Branch Behavior of a Commercial OLTP Workload on Intel IA32 Processors
Murali Annavaram, Trung Diep, John Shen
Microprocessor Research, Intel Labs (MRL)
{Murali.m.annavaram, trung.diep, john.shen} @intel.com

Abstract
This paper presents a detailed branch characterization of an Oracle based commercial on-line transaction processing workload, Oracle Database Benchmark (ODB), running on an IA32 processor. We ran a well-tuned ODB on Simics, a full system simulator, to collect the instruction traces used in this study. We compare the branch behavior of ODB with the branch behaviors of gcc, gzip and mcf from the SPECINT 2000 benchmark suite. Contrary to the popular belief that databases have unpredictable branches, we show that using larger predictors that capture enough branch history information, and using branch prediction schemes that reduce aliasing, conditional branches in ODB are more predictable than in gcc, gzip and mcf. Due to frequent context switching in ODB, a hardware return address stack is ineffective in predicting return addresses for ODB. Based on further analysis, we propose and evaluate an enhanced return address predictor, which reduces return address mispredictions in ODB by 40%.

1. Introduction
The increasing need to store and query large volumes of data has made database management systems (DBMSs) one of the most prominent applications to run on shared memory multiprocessor servers. Until recently processor research studies have largely ignored DBMSs in preference to scientific and integer applications such as SPECINT 2000. This bias is due to the lack of representative database workloads, proprietary nature of database source code, and the complexity of properly configuring a database system. With the advent of widely available processor simulators, and binary instrumentation tools that do not need source code access, researchers were better able to characterize database applications. As a result, several recent studies analyzed memory system behavior of DBMSs running on Alpha processors [2][3][4][5][7][8]. However, in spite of the prevalence of IA32 processor based database servers, few results are published on performance of database applications running on IA32 processors[6]. In particular, we are not aware of any work that analyzed branch behavior of database workloads on IA32 processors. Accurate branch prediction is essential to realize the performance potential of deep-pipelined superscalar processors, which are widely used in current database servers. We believe that understanding branch behavior of DBMSs is an important first step towards designing efficient branch predictors for future processors.

The key contributions of this paper are summarized below.
1. This paper presents a detailed branch characterization of an Oracle based OLTP workload, Oracle Database Benchmark (ODB), running on an IA32 processor.
2. By carefully exploring the design space of conditional branch predictors, we show that conditional branches in ODB are more predictable than in gcc, gzip and mcf from the SPECINT 2000 benchmark suite. ODB branch prediction accuracy can be improved by using larger predictors that reduce capacity misses, and by using branch predictors that reduce aliasing. These results counter the popular belief that databases suffer from higher conditional branch mispredictions than benchmarks such as SPECINT 2000.
3. We show that due to frequent context switching return address mispredictions are significantly higher in ODB. We analyze the reasons for return address mispredictions and propose a simple modification to a conventional return address predictor to improve prediction accuracy for ODB.

The rest of this paper is organized as follows. Section 2 describes related work on DBMS performance analysis. Section 3 explains ODB tuning and system level simulation process. Section 4 presents general characteristics of ODB. Section 5 presents our branch classification process and Section 6 presents results from our branch prediction experiments. We conclude in Section 7.

2. Related Work on DBMS Analysis
It is only recently that researchers have examined the performance impact of architectural features on DBMSs[2][3][4][5][9][12]. Most of these studies show that database applications have large instruction and data footprints and exhibit more unpredictable branch behavior than benchmarks that are commonly used in architectural studies (e.g. SPEC). Database applications have short loop counts and suffer from frequent context switches, causing significant increases in the instruction cache miss rates[3][1]. Ailamaki et al. [4] analyzed three commercial DBMSs on a Xeon processor and showed that TPC-D queries spend about 20% of their execution time on branch misprediction stalls and 20% on L1 instruction cache miss stalls. Several previous studies [2][3][4][6] analyzed memory system behavior of OLTP workloads based on monitoring existing systems using performance counters. Monitoring studies can quickly analyze large-scale production systems and can identify critical performance bottlenecks. These studies are, however, limited to studying existing
processor and system configurations and have limited capability to explore new design choices. Nevertheless, monitoring existing systems is an important tool for quickly tuning and scaling complex OLTP workloads. In our study, we used this approach to tune ODB on native hardware before starting simulations. Ranganathan et al. [8] used trace-driven simulations to study the benefits of out-of-order execution and superscalar design features for OLTP workloads. They used a 1KB gshare branch predictor and concluded that databases have higher branch misprediction rate than SPEC benchmarks. In our study, we focus only on branch behavior of ODB and show that most branch mispredictions in ODB are due to aliasing. Aliasing can be reduced either by increasing predictor size or by using prediction schemes such as yags that are specifically designed to reduce aliasing. Once aliasing is reduced, ODB actually has fewer branch mispredictions than gcc, gzip, and mcf.

This study differs from previous studies in one key respect: It presents a detailed branch characterization of a commercial OLTP workload while most previous studies characterized memory system performance.

3. Workload & Simulation Environment

3.1 Benchmark Description

In this study, we used an Oracle 8.1.6 based OLTP workload, which we call the Oracle Database Benchmark (ODB). ODB simulates an order-entry business system, where terminal operators (or clients) execute transactions against a database. The database is made up of a number of warehouses. Each warehouse supplies items to ten sales districts, and each district serves three thousand customers. Typical transactions include entering and delivering customer orders, recording payments received from customer, checking status of a previously placed order, and query the system to check inventory levels at a warehouse.

When ODB starts execution, it spawns three types of processes: user processes, server processes and background processes. A user process executes client’s application code, such as parsing a query, and submits an appropriate data access request to a server process. A server process accesses database on behalf of a user and provides user requested data. More than 90% of ODB execution time is spent in server processes. Background processes perform database maintenance tasks such as log management, committing modified data to disk, and managing locks amongst competing server processes. All Oracle processes, server as well as background processes, share a large memory segment called the System Global Area (SGA). A large portion of SGA is devoted to the database buffer cache, which holds the working set of a database in memory. The database buffer cache tracks the usage of the database blocks to keep the most recently and frequently used blocks in memory, significantly reducing the need for disk I/O.

To compare ODB with more widely studied benchmarks, we selected gcc, gzip and mcf from SPECINT 2000 benchmark suite.

All benchmarks are compiled on an Intel Pentium III workstation running Red Hat Linux 7.2 using the gcc compiler, with –O2 optimizations.

3.2 Setup and Validation of ODB

Commercial workloads, such as ODB, need careful tuning to reduce I/O wait time. A production ODB run typically uses hundreds of warehouses and requires hundreds of gigabytes of disk space and tens of gigabytes of physical memory. Such a setup is typically not amenable for detailed system level simulation studies. Hence, in this study we used an in-memory ODB setup, where the working set fits in memory with negligible amount of disk I/O.

To achieve the goal of running ODB completely in-memory we tune the setup parameters; increasing the SGA size, reducing number of warehouses to reduce the working set size, and using enough clients for optimal concurrency. Tuning requires repeated execution of user transactions after modifying each setup parameter and even building database from scratch multiple times. Since these operations are very time consuming, we first develop and test ODB on a native machine before running on a simulator.

We use a Pentium III system with 2 GB main memory and 70 GB disk space. Initially we use a 50-warehouse ODB workload. Our measurements from the native machine execution show that the idle loop overhead due to I/O stalls is nearly 50%. By trial and error, we scale down the ODB workload to use 10 warehouses, which occupies 35 GB of disk space. We configured the Oracle server to use 1.5 GB of memory for SGA, which is sufficient to cache frequently used blocks in memory, significantly reducing the need for disk I/O. A production ODB run typically uses hundreds of warehouses and requires hundreds of gigabytes of disk space and tens of gigabytes of physical memory. Nevertheless, some disk I/O does occur; mostly non-critical writes of the redo log buffers. Our tuned version of ODB has less than 2% idle loop overhead. After tuning ODB, we build a complete and exact disk image of the ODB workload that can then be executed by theSimics simulator.

3.3 Tracing Using Full System Simulator

We use Simics [10] a full system simulator, to simulate the tuned ODB workload identical to that ran on the native hardware. Figure 1 graphically depicts the various steps involved in our tuning and simulation process. Simics is a complete system level simulator that is capable of booting several unmodified commercial operating systems and running unmodified application binaries. Since we tuned ODB on a Pentium III running Linux Red Hat 7.2, we configured Simics to model a Pentium III processor.
As described in Section 3, ODB uses many concurrent threads (55 during our trace collection run) for maintaining undo logs, performing I/O and supporting lock management. Because of such high concurrency, ODB on average has a context switch every 75000 instructions. On the other hand, most context switches in gcc, gzip and mcf benchmarks are triggered due to timer interrupts that occur after completion of each OS allocated time quantum. Hence, these benchmarks context switch every 5-6 million instructions. All benchmarks experience negligible amount of idle loop overhead due to I/O wait. Hence, I/O read/writes are not critical bottlenecks in our benchmarks.

<table>
<thead>
<tr>
<th></th>
<th>ODB</th>
<th>gcc</th>
<th>gzip</th>
<th>mcf</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS Code</td>
<td>23%</td>
<td>4%</td>
<td>1%</td>
<td>0.4%</td>
</tr>
<tr>
<td>Branches</td>
<td>19%</td>
<td>20%</td>
<td>16%</td>
<td>26%</td>
</tr>
<tr>
<td>Switches</td>
<td>13,610</td>
<td>210</td>
<td>164</td>
<td>147</td>
</tr>
<tr>
<td>CPU Idle</td>
<td>0.00007%</td>
<td>0.002%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Processes</td>
<td>55</td>
<td>14</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Mem Reads</td>
<td>42%</td>
<td>28%</td>
<td>27%</td>
<td>43%</td>
</tr>
<tr>
<td>Mem Writes</td>
<td>22%</td>
<td>24%</td>
<td>18%</td>
<td>8%</td>
</tr>
<tr>
<td>I/O Reads</td>
<td>10,492</td>
<td>227</td>
<td>36</td>
<td>4</td>
</tr>
<tr>
<td>I/O Writes</td>
<td>25,781</td>
<td>10,850</td>
<td>10,880</td>
<td>10,012</td>
</tr>
</tbody>
</table>

Table 1 Benchmark Characteristics

5. Branch Classification

We classified branch instructions into six categories. Each category is associated with a branch predictor best suited to predict either target address or direction of that category.

1. Uncond: These are unconditional taken branches with Instruction Pointer (IP)-relative targets. An IP-relative target can be computed at decode time by adding displacement to the current IP. For the purpose of branch prediction experiments, we assumed that an IP-relative branch target is precomputed and stored with the instruction in the I-cache. Hence, Uncond branches do not need branch direction or target prediction.

2. Cond: These are conditional branches with IP-relative targets. When a conditional branch is executed, a branch direction predictor is used to obtain a branch direction prediction.

3. Call: All function call instructions with an IP-relative target are classified under this category. In our branch prediction experiments, these instructions are treated as unconditional branches. However, unlike the Uncond instructions, Call instructions push the instruction address immediately following the call onto the hardware Return Address Stack (RAS).

4. Ret: A callee function returns to its caller using return instructions. These instructions are treated as
unconditional taken branches. Their target address, however, is predicted by popping an address from RAS.

5. *Call Ind*: Indirect function call instructions whose targets are either computed at runtime or obtained from a register or memory. The target addresses of these instructions are predicted using an Indirect Branch Target Buffer (IBTB), as described in Section 6. *Call Ind* instructions also push the instruction address immediately following the call onto RAS.

6. *Jmp Ind*: These are branch instructions whose targets are computed from a register or memory. For the purpose of branch prediction, both *Jmp Ind* and *Call Ind* use same IBTB for branch target prediction.

![Figure 2 Branch Classification Distribution]

*Figure 2* shows the dynamic branch execution frequencies of the six branch categories as a percentage of total dynamic branches. The contribution of each of the six branch categories to the overall branch instructions remains almost the same for gcc and ODB. However, in gzip and mcf over 95% of all branches are either unconditional or conditional branches. Although the code size of ODB is nearly an order of magnitude larger than gcc, both exhibit similar histogram of branch instruction types; we surmise that control-flow constructs used in Oracle might be similar to those used in gcc.

6. Branch Prediction Results

This section presents results and analysis from our branch prediction experiments.

6.1 Conditional Branch Prediction

We implemented two conditional branch predictors, namely *gshare* [4] and *yags* [11] for direction prediction. The *gshare* predictor is an effective scheme, which has been shown to achieve nearly 90% prediction accuracy on integer benchmarks. However, gshare performance is limited by destructive aliasing that occurs when two branches map to the same PHT entry, but exhibit opposite branch behavior. The *yags* predictor has been shown to reduce aliasing significantly. In this study, as suggested in [11], when using a small *yags* predictor (1-5KB), 80% of the predictor space is allocated to bimodal predictor. When using a larger *yags* predictor (> 5KB), 50% of predictor space is allocated to bimodal predictor and 50% to the directional PHTs.

*Figure 3* shows the direction misprediction rates of conditional branches using gshare and yags predictors of various sizes. For the results presented in this paper we use a 6-bit PHT tag in yags predictor. Since it is not feasible to implement a 1KB and 4KB yags, we compare the performance of a 1.25KB yags with a 1KB gshare, and a 5KB yags with a 4KB gshare.

![Figure 3 gshare and Yags Misprediction Rates]

Three observations can be made from the gshare graph:

1. ODB suffers from significantly higher misprediction rates than gcc and gzip, when using gshare of less than 16KB. For instance, when using a 4KB gshare, ODB has 11.6% mispredictions rate as compared to 7.2% for gcc.

2. With a 16KB gshare, misprediction rate of ODB is comparable to gcc and gzip, and better than mcf.

3. Misprediction rate for ODB continues to decrease with increasing predictor size. However, other benchmarks do not benefit from larger than a 16KB gshare. Thus, surprisingly, misprediction rate of ODB is less than that of gcc, gzip and mcf when predictor size is more than 16KB.

From the above observations, we conclude that ODB has better control-flow predictability. When using a small branch predictor, however, ODB suffers from higher misprediction rate, due to destructive aliasing and capacity misses.

The second graph in *Figure 3* shows the misprediction rates using yags. As shown in [11], destructive aliasing can be significantly reduced by using the yags predictor. Two key observations can be made from this graph:

1. The yags predictor reduces ODB misprediction rates significantly over gshare when predictor size is less...
than 16KB. This observation further strengthens our conclusion that ODB suffers from destructive aliasing when using a small gshare. As the predictor size increases, destructive aliasing in PHT decreases. Hence, the advantage of yags over gshare decreases. Beyond 16KB yags performs slightly worse than gshare on ODB.

2. Our results also show that when using a yags predictor of less than 16KB, misprediction rates for gcc, gzip and mcf are higher than gshare. Eden et al. [11] showed that for applications that do not suffer significantly from destructive aliasing, using a small yags predictor could potentially reduce performance, due to the splitting of critical predictor resources. Thus, we conclude that these benchmarks do not suffer as much from destructive aliasing as ODB.

Ranganathan et al. [8] used a 12-bit (1 KB) gshare on an OLTP workload and concluded that OLTP workloads have significantly higher mispredictions than integer benchmark. Our results when using a 1 KB gshare corroborate with their results. However, they did not use longer history lengths to evaluate the effectiveness of gshare. Using a more detailed analysis, we conclude that a significant number of branch mispredictions in ODB are due to destructive aliasing. Destructive aliasing can be reduced either by increasing the gshare predictor size or by using a more complex prediction scheme such as yags. Once destructive aliasing is reduced, ODB actually has fewer branch mispredictions than gcc.

We also implemented two variations of the yags predictor: using a PHT tag size of 8 bits, and set associative PHTs. However, our results (not presented in this paper) show that performance improvements are marginal.

6.2 Return Address Prediction

We implemented the Return Address Stack (RAS) scheme suggested by Jourdan et al. [13] to predict return targets. Whenever a call instruction is executed, the address of the instruction immediately following the call is pushed onto RAS. When a return instruction is executed, an address from the top of RAS is popped and used as the predicted target address of that return.

Figure 4 shows the effect of varying the RAS size on return target mispredictions as a percentage of the dynamic return instruction count. Misprediction rates for ODB are much higher than for gcc, gzip and mcf.

By analyzing mispredicted return addresses and the contents of RAS immediately before and after a misprediction, we found three main reasons for RAS mispredictions:

1. When RAS size is small, a large fraction of mispredictions is due to capacity misses. Capacity mispredictions can be reduced by increasing the size of RAS. As can be seen from Figure 4, increasing RAS size from 32 to 128 reduces misprediction rate for all benchmarks. Increasing RAS size further, however, does not reduce the mispredictions significantly. We computed the distribution of the outstanding function call depths and found out that most call depths are less than 128 in all our benchmarks.

2. RAS mispredictions also occur when a callee function does not return to its caller. Instead, callee returns to an ancestor function, a function that was invoked several functions before the caller. In this case, the actual return address of the callee does not match the address on top of RAS; it will match with an address that is n-entries below the top of RAS.

3. The third reason for an RAS misprediction is when a function returns to an address that was never pushed on RAS. This anomaly can occur when a program explicitly modifies a return address stored in its stack memory before executing a return instruction. This program manipulation is not visible to RAS, as entries in RAS can be modified only by call and return instructions. Linux uses this technique to do context switching. As described in Section 3, ODB uses several concurrent processes and has frequent context switches. Frequent context switching in ODB results in such anomalous program manipulation of return addresses.

Figure 5 splits all return address mispredictions in ODB into six categories. The component labeled top-n shows percentage of mispredictions where the actual return address matches an address that is n-entries below RAS.
For instance, the bottom component in each bar (top-1) shows the percentage of mispredictions when the actual return address matches an address one entry below RAS top. The component labeled \( \geq \)top-5 shows when the actual return address matches five or more entries below RAS top. The top most component in each bar shows when none of the RAS entries matches the actual return address. These mispredictions may be caused either by capacity overflows or by return addresses that were never pushed onto RAS. However, from Figure 4 it is clear that a 128 entry RAS eliminates most capacity mispredictions. Hence, the topmost component in each bar corresponding to 128 and 512 entry RAS can be attributed to return instructions whose targets were never pushed onto RAS. Another interesting observation from Figure 5 is that nearly 40% of return addresses that were mispredicted actually match an address within the first four entries below RAS top.

6.3 Enhanced Return Address Predictor

Based on our observations from Figure 5, we implemented an enhanced RAS scheme, called Variable Pop RAS (VarPopRAS). In simple RAS scheme when a callee returns to an ancestor, its return address will be mispredicted. Furthermore, all subsequent returns, with no intermediate calls, will also result in mispredictions; even when subsequent functions return normally to their callers. In VarPopRAS, when a return instruction is executed, an address from RAS top is used as the predicted return target. However, if the predicted target does not match the actual return address, we compare the actual target address with each of the addresses stored in the top four RAS entries. If there is a match, the new RAS top is set to the entry below the matching entry. If there is no match, the predicted target address that was initially popped from RAS is pushed back onto RAS thereby leaving RAS unchanged.

By popping more than one address from RAS, the VarPopRAS scheme reduces subsequent mispredictions after a function returns to its ancestor. Furthermore, leaving RAS unchanged on a mismatch might result in a correct prediction later, if a callee eventually returns to its caller.

Figure 6 compares the misprediction rates of RAS and VarPopRAS for ODB. As can be seen, VarPopRAS consistently outperforms RAS scheme. The VarPopRAS scheme reduces return mispredictions by more than 40% with a 512 entry RAS. VarPopRAS implementation, however, requires a 4-way associative search of top four RAS entries.

6.4 Indirect Branch Target Prediction

Indirect branch targets are predicted by using Indirect Branch Target Buffer (IBTB). A \( 2^n \) entry IBTB is indexed by using the lower n-bits of indirect branch IP. Each IBTB entry stores the target address of an indirect branch during its most recent execution. When an indirect branch is executed, the lower n-bits are used as an index into IBTB and the address stored in the corresponding IBTB entry is used as the predicted target address of that indirect branch.

![Figure 7](image-url)

Figure 7 shows the misprediction rate of indirect branch targets for various IBTB sizes. Indirect branch misprediction rate is significantly higher than the misprediction rate for other branch categories in all our benchmarks. Although indirect branches are less than 3% of the total branches in ODB and gcc, when conditional and return mispredictions are considerably reduced, indirect branches can become potential performance bottleneck. We plan to explore new indirect branch prediction schemes by analyzing the reasons for high misprediction rates.

7. Conclusions

This paper presents a detailed branch characterization of ODB; an Oracle based commercial OLTP workload, running on an IA32 processor. We use Simics, a full system simulator, to simulate a tuned ODB workload to collect a trace of one billion instructions. Using trace-driven analysis, we classify branch instructions into six categories. We compare branch behavior of ODB with that of gcc, gzip and mcf from the SPECINT 2000 benchmark suite. Our results show that the distribution of six branch
categories in ODB is similar to gcc. We deduce that, although code size of ODB is nearly an order of magnitude larger than gcc, control-flow constructs used in Oracle seem to be similar to those used in gcc.

By carefully exploring the design space of gshare and yags conditional branch predictors, we show that conditional branches in ODB are actually more predictable than in gcc, gzip and mcf. These results counter the popular belief that databases suffer from higher conditional branch mispredictions than some SPECINT 2000 benchmarks. We show that ODB branch prediction accuracy can be improved by using larger predictors that capture enough branch history information, and using branch prediction schemes that reduce aliasing. Our results also show that a hardware return address stack is ineffective when predicting return addresses for ODB. We analyze the reasons for return address mispredictions and propose an enhanced return address predictor to improve prediction accuracy for ODB.

Indirect branch misprediction rate is significantly higher than the misprediction rate of other branch categories in all our benchmarks. Although indirect branches are less than 3% of the total branches, due to their high misprediction rate they can become potential performance bottleneck. We plan to explore new indirect branch prediction schemes by analyzing the reasons for high misprediction rates. We are currently looking at ways to enhance Simics to model a cycle accurate system level simulator, which can estimate the impact of branch mispredictions on the overall execution time.

To date, most branch prediction studies focused on improving prediction accuracy of conditional branches. Based on results presented in this paper, for achieving high performance on OLTP workloads, we believe, future processor designers should pay more attention to return and indirect branch target prediction mechanisms, which are traditionally considered to have low performance impact.

8. Acknowledgements

We would like to thank Brian Hirano, Andreas Sundquist, and Ravi Thammaiah from the Oracle Corporation for graciously providing us with Oracle database binaries and for patiently explaining the complex configuration parameters for setting up the database workload properly. This work also benefited from many thought provoking discussions we had with Brian Hirano, Yong Fong Lee, and Hubert Nueckel.

9. REFERENCES


