch or not. Such an approach assumes that the behavior of the branch will not change when the program is run with a different input set. This assumption is not always true since the input set used for profiling may not reflect all possible program behaviors. Kim *et al.* proposed a technique to detect the input-dependence behavior of the branch using only one input-set. [8].

2.2 2D-Profiling

2D-Profiling [8] is an efficient way of predicting the set of input-dependent branches in a program at compile time using only a single input set. The key insight of this mechanism is that if the prediction accuracy of a static branch changes significantly over time during the profiling run (with a single input set), then the prediction accuracy of that branch is more likely to be input-dependent. To detect the time-varying phase behavior of a branch, a 2D-profiler records the prediction accuracy of all static branches at fixed intervals during a profiling run. A function executed at the end of each interval collects data needed for input-dependence tests that are performed at the end of the profiling run. The purpose of these tests is to determine whether the branch has shown phase behavior during the profiling run. If the branch passes the tests, it is defined as input-dependent. The details of the three tests are described in [8].

3 Overview of ORC

The Open Research Compiler (ORC) project is a joint effort between Intel Corp. and the Chinese Academy of Sciences to provide compiler and computer architecture researchers with a leading-edge open source compiler for the $Itanium^{TM}$ Processor Family (IA64). The basis for ORC is Pro64, which is an open source compiler. Pro64 includes front-ends for C/C++ and it also includes support for several code optimizations. ORC has focused on a redesign of the code generation module to exploit the richness of the IA64 architectural features and to aid researchers. Some of the IA64 optimizations include predicate analysis, if-conversion, and control and data speculation with recovery code generation [13]. ORC includes profiling feedback and associated support and region-based compilation. Most optimizations in ORC are performed on each procedure, also called a Program Unit (PU), independently.

We chose to use ORC as our compiler infrastructure because IA64 supports predicated execution, making it extremely suitable for the research in which we are interested. ORC also provides profiling support at the WHIRL level (ORC's intermediate representation) and in the Code Generation (CG) phase. Profiling at the WHIRL level is an extension of Pro64 edge profiling, and the types of profiling done at CG are edge profiling, value profiling, and memory profiling. The results from the edge profiling are used for the if-conversion profitability analysis [6]. Hence, we focus more on edge profiling performed during CG.

3.1 Edge Profiling

An overview of edge profiling support in ORC is shown in Figure 1 on the following page. First, the source is compiled to an instrumented binary. The instrumented binary is then run on an Itanium machine to generate the feedback file. Finally, the feedback file is used as an input to the compiler together with the source to generate an optimized binary.

3.1.1 Instrumentation

To generate the instrumented binary, the user invokes the compiler with a specific command line option, *fb_create*, and specifies the name of the feedback file to be created [5]. The user can also specify the type of profiling for which the binary should be instrumented and the phase at which the instrumentation should be performed. Edge profiling at CG can thus be specified from the command line as the desired instrumentation mechanism. The *instrument* method for edge profiling is invoked by the main driver of ORC if that is what the user specifies on the command line [15]. The input to the *instrument* method is the Control Flow Graph (CFG) of a PU. The job of the *instrument* method is to insert calls to the functions in the instrumentation library into the control flow graph of the PU. The functions in the instrumentation library can be split into three categories:

1) Functions that set up the data structures required to collect all the profile information and also create a header for the feedback file.
2) Functions that collect information during the profile run. For example, the functions that increment the counter when a particular edge is traversed during the execution.
3) Functions that write the header and the collected data into the feedback file at the end of the profile run.

The *instrument* function iterates through the list of all basic blocks in the PU and inserts calls to the instrumentation functions as required. Each time a call needs to be inserted, an instrumentation basic block that calls the desired function is created and

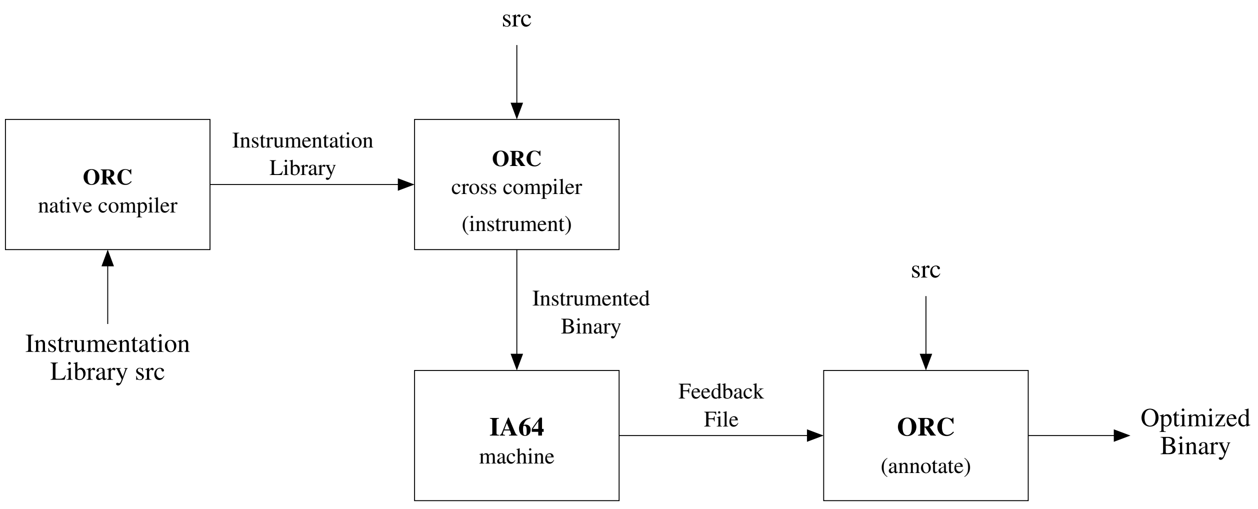


Fig. 1. Overview of edge profiling support in ORC

inserted in the CFG at the appropriate place. The arguments to the function are usually passed in as immediate values, hence minimizing the interaction of the instrumentation code with the actual code [4]. Figure 2(a) on the next page shows an example of a portion of a CFG before calling the *instrument* function, and Figure 2(b) shows the same CFG after the call to *instrument*.

The arguments that are required by each instrumentation function are dependent on the job of the function. For example, in the case of the function that collects the edge profiling data, the argument is a unique identifier for the edge to be profiled, which is passed in as an immediate value (as shown in Figure2(b)).

Once the instrumentation is done, the CFG is converted into native code to generate the instrumented binary. The instrumented binary, when run on an Itanium machine, dumps out a feedback file.

3.1.2 Annotation

Once the feedback file has been created, the compiler can be invoked with *fb_opt feedbackfilename* to specify that the compiler can read feedback from a certain file and perform optimizations. The main driver reads the header from the feedback file and decides which phase will need to use this information. If the header specifies that the file contains edge profile data, then the edge profile *annotate* method is called. Similar to *instrument*, *annotate* is also called once for each PU. The input to the *annotate* method is a CFG and the name of the feedback file. The *annotate* method reads the information from the feedback file regarding each edge in the PU and annotates the CFG with that information. For this annotation to be correct, a necessary condition is that this CFG is the exact same as the CFG that was input to the *instrument* method (when the instrumentation binary was created). The annotation involves reading information from the feedback file and storing it with each edge in the CFG. In the case of edge profiling, this information is the number of times an edge was traversed during the profile run.

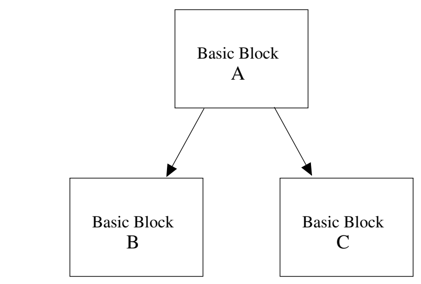
At the end of the annotation, the relative probability of executing each basic block is computed. For a basic block *Y* that is dominated by basic block *X*, the probability of executing Y is computed using Equation 1.

$$Probability(Y) = \frac{\#\, times\ execute\, Y}{\#\, times\ execute\, X} \qquad (1)$$

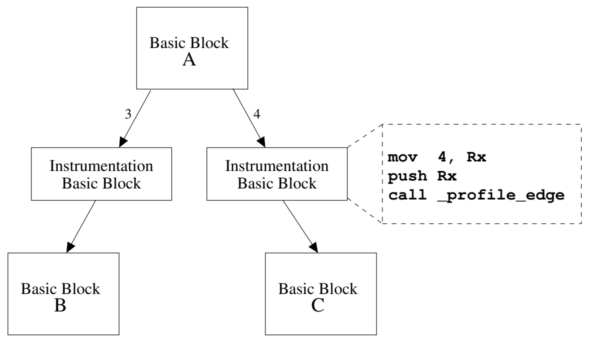
3.2 Predication Support

The feedback information from the edge profiling is used in making predication decisions in ORC. All the inner-most regions of the PU are considered for predication. Some of the regions are then selected on the basis of their control flow structure. Only simple control flow structures like *if-then* or *if-then-else* are considered good candidates for predication. Once a list of candidates has been selected, the list is passed to a method for profitability analysis.

Profitability analysis applies a very simple scheme to decide if performance is expected to improve by converting the area to predicated code. The code is converted to predicated code if the number of cycles taken by the predicated code are expected to be less than the number of cycles taken by the non-predicated code. To compute the execution time of predicated and non-predicated code, we first need to compute the number of cycles required to execute each basic block. The number of cycles required for each block are estimated using a simple heuristic: the length of the longest dependence chain in each basic block is



(a) CFG before adding instrumentation basic blocks



```
mov  4, Rx
push Rx
call _profile_edge
```

(b) CFG after adding instrumentation basic blocks

Fig. 2. Example CFG before and after calling the *instrument* function

used as an estimate for the number of cycles. The number of cycles required for predicated code are then computed as the sum of the execution times of the successor basic blocks. Equation 2 shows this computation for the CFG in Figure 2(a).

The probability of executing each block is computed using the feedback information, as described in Section 3.1, and Equation 1. An overhead of non-predicated code is the pipeline flush penalty, which occurs on a branch misprediction. In ORC, the branch misprediction rate is estimated by taking the minimum of the execution probabilities of the two successor blocks. Again using Figure 2(a) as an example, Equation 3 shows this computation. The branch misprediction penalty is set to 8 cycles in ORC. This number is set to 8 because the worst case pipeline flush penalty is 9 cycles in Itanium 1 and 6 cycles in Itanium 2 processor [11][17]. The execution time for the non-predicated code is then computed using Equation 4.

$$\begin{aligned}
&pred\_exec\_time = \\
&\quad exec\_time(B) + exec\_time(C) && (2)\\
&branch\_mispredict\_rate = \\
&\quad min(Probability(B), Probability(C)) && (3)\\
&non\_pred\_exec\_time = \\
&\quad [exec\_time(B) * Probability(B)] + \\
&\quad [exec\_time(C) * Probability(C)] + \\
&\quad [branch\_mispredict\_rate * \\
&\quad branch\_mispredict\_penalty] && (4)
\end{aligned}$$

The execution time for the predicated and non-predicated code is then compared to decide the profitability of the predicate transformation. If an area is predicated, then the analysis is run on all the areas in the PU once again to exploit any new predication opportunities that may have been created due to the transformation of the code. This continues until all possible candidates have been converted to predicated code.

4 Implementation

The goal of this project was to add support for 2D-Profiling in the ORC compiler. In addition to accomplishing this goal, we also added support for annotating the information provided by 2D-Profiling on the CFG, and applied this information in making the decisions for predication. Implementing 2D-Profiling required changes in several modules of ORC. We used the infrastructure already available in the compiler for edge profiling as our base. The work we did in developing the infrastructure can be split into three main tasks as described below.

4.1 Adding a *gshare* Branch Predictor

4.1.1 Why is it necessary?

One of the requirements for 2D-Profiling, and also ORC predication analysis, as described in Section 3.2, is that we should be able to estimate the branch misprediction rate of conditional branches with high accuracy. The 2D-Profiling mechanism was proposed assuming that an accurate estimate of branch misprediction rate will be available at profile time. On the other hand, ORC instrumentation does not incorporate a branch predictor. They estimate branch misprediction rate using Equation 3. A similar estimate can also be used for implementing 2D-Profiling provided that the estimate is accurate. We did some studies using the Pin Binary Instrumentation Tool [16] to evaluate the error when the misprediction rate is estimated using Equation 3. First, we implemented a Pin Instrumentation tool which simulated a 4-KB *gshare* [10] branch predictor, and then, we collected the statistics required to compute the execution probability of each basic block. Later, we used the results from the tool to compare the branch misprediction rate from the actual

TABLE I

ACTUAL BRANCH MISPREDICTION RATE FROM *gshare* VS. ESTIMATED MISPREDICTION RATE

Benchmark	bzip2	crafty	eon	gap	gcc	gzip	mcf	parser	perlbmk	twolf	vortex	vpr
Gshare	1.90	12.40	12.21	5.66	7.33	7.49	7.84	9.13	5.09	16.42	0.78	11.20
Estimated	2.04	15.62	13.72	7.79	8.88	9.87	13.49	15.37	6.89	18.90	0.91	11.36
Weighted Mean Sq. Error	2.15	10.87	7.90	7.50	8.13	6.47	10.71	12.96	8.15	11.37	2.67	4.69

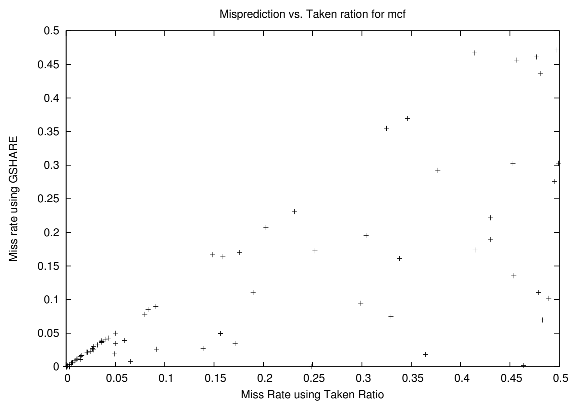


Fig. 3. A comparison between estimated misprediction rate and actual misprediction rate

branch predictor with the misprediction rate estimated using Equation 3.

The results are shown in Table I. The Weighted Root Mean Square Error (WRMSE) value was also computed in addition to the overall branch misprediction rates. WRMSE is a better measure of error than the overall misprediction rate since negative and positive errors can cancel each other when the overall prediction accuracy is computed. The error contribution from each branch was weighted according to the execution frequency of the branch. We discovered that the WRMSE is greater than 5% for 9 out of the 12 SPEC Integer benchmarks. The value is greater than 10% for crafty (10.87%), mcf (10.71%), parser (12.96%), and twolf (11.37%). An intuitive explanation for this error is that Equation 3 does not estimate the misprediction rate for branches that may be always taken for the first half of the program and always not-taken for the remaining half. In such a case, Equation 3 will estimate the misprediction rate to be 50%, but the actual misprediction rate will almost be 0%. Intuitively, it is also unlikely to be the case that the branch misprediction rate estimated using Equation 3 is lower than the actual misprediction rate.

Figure 3 shows a scatter plot to compare the estimated misprediction rate with the actual misprediction rate for the branches in *mcf* benchmark. Only the branches that were executed more than 10,000 times during the execution of the train input set are shown on this plot. It can be seen that there are some branches (lower left corner) that have the same actual and estimated misprediction rate, but these branches have a very low misprediction rate and are unlikely to be predicated. However, for the branches where the misprediction rate is somewhat higher, it can be seen that almost all of them are in the lower triangle where the estimated misprediction rate is higher than the actual misprediction rate. This means that using Equation 3 to estimate the misprediction rate can lead to a different decision than if the actual misprediction rate, which is more accurate, was used. We quantify this difference in Section 6.

4.1.2 Implementation of the Branch Predictor

Based on all of the above data, we decided to proceed with implementing a *gshare* simulator inside the profiling library for ORC. To simulate a *gshare* branch predictor, we need the address and outcome of all the conditional branches. Since the support provided by ORC for edge profiling already inserts an instrumentation basic block at each edge, we decided to take advantage of this existing support. The normal edge profiling mechanism calls a function in the instrumentation library with a unique identifier for each edge. In addition to the edge identifier, we also require some more information regarding the edges, for example if two edges are of the same conditional branch instruction. To accomplish this, we decided to pass a unique branch identifier, together with an edge identifier, to the instrumentation function. We also handled the case when an edge was connected to an unconditional branch by having a bit to represent if the edge was actually connected to a conditional branch. Finally, we need to know which direction of the branch an edge corresponds to in order to update the history register in *gshare*, compare our prediction, and compute the misprediction rate. We passed the direction information as another argument to the instrumentation function.

One other piece of information that was required was the address of the branch, which is not available until the binary is generated. Since the instrumentation function call is created during the instrumentation phase, we did not have access to the address of the actual branch. We decided to use a simple approach and use the branch identifier we passed to the profile function as the address of the branch in the *gshare* predictor. The idea was based on the assumption that

TABLE II
BRANCH MISPREDICTION RATE FROM *gshare* VS. GSHARE-ARBIT

Benchmark	bzip2	crafty	eon	gap	gcc	gzip	mcf	parser	perlbmk	twolf	vortex	vpr
Gshare	1.90	12.40	12.21	5.66	7.33	7.49	7.84	9.13	5.09	16.42	0.78	11.20
GSHARE-arbit	1.85	12.24	12.14	6.29	7.70	7.54	7.76	9.41	5.48	17.39	0.88	11.72
Weighted Mean Sq. Error	1.77	3.89	0.46	7.90	4.99	0.78	0.53	1.08	4.27	3.70	4.08	3.58

the performance of *gshare* is not heavily dependent on the actual address of the branch, but instead it just uses the address to get a better hash function into a 2-bit counter prediction table. Hence, even using an arbitrary branch identifier should give results similar to the actual *gshare* branch predictor. We validated this assumption by simulating another *gshare* branch predictor that used an arbitrary branch identifier in its hashing function. We called this branch predictor *GSHARE-arbit*. Table II shows the results of comparison between *gshare* and GSHARE-arbit. It can be seen that GSHARE-arbit has a WRMSE of less then 5% for 11 out of 12 SPEC benchmarks. The only benchmark where the WRMSE is greater than 5% is *gap*. We did not get a chance to investigate the reason for this difference in the *gap* benchmark, but we plan to do this in the future.

Based on the above results, we implemented the GSHARE-arbit predictor in the instrumentation library. The predictor is called every time an edge is encountered that has the conditional branch bit set. The branch identifier and the outcome of the branch are passed in as arguments to the predictor. The predictor makes a prediction for the branch, compares the prediction with the outcome, and updates the profile stats.

At the end of the profiling run, the stats collected by the branch predictor are written in the feedback file. The information is then read by the compiler and annotated on the CFG. 2D-Profiling information for each branch is stored in the basic block where the branch operation actually exists. This information can be accessed by any of the later phases in the compiler.

4.2 Adding 2D-Profiling

We implemented the 2D-Profiling algorithm as described in [8]. The branches that are identified as input-dependent by the 2D-Profiling mechanism are marked during the profiling run. The input dependence information is also written to the feedback file together with the branch prediction information. Similarly, the information is read from the feedback file and annotated on the CFG along with the branch prediction information. A sample feedback file is attached as Appendix II

4.3 Predication and Wish Branches

One of the target uses of the 2D-Profiling information and the branch prediction accuracy is to make better predication decisions. We made a change in ORC's predication mechanism such that it could use the misprediction rate of a conditional branch from new feedback instead of using Equation 3 to estimate the branch misprediction. We also added support to count the number of input-dependent branches that can be predicated. Finally, since the target application of 2D-Profiling is Wish Branches [7], we did a potential study for how many branches can be actually converted into Wish Branches.

5 Methodology

5.1 Experimental Setup

We used the ORC cross compiler together with natively compiled instrumentation libraries to generate our instrumented binaries. The instrumentation binaries needed to run on actual IA-64 machines; however, the IA-64 machines that we had available for this project could not be used for this purpose. The reason is described in Appendix I. Instead, we had to run the instrumented binaries on NUE IA-64 emulator [12]. NUE is a tool that was developed by HP for IA-64 emulation on IA-32 machines. NUE allowed us to run the IA-64 instrumented binaries, but at a significantly slower rate compared to a real Itanium machine. We were, however, able to run the non-instrumented binaries on a real machine; hence, we could run both the baseline and optimized binaries on the real machine and note differences in execution time. The Itanium machine we used was one of the HP TestDrive machines that are available for public use. We used the machine td187, which has 16 Intel Itanium 2 processors running at 1500MHz. The machine runs the Red Hat Enterprise Linux AS 4.0 operating system.

5.2 Benchmarks

We used a subset of the SPEC CPU2000 Integer benchmark suite for our studies. All benchmarks were run to completion, and we used the test or train input sets for the profiling run of the benchmark (due to extremely long execution times). The benchmarks and input sets we used are reported in Table III.

TABLE III

BENCHMARKS AND INPUT SETS USED

eon	train
gap	test
gzip	test
mcf	test
parser	test
vortex	test

TABLE IV

NUMBER OF INPUT DEPENDENT BRANCHES IDENTIFIED

Benchmarks	Static Branches	Input Dep Branches
eon	2014	26
gap	2099	192
gzip	418	40
mcf	164	25
parser	1889	679
vortex	5853	285

In addition to the SPEC benchmarks, we also used one synthetic benchmark that we created specifically for this project. This benchmark enabled us to directly measure the impact of our added enhancements to the predication support in ORC on an actual Itanium 2 machine. The source code for our benchmark is available in Appendix III. The benchmark contains a *while* loop that executes a fixed number of times. There is an *if* statement in the while loop for which the taken rate can be adjusted by the using a command line input. This enables us to study differences in performance as the branch misprediction rate changes.

5.3 2D-Profiling Setup

We used different parameters for 2D-Profiling than proposed by Kim *et. al.* in [8]. We had to adjust two parameters since we used the test input set for the majority of our benchmarks instead of the train input set. We set the size of each slice to 300 million branches instead of 1500 million branches since our experiments are shorter. Similarly, we used an *exec_threshold* of 500 instead of 1000 since the size of each slice is shorter. The rest of the parameters were used as proposed in [8].

6 Results

In this section we will discuss the results of our experiments. In addition, we will also attempt to provide explanations for the results.

6.1 2D-Profiling Results

We measured several different statistics for each of the benchmarks during their profile run. Table IV shows the number of static conditional branches that were seen in the execution of each of the benchmarks. The table also shows the number of branches that were identified as input-dependent by 2D-Profiling. The number of input-dependent branches in each benchmark differs significantly from [8]. There are several reasons that can explain these changes. Some of the most eminent reasons are the change in ISA (IA64 vs. x86), change of compiler (ORC vs. gcc 3.3.3), input set (test vs. train), and the difference in 2D-Profiling parameters (stated in Section 5.3).

Figure 4 on page 10 displays the distribution of all input-dependent branches, which are classified into six different categories based on their prediction accuracy. The data shows that the majority of the input-dependent branches are easy to predict. In Figure 5 on page 10, we present similar data by plotting the fraction of input-dependent branches in each branch prediction category. The figure shows that a high percentage of branches with prediction accuracy greater than 90% are input-dependent.

TABLE V

PREDICATION OF INPUT DEPENDENT BRANCHES

Benchmark	MAX	est-predicated	bp-predicated	Disagree
eon	2200	895	888	7
gap	1620	1094	1083	23
gzip	78	39	37	2
mcf	37	21	20	5
parser	210	124	123	21
vortex	649	334	336	18

6.2 Predication

We also did some studies to find out how many branches that can be predicated in ORC are actually input-dependent. Table V shows our results. Remember that, in ORC, only the inner-most branches that have simple control flow structures are possible candidates of predication. We call the set of these branches *MAX*. Profitability analysis is performed in ORC to select a subset of these branches that are then actually predicated. Originally, ORC used an estimated branch misprediction rate in the profitability analysis. We call the set of branches that were chosen for predication using estimated branch misprediction rate *est-predicated*. The set of branches that were chosen for predication when the actual branch misprediction rate from our branch predictor was used are called *bp-predicated*. The *Disagree* column shows the number of times the decision to predicate or not differed between the two misprediction rate schemes. It can be seen that several predication decisions will be different if the actual misprediction rate was used in place of the estimated misprediction rate. This can translate into noticeable performance improvement if that particular branch is executed very frequently.

Figure 6 shows the the fraction of input-dependent branches among all the branches that can be predicated (*MAX*). These branches can thus be converted to Wish Branches for future applications. Figure 7 shows the fraction of input-dependent branches among *est-predicated* branches. Since these branches are identified as input-dependent, it implies that they may result in loss of performance when the program is run with a different input set than the one that was used for the profiling run.

6.3 Experiments with Synthetic Benchmarks

We compiled the instrumented binary for the synthetic benchmark. We then set the taken ratio of the third if-statement such that it was taken for the first half of the benchmark and not-taken for the second half of the benchmark. As expected, the branch misprediction rate from the actual predictor was less than 1%, whereas the misprediction rate estimated using the taken ratio was close to 50%. Therefore, when the estimated branch misprediction rate was used, the compiler predicated the third branch. We also generated another binary where the actual branch misprediction rate was used to decide the worth of predication. In this case, the compiler did not predicate the branch. We ran both the binaries, predicated and unpredicated, on the real Itanium 2 machine. We observed a performance improvement of 5% in execution time when using the binary generated using the actual branch misprediction rate.

7 Future Work

We plan to further analyze the results we obtained from 2D-Profiling and evaluate these results using the metrics used in [8]. This will also help us find optimal values for 2D-Profiling parameters in an IA-64 setting since the previous parameters are for a different ISA on a different compiler. We will also need to run the instrumented binaries with the train input set for our results to be more correct. Doing so will require access to Itanium machines that can run the instrumented binaries.

Even though it may not impact our results, we still intend to research different techniques to enhance GSHARE-arbit. We would like the accuracy of this predictor to be as close to an actual hardware predictor as possible.

A separate but interesting study is to analyze the difference in 2D-Profiling results between the x86 setting and IA-64 setting. Understanding the differences between the two architectures and the two compilers can lead to several interesting research topics.

Another important study is to compare the performance overhead of 2D-Profiling instrumentation to edge profiling instrumentation. This will require running the instrumented binaries on an actual Itanium 2 machines. We do not have access to machines that can run instrumented binaries at this point, but we plan to get this data once we have the resources.

8 Conclusion

We have added the support for 2D-Profiling in ORC as a part of this project. 2D-Profiling is a mechanism that identifies input-dependent branches. This input-dependence information can help make better predication decisions in ORC. We implemented the mechanism to generate instrumented binaries in ORC that could gather 2D-Profiling data and write it in a file. We also developed the mechanism in ORC to read the feedback file and annotate the data on the Control Flow Graph of the program. Finally, we present our results for identifying input-dependent branches. Using our synthetic benchmark, we show the potential benefit of having 2D-Profiling information for making predication decisions.

References

[1] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 177–189, New York, NY, USA, 1983. ACM Press.

[2] Po-Yung Chang, Eric Hao, Yale N. Patt, and Pohua P. Chang. Using predicated execution to improve the performance of a dynamically scheduled machine with speculative execution. In *PACT '95: Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques*, pages 99–108, Manchester, UK, UK, 1995. IFIP Working Group on Algol.

[3] Joseph A. Fisher and Stefan M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *ASPLOS-V*, 1992.

[4] Intel Research Corp. and Institute of Computing Technology, CAS. *ORC Source Code*. orc-2.1-src.tar.gz.

[5] Intel Research Corp. and Institute of Computing Technology, CAS. *ORC FAQs*, July 2003.

[6] R. Ju, S. Chan, T.-F. Ngai, C. Wu, Y. Lu, and J. Zhang. Open research compiler (orc) 2.0 and tuning performance on itanium. Tutorial presented at Micro 35, November 2002.

[7] Hyesoon Kim, Onur Mutlu, Yale N. Patt, and Jared Stark. Wish branches: Enabling adaptive and aggressive predicated execution. *IEEE Micro*, 26(1):48–58, 2006.

[8] Hyesoon Kim, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. 2d-profiling: Detecting input-dependent branches with a single input data set. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 159–172, Washington, DC, USA, 2006. IEEE Computer Society.

[9] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 45–54, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[10] Scott McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, 1993.

[11] Cameron McNairy and Don Soltis. Itanium 2 processor microarchitecture. *IEEE Micro*, 23(2):44–55, 2003.

[12] NUE. Nue and ski product information. http://www.hpl.hp.com/research/linux/ski/nue-info.phpes.

[13] ORC. Open research compiler for itanium processor family. http://ipf-orc.sourceforge.net, May 2006.

[14] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 16–27, New York, NY, USA, 1990. ACM Press.

[15] S. Pop. Interface and extension of the open research compiler. Technical report, INRIA, 2002.

[16] Vijay Janapa Reddi, Alex Settle, Daniel A. Connors, and Robert S. Cohn. Pin: A binary instrumentation tool for computer architecture research and education. In *Proceedings of the Workshop on Computer Architecture Education*, June 2004.

[17] Harsh Sharangpani and Ken Arora. Itanium processor microarchitecture. *IEEE Micro*, 20(5):24–43, 2000.

[18] Gary Scott Tyson. The effects of predicated execution on branch prediction. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 196–206, New York, NY, USA, 1994. ACM Press.

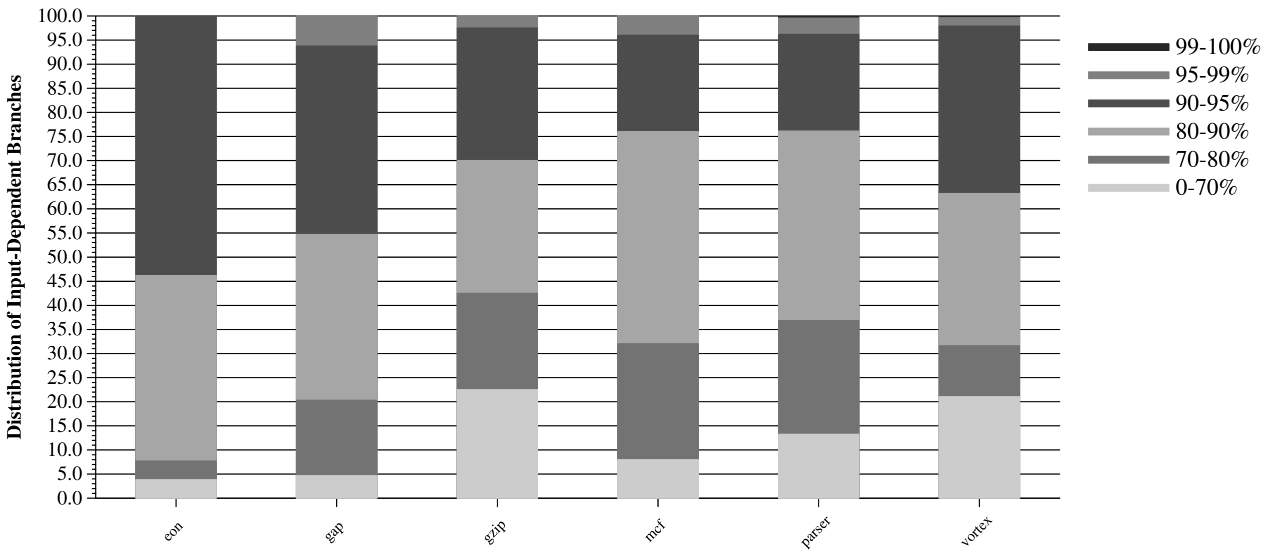


Fig. 4. The distribution of input-dependent branches based on their branch prediction accuracy

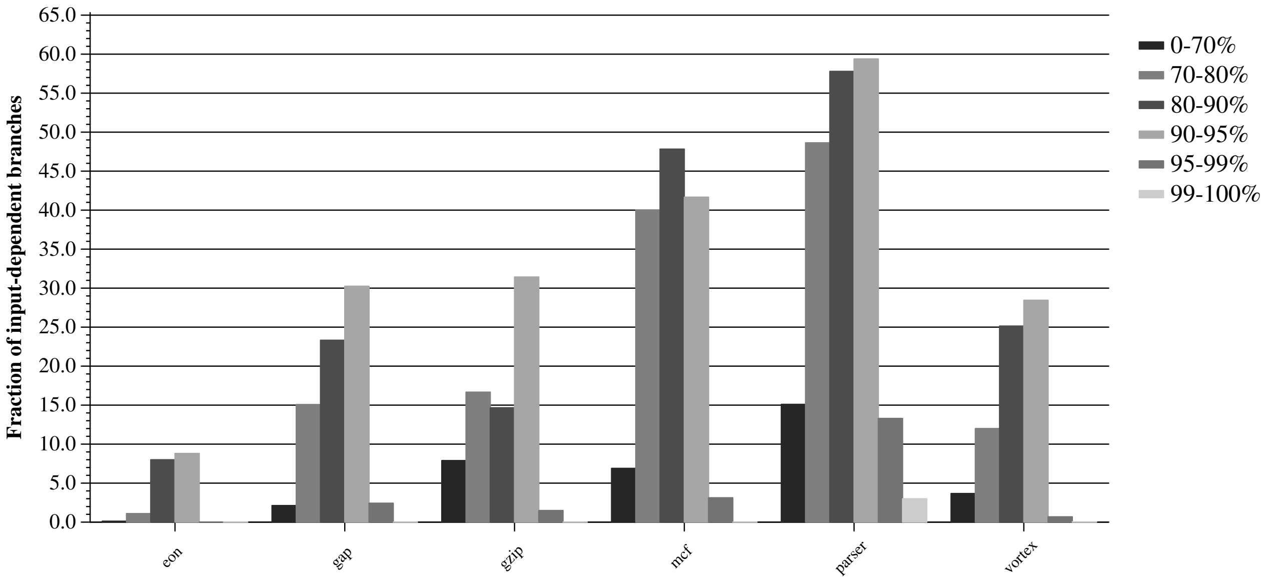


Fig. 5. The fraction of input-dependent branches in different prediction accuracy categories

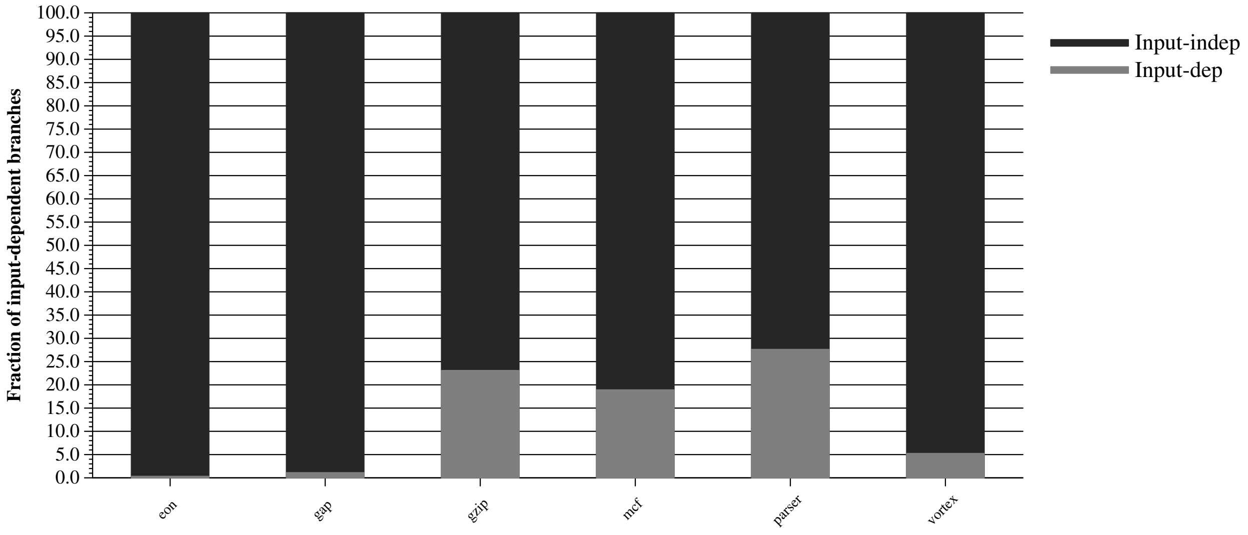


Fig. 6. The fraction of input-dependent branches in MAX

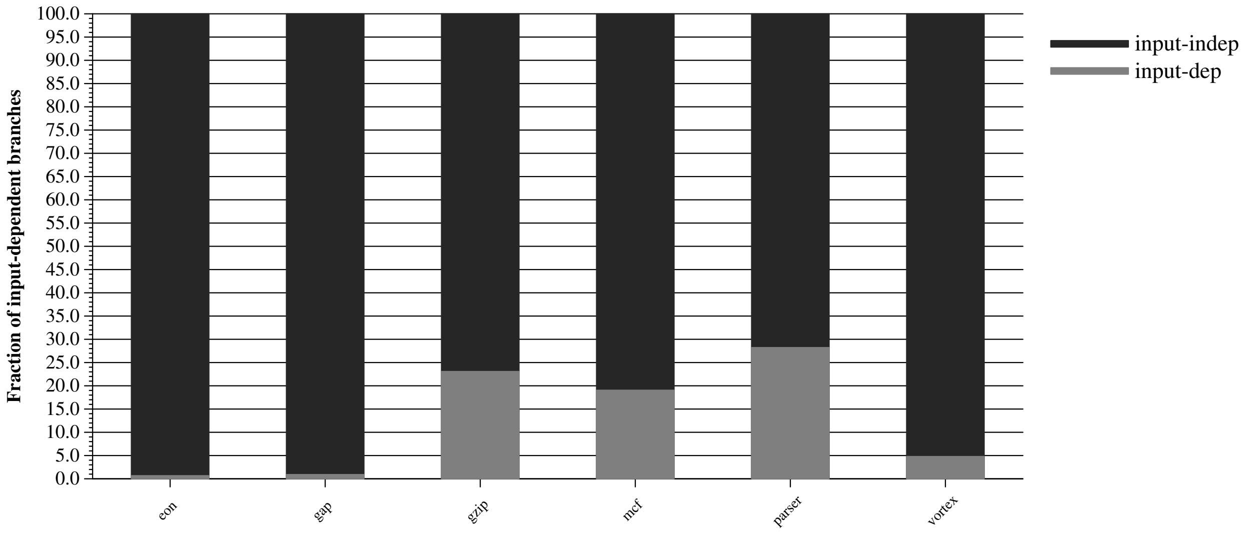


Fig. 7. The fraction of input-dependent branches in *est-predicated*

Appendix I
Problems faced

Overall this project has been a great learning experience. Dealing with a real compiler has taught us several new lessons. There were several problems that we faced while dealing with ORC during the course of this project that caused delays and hindered our progress. We want to list some of them here so others working with ORC do not have to suffer from them.

1) We were unable to run the instrumented binaries that were compiled using the ORC cross compiler on a real Itanium machine. We were able to run the non-instrumented binaries correctly. After much investigation we realized that it was an issue with the glibc version. Only the instrumentation libraries were compiled using pre-built ORC binaries inside NUE. However, the glibc version inside NUE was not compatible with the glibc version on the Itanium machines we had access to. Therefore, the instrumented binaries crashed during the execution of instrumentation code.
2) To find a solution for the first problem, we even tried to compile the native ORC compiler on the Itanium machine. However, the native ORC did not compile successfully since ORC requires gcc 2.96 for its compilation, which is not available on the native machine.
3) Another problem we faced was related to writing and reading the feedback file. The size of the data structure that was written to the file and that was read from the file was different. This resulted in garbage data being read and annotated from the feedback file. On further investigation, we found that the structure we were writing out to the file had a variable of type UINT32. This variable was interpreted as a 64-bit unsigned integer when the instrumentation libraries were compiled using the pre-built ORC native binaries inside NUE. Whereas, UINT32 was considered a 32-bit unsigned integer in the cross compiler build on the native x86 machine. Changing the type of the variable to UINT64 solved the problem since now it was 64-bit at both ends.
4) This problem was more related to the interaction between ORC and the Linux utility mktemp. ORC uses the Linux utility “mktemp” to generate unique filenames for the feedback file. This ensures that ORC never overwrites an older feedback file. We noticed that the Linux operating system actually marks all these files as temporary files. These files are deleted automatically after some time. We had to repeat several experiments since we unexpectedly lost feedback files.

Appendix II
Sample feedback file

```
Start to dump data from feedback file: back_up

**********      FILE HEADER     **************
fb_ident = 0123456789abcde
fb_version = 2
fb_profile_offset = 0
fb_pu_hdr_offset = 48
fb_pu_hdr_ent_size = 128
fb_pu_hdr_num = 1
fb_str_table_offset = 176
fb_str_table_size = 12
phase_num = 4

**********   PU Header No 0   **************
pu_checksum = 25
pu_size = 54321
runtime_fun_address= 65542
pu_name_index = 0
pu_file_offset = 0
pu_inv_offset = 188
pu_num_inv_entries = 0
pu_br_offset = 188
pu_num_br_entries = 0
pu_switch_offset = 188
pu_switch_target_offset = 188
pu_num_switch_entries = 0
pu_cgoto_offset = 188
pu_cgoto_target_offset = 188
pu_num_cgoto_entries = 0
pu_loop_offset = 188
pu_num_loop_entries = 0
pu_scircuit_offset = 188
pu_num_scircuit_entries = 0
pu_call_offset = 188
pu_num_call_entries = 0
pu_icall_offset = 333
pu_num_icall_entries = 0
pu_handle = 11111
pu_edge_offset = 188
pu_num_edge_entries = 25
pu_instr_count = 0
pu_instr_exec_count = 0
pu_value_offset = 1788
pu_ld_count = 0
pu_stride_offset = 1788

************   Str table    **************
No 0 : main.c/main

*********** PU Data No 0 ************
0: _type =   1   |  _value =          1 | v=0 |N=         0 |STD=  0.000000 |MEAN=  0.00000
1: _type =   1   |  _value =          1 | v=0 |N=         0 |STD=  0.000000 |MEAN=  0.00000
2: _type =   1   |  _value =          1 | v=0 |N=         0 |STD=  0.000000 |MEAN=  0.00000
```

```
3: _type =   1   |  _value =         1 | v=0 |N=         0 |STD=  0.000000 |MEAN=  0.00000
4: _type =   1   |  _value =         0 | v=0 |N=         0 |STD=  0.000000 |MEAN=  0.00000
5: _type =   1   |  _value =         1 | v=0 |N=         0 |STD=  0.000000 |MEAN=  0.00000
6: _type =   1   |  _value =         1 | v=1 |N=        20 |STD=  0.000000 |MEAN=  1.00000
7: _type =   1   |  _value =     99999 | v=1 |N=        20 |STD=  0.000000 |MEAN=  1.00000
8: _type =   1   |  _value =     50001 | v=0 |N=         0 |STD=  0.000000 |MEAN=  0.00000
9: _type =   1   |  _value =     49999 | v=0 |N=         0 |STD=  0.000000 |MEAN=  0.00000
10: _type =   1   |  _value =     49999 | v=1 |N=        20 |STD=  0.000000 |MEAN=  0.9998
11: _type =   1   |  _value =     50001 | v=1 |N=        20 |STD=  0.000000 |MEAN=  0.9998
12: _type =   1   |  _value =     50038 | v=0 |N=         0 |STD=  0.000000 |MEAN=  0.0000
13: _type =   1   |  _value =     49962 | v=0 |N=         0 |STD=  0.000000 |MEAN=  0.0000
14: _type =   1   |  _value =     49962 | v=1 |N=        20 |STD=  0.000013 |MEAN=  0.4995
15: _type =   1   |  _value =     50038 | v=1 |N=        20 |STD=  0.000013 |MEAN=  0.4995
16: _type =   1   |  _value =    100000 | v=0 |N=         0 |STD=  0.000000 |MEAN=  0.0000
17: _type =   1   |  _value =         1 | v=0 |N=         0 |STD=  0.000000 |MEAN=  0.0000
18: _type =   1   |  _value =         0 | v=0 |N=         0 |STD=  0.000000 |MEAN=  0.0000
19: _type =   1   |  _value =         0 | v=0 |N=         0 |STD=  0.000000 |MEAN=  0.0000
20: _type =   1   |  _value =         0 | v=0 |N=         0 |STD=  0.000000 |MEAN=  0.0000
21: _type =   1   |  _value =         1 | v=0 |N=         0 |STD=  0.000000 |MEAN=  0.0000
22: _type =   1   |  _value =         1 | v=1 |N=         0 |STD=       nan |MEAN=
23: _type =   1   |  _value =         0 | v=1 |N=         0 |STD=       nan |MEAN=
24: _type =   1   |  _value =         1 | v=0 |N=         0 |STD=  0.000000 |MEAN=  0.0000
************   End of dump   **************
```

Appendix III
Synthetic Benchmark

```
#include<stdio.h>
#include<stdlib.h>

int array[20];

int init(){
  array[ 0  ] = -10;
  array[ 1  ] = -9;
  array[ 2  ] = -8;
  array[ 3  ] = -7;
  array[ 4  ] = -6;
  array[ 5  ] = -5;
  array[ 6  ] = -4;
  array[ 7  ] = -3;
  array[ 8  ] = -2;
  array[ 9  ] = -1;
  array[ 10 ] = 0;
  array[ 11 ] = 1;
  array[ 12 ] = 2;
  array[ 13 ] = 3;
  array[ 14 ] = 4;
  array[ 15 ] = 5;
  array[ 16 ] = 6;
  array[ 17 ] = 7;
  array[ 18 ] = 8;
  array[ 19 ] = 9;
}


int main(int argc, char** argv)
{
  int k = 0;
  int first_taken=0;
  int first_ntaken = 0;
  int second_taken=0;
  int second_ntaken = 0;
  long long max_k = 0;
  int value = 0;
  int period;

  init();

  if ( argc == 1 ){
    max_k = 100000;
  }
  else if ( argc == 2 ){
    max_k  = atoi ( argv[1] );
  }
  else if (argc == 3){
    max_k  = atoi ( argv[1] );
    period = atoi ( argv[2] );
  }
  else{
    exit(0);
  }

  while ( k < max_k ){
    if ( rand() > (RAND_MAX/2) ){
      first_taken++;

    }
    else{
      first_ntaken++;
    }


    if ( !(k % (period/2)) ){
      value = (value == 0 )?1:0;
    }

    if ( value ){
      second_taken++;
      array[0] = second_taken;
      array[1]=array[0]+array[0];
      array[3]=array[1]+array[0];
      array[4]=array[2]+array[1];
      array[5]=array[3]+array[2];
      array[6]=array[4]+array[3];
      array[7]=array[5]+array[4];
      array[8]=array[6]+array[5];
      array[9]=array[7]+array[6];

    }
    else{
      second_ntaken++;
      array[10] = second_ntaken;
      array[11]=array[10]+array[10];
      array[13]=array[11]+array[10];
      array[14]=array[12]+array[11];
      array[15]=array[13]+array[12];
      array[16]=array[14]+array[13];
      array[17]=array[15]+array[14];
      array[18]=array[16]+array[15];
      array[19]=array[17]+array[16];
    }

    k++;
  }

  printf("first_taken   : %d \n", first_taken);
  printf("first_ntaken  : %d \n", first_ntaken);
  printf("second_taken  : %d \n", second_taken);
  printf("second_ntaken : %d \n", second_ntaken);

}
```