Complex Branch Profiling for Dynamic Conditional Execution

Rafael R. dos Santos¹, Tatiana G. S. dos Santos¹, Mauricio L. Pilla¹, Philippe O. A. Navaux¹, Sergio Bampi¹, Mario Nemirovsky²
¹ Universidade Federal do Rio Grande do Sul
Porto Alegre, RS, Brazil
{rrsantos,tatiana.pilla,navaux,bampi}@inf.ufrgs.br
² Kayamba
San Jose, CA, USA
{mario@ieee.org}

Abstract—
Branch predictors are widely used as an alternative to deal with conditional branches. Despite the high accuracy rates, misprediction penalties are still large in any superscalar pipeline. DCE, or Dynamic Conditional Execution, is a alternative to reduce the number of predicted branches by executing both paths of certain branches thus reducing the number of predictions and therefore the misprediction occurrence. The goal of this work is to analyze the complexity of branch structures and determine the number of branches that can be predicated in DCE and the distribution of mispredictions according to the classification proposed. The complex branch classification proposed extends the classification presented by Klauser [KLA98]. As result, it is showed that an average of 35% of all branches can be predicated in DCE and around 32% of mispredictions fall into these branches.

Keywords—Superscalar architectures, Multipath execution, Branch prediction, Dynamic predication

I. INTRODUCTION

Conditional branches present a challenge to increase performance in current processors. Many mechanisms have been presented to mitigate their impact, like branch prediction, speculative execution, trace caches, instruction prefetching, and multipath execution. These mechanisms rely on fetching more instructions to feed the starving functional units [UHT95, TYS97, HEI96, KLA98, SAN98, SKA99].

The most common technique, implemented in all modern processors, is branch prediction. Branch prediction is widely used to predict the next instruction fetch address when conditional and/or unconditional branches are fetched. Nevertheless, despite the accuracy of current predictors, mispredictions still degrades considerably the performance.

Superscalar pipelines are getting deeper to support increasing demands for clock frequency. Therefore when more stages are added, the penalty imposed by branch mispredictions increases. The accuracy of state of the art branch predictors, however, is not improving.

Increasing the accuracy of current predictors implies an extraordinary increase in complexity. Since the predictor has

now to be split among several stages of fetch, many levels of predictions are necessary, again, due to the high clock frequencies required.

One can avoid branch predictions by means of executing both paths of a conditional branch. For instance, multipath execution was extensively studied in the past but simply executing all paths of all branches proved not to be efficient.

DCE [SAN01, SAN03] presents an alternative where only certain conditional branches are predicated. DCE can perform dynamic predication of simple and complex conditional branches without requiring a special instruction set nor special compiler optimizations hence it can be applied to legacy code.

This paper presents an analysis on the behavior and patterns of direct conditional branches in order to better understand the dynamics of control structures and to quantify the number of branches that can be predicated. First, DCE is briefly presented in Section II. Then, Section III introduces a classification for hammocks that extends the classification presented by Klauser et al [KLA98]. The simulation environment is presented in Section IV, and the result analysis is discussed in Section V. The conclusions are drawn in the last section.

II. DYNAMIC CONDITIONAL EXECUTION

DCE combines dynamic predication and multipath to reduce the complexity and disruptions of the fetch. This is achieved by fetching sequentially through branches that qualify for predication.

In order to determine if a branch qualify for predication, an extension of the selection mechanism proposed in [KLA98] was developed. In their selection mechanism, only simple branches qualify for predication. The selection scheme used in DCE also qualify complex branches.

The selection mechanism is static and runs at compilation time, marking branches which can be predicated according to the target locality. The compiler does not change the

*Grants from CNPq, CAPES and UNISC
original code, it only marks instructions valid for predication [SAN03]. At execution time, the fetch engine decides whether a selected branch will be predicted or not, based on the availability of resources.

Therefore, DCE is a combination of a static selection mechanism and hardware support to execute certain branches eagerly. In DCE, selected branches are treated, by the fetch engine, as regular instructions and they never disrupt the fetching. As no prediction of control transfer is made at fetch time for these instructions, they can not cause mispredictions.

The main difference between Klauser et al. [KLA98] and DCE [SAN03] is that DCE predicates both simple and complex branches and it does not use conditional moves to satisfy data dependences at the join point of predicated branches.

In Klauser et al., conditional moves are inserted dynamically at the joint point to block the issue of instructions from the same data chain of the predicted paths. When the branch resolves, the conditional moves can be issued to copy the data from the correct physical register to the correct source register. Thus, consumer instructions that use the respective register become ready for issue only after the conditional move instruction has been executed.

In DCE, a register renaming technique derived from Chaves et al. [CHA99] generates replicas of instructions at the join point of predicated branches. Replicas are instructions that use data that is produced in one of the paths of a predicated branch. Therefore, there is one replica for each path predicated. As DCE does not use conditional moves, it does not block the issue of the dependent instructions. In DCE, replicated instructions can be issued as soon as the the corresponding source physical registers are ready (i.e. the source data is available).

If an instruction at the join point reads a register that is produced logically before a predicated branch, DCE will not introduce multiple replicas of that instruction since the instruction is control independent [ROT99] and data independent with relation to the path being predicated.

Figure 1 shows the misprediction reduction for the different classes of branches in DCE. Simple is the same class of branches predicated in the machine proposed by Klauser et al. The variations of complex are the extended model proposed here which include Simple and other 4 combinations of nested hammocks.

The horizontal axis presents the different configurations simulated varying the number of mapping tables (M) and the architecture width (W). The number of mapping tables determines the maximum number of active paths supported and the width determines the number of instructions that can be renamed/issued/executed/committed per cycle. Also, there are a maximum of 64 instructions between branches to be predicated and their join points (Distance 64).

Remarkably, predicating Simple branches allows a reduction of 3.3-5.3% of mispredictions but the reduction remains almost constant when the number of mapping tables is increased, i.e., it does not improve when the ability to predicate more branches increases. Nonetheless, the misprediction reduction when predicing complex branches almost doubles when the number of mapping tables is increased, which shows the potential of the technique.

III. Hammock Classification

Two important questions for the performance of DCE are: (i) how many branches have near targets and contain only valid nested branches; and (ii) how many mispredictions fall into these branches. In order to answer these questions, the behavior of branches found in general applications were studied and further classified. This classification defines the predication limits in DCE. The goal is to find the best set of branches to be selected and optimize the architecture accordingly.

The compiler was modified to analyze the static code and look for conditional forward branches. It then classifies the branches according to the number and type of nested branches that they contain and the distance to the taken target.

A conditional forward branch that has no nested branches may have one (if-then) or two sides (if-then-else). These branches are called simple hammocks single sided (a) or double sided (b), respectively (figure 2).

Conditional forward branches that have other conditional forward branches inside may be classified as follows (figure 3):

1. one or more nested conditional forward branches totally contained are called pure complex hammocks (a)
2. one or more conditional forward hammocks whose target address coincide with the join address of the outer hammock are called multiple join complex hammocks (b)
3. one or more conditional forward branches whose target
address is beyond the join address of the outer hammock are called multiple target complex hammocks (c) One or more conditional forward branches whose target address targets the body of the taken path are called overlap complex hammocks (d) A hammock may not qualify for predication due to the occurrence of any of the following: backward branches, indirect branches, unconditional jumps that are NOT related to one or more conditional branches (then path skip jump), subroutine calls or returns and system calls. Figures 2 and 3 presented the five different hammock classifications. The diagrams presented there are such as each number corresponds to an instruction and each arrow represents a branch to a given target, i.e. another instruction. The source of the arrow is the instruction which originated the branch (conditional or unconditional). The end of the arrow indicates the taken target instruction.

Figure 2 (a) presents the most basic hammock structure. It is a typical if-then, where a condition is investigated in instruction 1 and, depending on the result, the instruction flow continues sequentially or is redirected to instruction 8. Note that instruction 8 is part of any flow path started in 1, taken or not, so it is called join point. Because in this structure there are no nested branches and there is only one side (if-then), this category is called One-sided Simple.

Figure 2 (b) shows a two-sided hammock. The example corresponds to an if-then-else, which has a condition evaluation in instruction 1 and a unconditional branch in a later instruction, represented by instruction 7. This unconditional branch is responsible for the flow redirection demanded by the else command. It is possible to see that the unconditional branch is the instruction right before the target of 1, i.e. instruction 8. In this example, the join point is given at instruction 10 and the category is called Two-sided Simple. For instance, for a branch to fall into this category it must have the unconditional jump right before the target of the first conditional branch (branch delay slots were not considered). If this condition is not true then the branch falls off this valid category.

If a hammock has any nested hammocks it is then called Complex. It is important to mention that the classification for complex branches presented here extends the terminology and analysis once presented by [KLA98] addressing in more details the peculiarities of complex hammocks.

An example of a Complex pattern is showed in Figure 3 (a). In this case, there are nested hammocks within the outer hammock. The outer hammock is an if-then-else similar to the one presented earlier, where instruction 1 jumps to 8, if the condition evaluated is true. Furthermore, there is an unconditional jump in instruction 7, jumping to the join point, i.e. instruction 10. Inside this hammock, instruction 3 is a simple if-then, like the one in the first example of this section. For this example of nested hammocks, the target of the second branch is totally contained within the most external hammock, instruction 5. When all nested branches have their targets totally contained within the boundaries of the most external branch, that branch is called Complex Pure.

Figure 3 (b), presents an if-then-else hammock with multiple join points. This means that one of the sides of the most external branch (then or else) has a nested branch whose target is the same as the first, most external branch. In the example, instruction 8 is the target of two conditional branches (instructions 1 and 3). Then, the most external branch has the same target as the most internal branch and then it is called Complex with multiple joins. The join point of this hammock is instruction 10 as this instruction is the first instruction common to any path starting in 1. Observe though that branch 3 would have a join point at 8 if it was not a nested branch of 1. When classifying complex hammocks the join point is considered to be the first instruction common to all paths starting from the outer hammock.

In other cases, nested branches may not have targets that are coincident with the target of the external branch. In this cases the target may be inside the else path while the branch is within the then path or it may be beyond the join point of the external branch, Figures 3 (c) and (d).

When the target of a nested branch, located within the then
path, is actually in the else path of the most external branch, example (c), the two branches are overlapped and the category in each they are included is called Complex overlapped. The join point is still the common instruction to all paths, i.e., instruction 10.

Example (d) shows the nested branch 3 which has a target I2 beyond the join point of the most external branch 1. In this case the join point is instruction I2 as it is the first instruction common to any path that starts in 1. This type of Complex branch is called Complex with multiple targets.

DCE is designed to handle any possible combination of the above described categories of simple and complex hammocks. The complexity level is given by the highest number of paths or instructions required to execute all paths according to the DCE architecture and is as follow, in increasing levels of complexity: pure, multiple joins, multiple targets and overlapped, respectively.

IV. SIMULATION ENVIRONMENT

The simulator used in the experiments was developed using the SimpleScalar tool set [BUR96], which includes a very detailed cycle simulator. The simulator implements out-of-order issue using a Register Update Unit (RUU) [SOH90] and was leveraged with code provided by Klausner et al [KLA98] for hammock profiling. The original hammock profiling algorithm was extended to handle complex hammocks and their variations.

The main goal of this work is to understand the behavior and pattern of short conditional branches in order to allow a better design of the DCE architecture. Although the performance of the architecture is not the goal of this work, the accuracy of the branch predictor employed in the experiments is relevant in the study of the results presented.

An hybrid branch predictor mechanism was used, combining bimodal and 2-level adaptive branch predictors [MCF93]. The configuration adopted was the following:
a meta-table with 2048 entries; a BTB with 2048 entries and a 2-bit saturated counter (bimodal portion); one single entry in the first level (gshared xor) and 8192 entries in the second level with 13 history bits (2-level adaptive predictor).

Each benchmark was compiled with 3 different optimizations (–O1, –O2 and –O3) using gcc version 2.6.3. This methodology was used in order to observe the impact on the number of short branches introduced by ach optimizations. In addition, the option –funroll–loops was used in all 3 levels of optimizations.

The benchmarks used in simulations were ammp, equake, gcc, gzip, parser, vpr_place, vpr_route and vortex, all from the SPEC2000 suite. For each configuration, 2 billion instructions were simulated except for benchmarks vpr_place and vpr_route which were simulated up to their completion.

V. RESULTS

In this section, the results obtained through simulation are presented. The dynamic analysis is discussed and conditional branches are classified according with the taxonomy presented previously.

In DCE, any branch qualifying to one of the categories presented can be predicated. Non-qualifying branches need to be predicted. The architecture of DCE was designed and optimized to handle any branch that qualifies according to the rules discussed earlier. Those are the branches that can provide improvements in terms of reducing the misprediction as they dont need to be predicted.

Both the compiler (parser) and simulator were modified to instrument the code. The parser provides information on to whether a branch qualifies for predication or not according to the DCE methodology. This selection is static. During simulation, branches are monitored and counted according to their category. Mispredictions are then counted also based on the category of the branch that produces the misprediction.

In the simulations performed in this section, a conventional superscalar simulator was used. No predication was actually applied in these simulations as predication would change the ratio of branches predicted thus changing the results and misprediction rates.

A. Total number of branches

Table I presents the number of branches executed of each category per benchmark, defined by the static analysis of the source code. The first column indicates the optimization option and the number of sides (one or two). The second column defines the categories, while the remaining columns show the percentage of branches in each category, based on the number of dynamic branches counted in the simulations.

There is almost no difference in the average of qualified branches for the different optimization levels. However, the distribution of branch categories changes from one optimization level to another, due to the way that the compiler reorganizes the code in order. The level of complexity tends to increase as the optimization level is increased.

Most qualified branches are classified in either Simple or Pure Complex categories, and most of them are also single-sided branches (if–then constructions). One-sided branches produce less overhead than two-sided branches during the predication as only one extra path is added to the execution in order to guarantee that no misprediction will happen.

When a one-sided branch is predicted, then path is executed only when the condition evaluated is true. However, the not taken path is always executed regardless of the condition as it is control independent. Therefore, when a one-sided branch is predicated, only one extra path (the then path) is added to the execution.

When two-sided branches are predicated, both the not
Table I: Classified hammocks

<table>
<thead>
<tr>
<th>Hammocks</th>
<th>Unamp</th>
<th>Equal</th>
<th>Gec</th>
<th>Gec</th>
<th>Forrest</th>
<th>versus</th>
<th>Place</th>
<th>Route</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-side</td>
<td>Simple</td>
<td>Comp</td>
<td>Multi Jain</td>
<td>Multi Jain</td>
<td>Multi Jain</td>
<td>Multi Jain</td>
<td>Multi Jain</td>
<td>Multi Jain</td>
</tr>
<tr>
<td></td>
<td>4.09</td>
<td>9.70</td>
<td>9.20</td>
<td>15.62</td>
<td>15.62</td>
<td>9.09</td>
<td>1.79</td>
<td>33.31</td>
</tr>
<tr>
<td>G0</td>
<td>Comp</td>
<td>1.07</td>
<td>7.62</td>
<td>9.27</td>
<td>2.74</td>
<td>5.87</td>
<td>16.47</td>
<td>11.04</td>
</tr>
<tr>
<td>O1</td>
<td>Multi Jain</td>
<td>0.19</td>
<td>1.58</td>
<td>0.54</td>
<td>8.45</td>
<td>0.03</td>
<td>0.14</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>Multi Tag</td>
<td>0.67</td>
<td>4.92</td>
<td>3.95</td>
<td>0.08</td>
<td>1.68</td>
<td>1.12</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>Multi Overl</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Two-side</td>
<td>Simple</td>
<td>40.79</td>
<td>21.03</td>
<td>0.34</td>
<td>8.55</td>
<td>2.86</td>
<td>0.04</td>
<td>4.71</td>
</tr>
<tr>
<td>G0</td>
<td>Multi Jain</td>
<td>3.23</td>
<td>3.06</td>
<td>1.38</td>
<td>4.70</td>
<td>0.34</td>
<td>0.29</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>Multi Tag</td>
<td>0.30</td>
<td>2.08</td>
<td>0.60</td>
<td>0.04</td>
<td>1.66</td>
<td>0.01</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>Multi Overl</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Total</td>
<td>Simple</td>
<td>45.29</td>
<td>25.25</td>
<td>9.26</td>
<td>18.25</td>
<td>5.72</td>
<td>2.83</td>
<td>6.02</td>
</tr>
<tr>
<td>G0</td>
<td>Multi Jain</td>
<td>1.53</td>
<td>1.58</td>
<td>0.56</td>
<td>8.36</td>
<td>0.03</td>
<td>0.14</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>Multi Tag</td>
<td>0.67</td>
<td>4.92</td>
<td>3.95</td>
<td>0.08</td>
<td>1.68</td>
<td>0.68</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>Multi Overl</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Two-side</td>
<td>Simple</td>
<td>40.79</td>
<td>21.03</td>
<td>0.34</td>
<td>8.55</td>
<td>2.86</td>
<td>0.04</td>
<td>4.71</td>
</tr>
<tr>
<td>G0</td>
<td>Multi Jain</td>
<td>3.23</td>
<td>3.06</td>
<td>1.38</td>
<td>4.70</td>
<td>0.34</td>
<td>0.29</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>Multi Tag</td>
<td>0.30</td>
<td>2.08</td>
<td>0.60</td>
<td>0.04</td>
<td>1.66</td>
<td>0.01</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>Multi Overl</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Total</td>
<td>Simple</td>
<td>45.29</td>
<td>25.25</td>
<td>9.26</td>
<td>18.25</td>
<td>5.72</td>
<td>2.83</td>
<td>6.02</td>
</tr>
<tr>
<td>G0</td>
<td>Multi Jain</td>
<td>1.53</td>
<td>1.58</td>
<td>0.56</td>
<td>8.36</td>
<td>0.03</td>
<td>0.14</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>Multi Tag</td>
<td>0.67</td>
<td>4.92</td>
<td>3.95</td>
<td>0.08</td>
<td>1.68</td>
<td>0.68</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>Multi Overl</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Figure 4. Qualified and non-qualified branches

![Figure 4](image1)

![Figure 4](image2)

Figure 5. Average size of qualified hammocks

![Figure 5](image3)

Figure 5. Average size of qualified hammocks

The average of qualified hammocks exceeds 40% of all conditional branches for the -O2 optimization level. This means that using O2, the set of qualified branches is larger and more branches are going to be selected for predication. DCE will, then, dynamically decide whether the predication is really going to take place based on the dynamic availability of resources.

Although the optimization -O2 produces more qualifying branches the difference is not large which does not disqualify other levels of optimizations for the application of DCE.

B. Average size of hammocks

Another important aspect for DCE is the average length of each branch. This size can also be defined as the number of static instructions between the branch and its target. In DCE, the shorter the branch, the less expensive will be the predication.

A summary of the average hammock length is shown in Table II. The first column presents the number of sides and the optimization options. The hammock category is indicated
in the second column, and the remaining columns show the average distance in instructions from a branch to its target. For two-sided hammocks, the size is defined as the average between the then and the else sides. Thus, the average number of instructions for a two-sided hammock is twice the value referred in Table II.

Figure 5 presents the average size of all qualified branches for the optimization options. The number of instructions appears in the vertical axis, while the benchmarks are shown in the horizontal axis. The average size of qualified hammocks is relatively small, ranging from 5 to 12 instructions per side.

C. Branch mispredictions

Another important issue for DCE is whether the optimization options affect the misprediction rates. Benchmarks with large misprediction rates and short hammocks tend to benefit more from the DCE predication. Figure 6 shows the misprediction for the three optimization levels.

For benchmarks ammp, gcc and parser, most of the mispredictions occur in non-qualified branches, reducing the performance improvements that may be obtained in DCE. On the other hand, quake, gzip, vortex, vpr_place and vpr_route presented a significant number of mispredictions in branches that may be predicated by DCE, thus showing a significant potential for performance improvement. An interesting aspect is that even when using -O3, which turns on function inlining, the number of qualified branches remains almost the same. As most non-qualified branches are caused by function calls, it was expected that function inlining would provide more branches suitable to be predicated. However, simulations showed that the code inserted by inlining functions contains invalid, complex structures that cannot be exploited by DCE. This phenomenon is going to be studied in a future research.

In the next graph (Figure 7), the average branch mispredictions for all benchmarks is shown grouped by optimization level. As mentioned before, the optimization levels do not significantly impact misprediction rates for the studied benchmarks.

The potential of mispredictions that can be avoided by DCE predating all qualified hammocks, without considering resource limits, is about 32%. But it is important to consider the size of a hammock when deciding whether to pursue predication, since the overhead of extra instructions may hide the benefits of avoiding the mispredictions.

VI. CONCLUSIONS

This work studied the structure of branches in terms of hammocks for a set of benchmarks from SPEC CPU 2000 to identify patterns that are generated by the compiler and qualify for eager execution in DCE.

Results have shown that 35% of all conditional branches
The IPC of DCE was compared against a very aggressive reference machine. The details about DCE are not discussed here due to the space limitations but can be found in [SAN03].

The results show that the predication of complex branches, in special complex pure can provide as much as 50% increase in speedup for vary wide architectures. While single branches provide constant speedups across the different configurations used, the potential for exploiting complex branches exists but is limited mainly by the resource contention as a large overhead is generated.

The overhead introduced by multiple instructions from different paths decreases the performance, but alternatives, such as instruction reuse, are being studied. In a preliminary study [SAT03] it was found that almost 30% of the instructions executed in DCE can be reused and that traces of reusable instructions are in some cases twice as big as traces of reusable instructions when only prediction is applied.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the support from CNPq/CAPES Brazilian agencies for R&D and UNISC in the form of scholarships and grants. The authors also thank Arthur Klauser for providing the simple hammock parser which leveraged the development of the complex parser and DCE.

Table II: Average size of valid hammocks

<table>
<thead>
<tr>
<th>Hammocks</th>
<th>Simple</th>
<th>Compiler</th>
<th>Main</th>
<th>Multi-Join</th>
<th>Multi-Targ</th>
<th>Multi-Over</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>O1</td>
<td>1.64</td>
<td>4.79</td>
<td>2.31</td>
<td>4.09</td>
<td>1.85</td>
<td>0.55</td>
<td>3.82</td>
</tr>
<tr>
<td>O2</td>
<td>1.54</td>
<td>4.62</td>
<td>1.74</td>
<td>3.14</td>
<td>2.98</td>
<td>0.59</td>
<td>3.84</td>
</tr>
<tr>
<td>O3</td>
<td>1.64</td>
<td>4.81</td>
<td>2.31</td>
<td>4.09</td>
<td>1.85</td>
<td>0.55</td>
<td>3.82</td>
</tr>
</tbody>
</table>

The results show that the predication of complex branches, in special complex pure can provide as much as 50% increase in speedup for vary wide architectures. While single branches provide constant speedups across the different configurations used, the potential for exploiting complex branches exists but is limited mainly by the resource contention as a large overhead is generated.

The overhead introduced by multiple instructions from different paths decreases the performance, but alternatives, such as instruction reuse, are being studied. In a preliminary study [SAT03] it was found that almost 30% of the instructions executed in DCE can be reused and that traces of reusable instructions are in some cases twice as big as traces of reusable instructions when only prediction is applied.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the support from CNPq/CAPES Brazilian agencies for R&D and UNISC in the form of scholarships and grants. The authors also thank Arthur Klauser for providing the simple hammock parser which leveraged the development of the complex parser and DCE.

Figure 7. Average branch misprediction can be predicated, and that 32% of all branch mispredictions occur in these branches. The chance to work around this large percentage of all mispredictions can represent a great improvement in the overall performance.

Most of the non-qualified hammocks were caused by function calls, and even inlining these functions did not increase the number of selected hammocks. Thus, special code generation for these cases is an interesting work to be developed in the future.

Although the study of the DCE performance is not the main goal of this work, graph 8 presents the DCE speedup comparing the predication of complex branches against single branches. Three benchmarks of the SPEC95 with the highest misprediction rates were used in this simulations.
REFERENCES


