

Comparing and Contrasting a Commercial OLTP Workload with CPU2000 on IPF

Jeff Rupley II, Murali Annavaram, John DeVale, Trung Diep, Bryan Black
Microprocessor Research, Intel Labs (MRL)

{jeffrey.p.rupley.ii,murali.m.annavaram,john.p.devale,trung.diep,bryan.black}@intel.com

Abstract

With the recent introduction of Itanium Processor Family (IPF) microprocessors for enterprise servers it is imperative to understand the behavior of server class applications. This paper analyzes the behavior of the Oracle Database Benchmark (ODB), an online transaction processing (OLTP) workload, and compares it with SPEC CPU2000.

This study examines code mix, instruction and data supply, and value locality. The results show that while IPF's bundle constraints cause a large injection of NOPs into the code stream, IPF's register stack engine successfully reduces the number of memory operations by nearly 50%. The control-flow predictability of ODB is better than CPU2000, in spite of ODB's large active branch footprint. Due to ODB's large memory footprint, cache misses (particularly instruction cache misses) are a much more serious problem than in CPU2000.

1. Introduction

While the SPEC CPU benchmarks are well known and have influenced the design of many microprocessors, the need to store and query large volumes of data makes database management systems (DBMSs) one of the most prominent applications that run on today's enterprise class servers. The introduction of Itanium Processor Family (IPF) microprocessors targeted at enterprise class server systems creates a need to understand the behavior of not only SPEC CPU2000, but also DBMS workloads. Significantly more work must be done on DBMS if the impact of these workloads are to become as ubiquitous as SPEC.

Recent work examines the performance impact of architectural features on DBMSs [6][8][14][17][1][3]. Most of these studies show that database applications have large instruction and data footprints and exhibit more unpredictable branch behavior than benchmarks that are more 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 [8]. Ailamaki *et al.* [1] analyzes three commercial DBMSs on a Xeon processor and show that TPC-D queries spend ~20% of execution time stalled on branch misprediction and ~20% waiting on L1 instruction cache miss. This work further investigates DBMSs at the microarchitecture level and compares them to SPEC.

Several previous studies [6][11][1] analyze memory system behavior of DBMSs workloads using hardware performance counters to gather runtime statistics. Two other investigations [20][15] use hardware counters to evaluate SPEC performance on Itanium microprocessors. Studies that utilize hardware counters can quickly analyze large-scale production systems and identify critical performance bottlenecks. However, these studies can only evaluate existing processor and system configurations, and thus have only limited capability to explore new design options. In this study an IPF simulator is used to examine the effects of varying processor configurations.

Since IPF is relatively new, very little published work has focused on IPF DBMS workloads. The goal of this paper is to study, in an IPF context, an Oracle based commercial OLTP workload (ODB) and contrast the findings with the better known SPEC CPU2000 benchmarks. This work focuses on benchmark analysis and fundamental architecture/microarchitecture features rather than specific microarchitecture features of existing IPF processors, such as Itanium or Itanium-2 [18].

The remainder of this work reviews the workload and simulation environment, and examines interesting program characteristics relating to the processor execution core, such as code mix and available ILP. The instruction supply is studied to characterize the fetch/branch control portions of a microprocessor, and the data supply is investigated because it is fundamental to the cache/memory hierarchy of a microprocessor. Finally, value locality is also examined.

2. Workload and Simulation Environment

This section describes the DBMS and SPEC workloads used in this study, as well as the microarchitecture simulation environment used to gather the desired results.

2.1. Oracle Database Benchmark

This study, continuing previous research[2], uses an Oracle 9i based OLTP workload, referred to as 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 a customer, checking the status of a previously placed order, and querying the system to check inventory levels at a warehouse.

All Oracle processes, server as well as background processes, share a large memory segment called the System Global Area (SGA). A large portion of the 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.

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, this study uses an in-memory ODB setup, where the working set fits in memory with negligible amount of disk I/O.

In order to achieve a balanced OLTP run, ODB requires tuning of three important setup parameters: SGA size, number of warehouses, and number of clients. Repeated execution of transactions after modifying each setup parameter and even building the database from scratch multiple times is required for successful tuning. In order to reduce the working set size, the SGA size is increased, the number of warehouses decreased, and the number of clients is reduced to the minimum required for optimal concurrency. Since these operations are very time consuming, ODB is first tuned and tested on a native IPF machine. ODB is compiled on an Itanium server running Red Hat Linux 7.2 using the Intel Electron compiler, with `-O1` and `-pgo` optimizations, and results in a binary that is approximately 100 MB. Compiler improvements are in progress and some results may change as a consequence.

For this study the ODB workload is configured to use 10 warehouses, which occupies 35 GB of disk space, and to use 2.5 GB of memory for the SGA, which is sufficient to cache frequently accessed database tables and metadata in main memory. Nevertheless, some disk I/O does occur, mostly large sequential writes from the redo log buffers to the redo log. After successfully tuning ODB, a complete and exact disk image of the resulting workload is created to facilitate execution by the Simics full system simulator, used to generate an execution trace.

After tuning of the workload is complete, a trace needs to be generated so that the workload can be evaluated through simulation. Collection of the execution trace is obtained by simulating ODB on Simics [13], a full system simulator.

Simics is a complete system-level simulator that is capable of booting several unmodified commercial operating systems and running unmodified application binaries. To generate a trace, Simics is configured to model an Itanium processor running Red Hat Linux 7.1. The ODB workload is simulated in the Linux environment on Simics for a sufficiently long time to warm up the main memory buffer cache in the simulator before checkpointing the simulator state. The checkpoint is used as a starting point to collect a 1 billion syllable trace of all instruction, data, and exception events.

2.2. SPEC

SPEC CPU2000 is the industry standard benchmark suite used to measure and compare compute intensive performance, and is intended to stress the performance of the processor, memory, and compiler [19]. The SPEC binaries used in this study are created with the Intel Electron compiler using peak optimizations. The train input set is used for all simulations. Two floating point benchmarks (facerec and galgel) are omitted from this study due to simulation problems. For brevity, the SPECint2000 and SPECfp2000 subdivisions of SPEC CPU2000 will be referred to as INT and FP for the remainder of the paper.

To avoid startup effects, simulation is advanced at least one billion syllables before entering cycle accurate simulation. Caches are warmed for the last 50 million syllables of the advance, and then each benchmark is simulated for one billion syllables.

2.3. Microarchitecture Simulation

Microarchitecture statistics are gathered using the IPFsim tool. IPFsim is a fully functional cycle accurate simulator of the IPF architecture. It is an Intel tool internally developed and designed for the evaluation of future IPF microprocessor implementations. IPFsim can simu-

late a wide variety of microprocessor microarchitectures, projecting performance, power, and area.

The IPFsim simulation environment can either execute Windows-NT binaries directly or read traces that contain instruction, data, and exception information. In this study all of SPEC is functionally executed in IPFsim, while ODB benchmark simulation is trace driven.

3. Program Characteristics

In order to understand the benchmarks and ISA it is important to enumerate some fundamental characteristics, such as instruction mix and instruction level parallelism (ILP). The instruction mix and ILP impact the size and functionality of a microprocessor’s execution core.

3.1. Code Mix

IPF is an EPIC architecture that uses instruction *bundles* to group operations called *syllables*. Bundles enable the EPIC architecture to explicitly define data dependencies and structural hazards for small groups of syllables, using simple stop-bits. Specifically, the IPF ISA [9] has 3 syllables per bundle. There are 5 primary syllable types **M** (Memory), **I** (Integer), **A** (ALU), **F** (Floating Point), and **B** (Branch). The **A** and **I** types appear to be redundant, but they are distinct in that **A** type syllables are a subset of integer operations that are allowed to execute in the memory execution unit. This is an interesting aspect of IPF that allows more flexible syllable dispatch in an in-order machine. For simplicity **A** and **I** syllable types are combined in this discussion.

Figure 1 illustrates the average syllable mix for FP, INT, and ODB. Detailed benchmark results for INT and FP are in Table 4 and Table 5 of the Appendix. Unexpectedly, NOPs represent between one-fifth and one-third of the code supply. This large NOP count is an artifact of the IPF bundle definitions and branch target alignment. In IPF there is a very limited decode space for

bundle definition. Consequently, not all possible combinations of syllable types and stop-bits can be in the architecture. When the compiler can not fit the code sequence into the defined bundle set it is forced to align syllables with NOPs. Another source of NOPs is the alignment of branch targets that must be at the beginning of a bundle.

Another very interesting observation is the very low percentage of memory syllables. Typically, one would expect memory operations to be 35%-40%. However, even adjusting for NOPs, the percentage of memory syllables is still just ~25%. The significant reduction of memory operations is primarily due to IPF’s register stack engine [9].

There is also unusual floating point activity in the ODB and integer benchmarks. This is partially caused by the implementation of integer multiply and divide as sequences of floating point instructions in IPF.

It is not surprising to find that for general syllable mix, ODB looks much more like the integer than the up benchmarks. However, it is interesting to note that ODB not only has a somewhat higher percentage of NOPs than INT on average, it also has a higher percentage of NOPs than any individual integer benchmark.

Table 1 Detailed ALU Syllables

	FP	INT	ODB
ALU (of total)	21%	48%	42%
ADD	69.3%	60.1%	62.1%
CMP	10.9%	21.5%	20.9%
SHLADD	7.8%	5.2%	2.3%
LOGICAL	1.6%	3.3%	3.1%
OTHER	10.4%	9.9%	11.7%

Table 1 further examines ALU syllables, decomposing a few specific operation types. The “OTHER” category includes miscellaneous operations such as shift, sign extend, and population count. ODB again matches

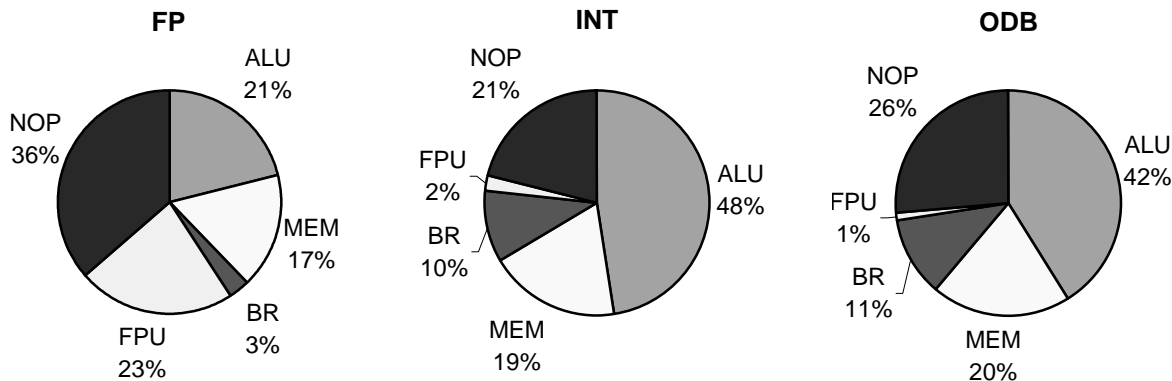


Figure 1 Dynamic Syllable Mix

average INT fairly well. Another observation that illustrates the differences of IPF from most architectures is that FP has few compares. This is because IPF has a loop branch that is aggressively used by the compiler in floating point code.

Table 2 analyzes the different classes of memory syllables. Another syllable unique to IPF is ld.s. Ld.s is a non-faulting speculative load that is used to hoist loads above branches. The “OTHER” category primarily consists of a variety of synchronization syllables such as compare-exchange.

Table 2 Detailed Memory Syllables

	FP	INT	ODB
Memory (of total)	17%	19%	20%
LD	12.8%	53.3%	58.6%
LD.S	2.8%	12.4%	7.9%
LDF	43.1%	3.8%	0.7%
LDF.S	0.8%	0.4%	0.0%
ST	4.6%	23.8%	30.7%
STF	19.5%	3.5%	1.5%
PREFETCH	16.5%	2.9%	0.2%
OTHER	0.0%	0.0%	0.5%

The floating point benchmarks contain many more software-controlled prefetches than ODB, even though ODB has a much bigger data footprint (as will be seen later in Section 5.). This is indicative of the much easier task the compiler has of software prefetching for the loop/matrix dominated floating point benchmarks than the types of data structures in ODB. Overall, ODB is again fairly similar to the INT average, although it does have a somewhat higher percentage of stores.

3.2. Useful Syllable Mix

Although it is typically true that the number of useful operations is very close to the total number of operations,

it is decidedly not the case for IPF. In IPF there are two sources of a large number of useless syllables: NOPs and predication [9][5]. The IPF architecture uses predication to eliminate difficult to predict branches. As a result both paths beyond a branch can be executed in the code stream, where the correct path is predicated true and the incorrect path is predicated false. Predicated false syllables are considered useless because they are not allowed to update machine state. As shown in Figure 1 ~20% of syllables are NOPs. Figure 2 shows that another ~4% of syllables are predicated false, therefore only 76% percent of the syllables in the instruction stream are useful. For the remainder of this work the term instruction is used as a subset of the syllables that do useful work, which includes branches, syllables that are not-predicated, and predicated-true syllables. It is interesting to note that floating point has a significant number of predicated true syllables, because of the unique software pipelining defined by IPF.

3.3. Instruction Level Parallelism

IPF specifies parallelism in the ISA using stop-bits to explicitly indicate structural hazards and data dependencies. The compiler also limits parallelism to 6, because it only attempts to find 6 independent syllables, due to the 2 bundle wide Itanium and Itanium-2 microprocessors. In an in-order machine the number of instructions between stop-bits is the maximum available ILP. Figure 3 shows the average number of independent syllables and instructions the compiler is able to find, expressed in syllables per stop-bit and instructions per stop-bit. The difference between syllables per stop-bit and instructions per stop-bit is the impact of useless syllables fetched. In a machine implementation this maximum possible ILP is also degraded by cache misses (see Section 5.) and branch mispredicts (see Section 4.2.).

The FP parallelism of 5 syllables/stop-bit is within 20% of the maximum ILP, which is expected given the large amount of ILP known to exist in floating point

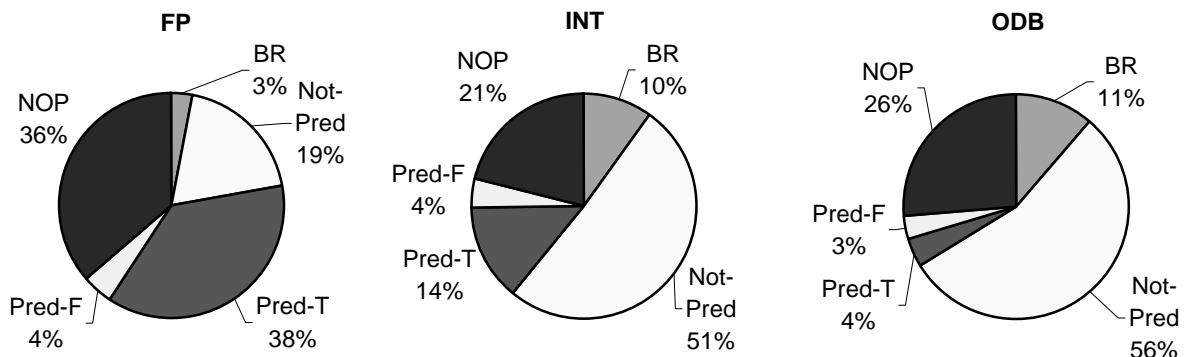


Figure 2 Useful Syllable Mix

benchmarks. However, the useful ILP extracted, expressed in instructions/stop-bit, is only 60% of the syllables/stop-bit. The difference is primarily due to a large number of NOPs, which is caused by the lack of appropriate templates, particularly those with F syllables.

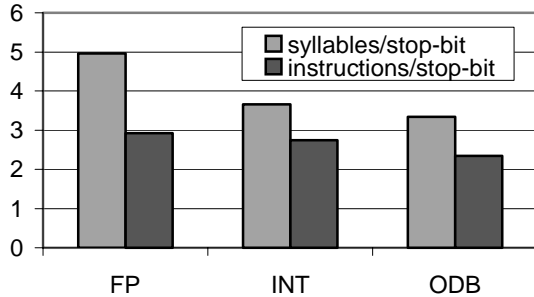


Figure 3 Parallel Instruction Groups

In general, the compiler-expressed useful ILP is not dramatically different between FP, INT, and ODB (less than a 25% difference). The compiler is able to extract ~5% less useful ILP for INT than FP, and ODB is another ~15% less than INT. This indicates that ODB has somewhat less compiler extractable ILP available than typical integer benchmark.

The primary conclusion that can be drawn here is that the similarities between ODB and INT are significant. A processor execution core that performs well on INT would also do well on ODB, if it were well supplied with instruction and data. However, as will be shown shortly, these conditions are difficult to meet.

4. Instruction Supply

This section examines the instruction supply problem, namely that of keeping the execution core busy. These studies examine basic block size and the branch predictability of SPEC and ODB. SPEC data throughout Section 4. is given as a harmonic mean.

4.1. Basic Block Size

Figure 4 describes the instruction supply per branch and per taken branch. As expected, FP basic blocks are much larger than INT or ODB due to the unrolled loops.

ODB behavior is slightly worse than average integer behavior, but both provide over 15 useful instructions per taken branch. This is very good for fetch and prediction mechanisms because it suggests there is potential for significant latency tolerance in the primary predictor loop, and complicated mechanisms like a trace cache can be avoided unless very wide machines are considered. Complete benchmark results are in Table 6 and Table 7 of the Appendix.

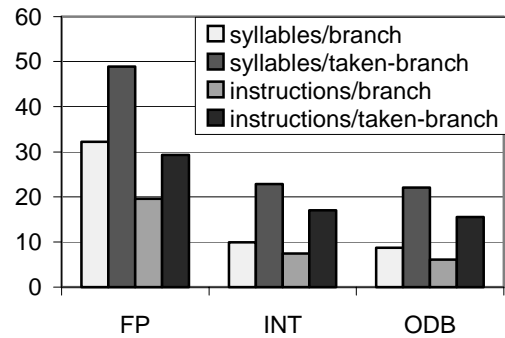


Figure 4 Instruction Supply Per Branch and Taken Branch

4.2. Branch Prediction

This section explores dynamic branch behavior. The goal is to find out how different predictors behave and illuminate benchmark activity. Results are reported in instructions per mispredict, which is a useful metric when considering fetch supply because it combines the instructions per branch metric with the mispredict ratio.

The first item to examine is the total number of unique taken branches, because this has implications for predictor capacity. Table 3 describes the static number of unique taken branches. ODB has a far larger number of branches than even the biggest SPEC benchmark (gcc).

Table 3 Unique Taken Branches

FP	INT	gcc	ODB
78	319	15,651	20,281

Figure 5 shows the number of instructions per mispredict for a bimodal [21] branch predictor as capacity is increased from 256 entries to 64K entries. These numbers assume a baseline predictor which handles procedure re-

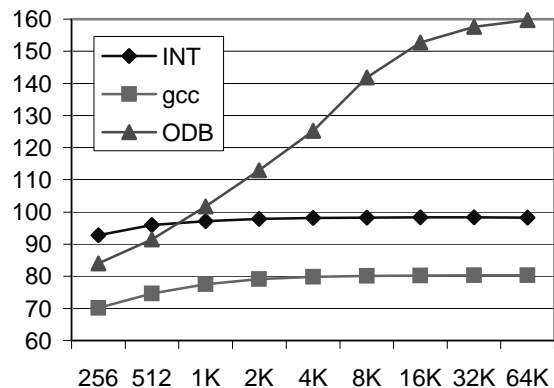


Figure 5 Instructions/mispredict as a function of bimodal predictor capacity

turns and loop based branches through mechanisms other than the primary branch predictor. Floating point predication rates are sufficiently high (99+% with up to 900 instructions per mispredict) that they are not of particular interest and thus are not shown.

For INT increasing capacity loses effect at 1K-2K entries, but much larger tables are very useful for ODB. This is a direct effect of the very large active branch footprint in ODB.

Many branch predictors more accurate than a bimodal predictor can be found. Figure 6 examines the bimodal predictor along with three more complicated predictors: gshare [16], local [23], and neural [10]. For simplicity all predictors are fixed at 64K entries. Similar to the x86 study in [2], when branch predictor capacity is large ODB instruction supply is much better than gcc for all predictor types.

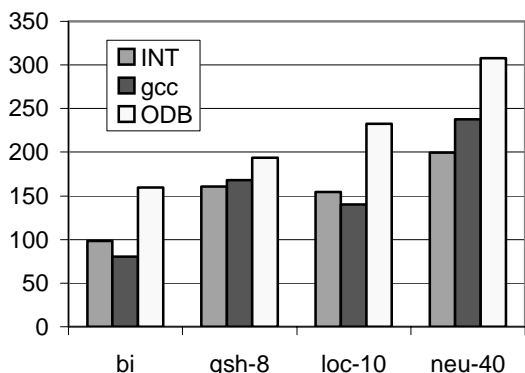


Figure 6 Instructions / Mispredicted Branch for Various 64K entry Predictors

Figure 7 examines the global history length of a 64K entry gshare predictor. The ODB instruction supply maximizes at a global history length of 8, while both gcc and average integer continue to improve with larger history up to 14 bits. This dramatic difference is due to ODB's large branch footprint, which creates more potential for prediction aliases. If the capacity of the predictor is in-

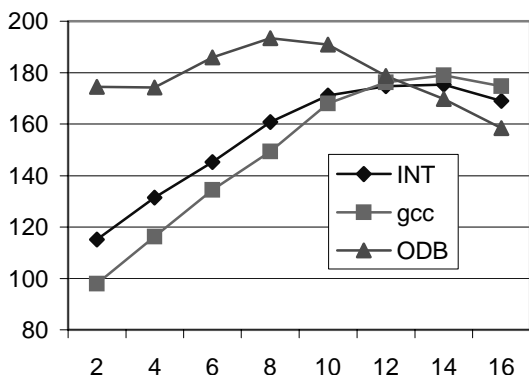


Figure 7 Instructions / Mispredicted Branch vs. History Length for Gshare Predictor

creased sufficiently, this problem should diminish and ODB can improve further. Further increases can also be found by combining these and other predictors using hybrid prediction mechanisms.

The primary conclusions that can be drawn from this instruction supply study is that while ODB requires a much larger capacity predictor than INT, the branches are equally easy or easier to predict. A prediction mechanism like the cascaded predictor [7] can accommodate a much larger (i.e. slower) capacity without impacting the latency of the primary predictor, and may be well suited for a general purpose enterprise microprocessor.

5. Data Supply

Data supply is another critical part of microprocessor design. A simplified two-level hierarchy is used to study the effects of cache size and associativity on the miss rates of these benchmarks. Results are reported in the number of misses per 1000 instructions (MPI).

5.1. L1 Cache Size

Figure 8 shows the effect of I and D-cache size on the MPI of the FP, INT, and ODB benchmarks. The results presented in this graph are from a 2-way set-associative cache with 64 byte cache lines. The 3 sets of bar graphs on the left show the I-cache MPI, and the right 3 sets of bar graphs show the D-cache MPI.

Several key observations can be made from the I-cache data. First, the MPI of ODB is nearly an order of magnitude larger than that of the floating point and integer benchmarks. Due to the very large instruction footprint of ODB, even with a 64KB I-cache, it suffers 37 MPI, while FP has 0.15 MPI and INT has 2 MPI. Further analysis reveals that the instruction working set of FP and INT is less than 32KB and ODB it is nearly 1M. This is also consistent with the branch footprint data seen previously in Table 3 of Section 4.2. Furthermore, doubling the cache size reduces the MPI of FP and INT by at least 33%, while ODB MPI decreases by only 15%.

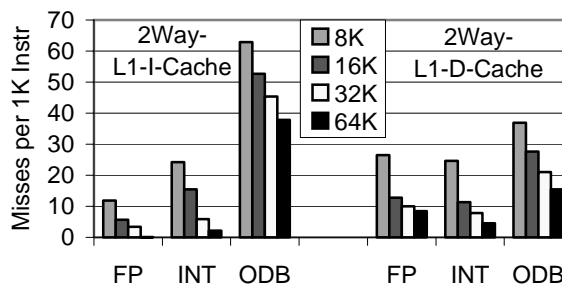


Figure 8 Effect of Cache Size on MPI

Two important observations are made from the D-cache MPI graphs. First, the D-cache MPI of ODB is higher than that of FP and INT, although the difference in MPI is not as significant as the difference in I-cache MPI. Also, a 64KB D-cache still has a significant number of misses in all benchmarks, indicating that the data working set of these benchmarks does not fit in these cache sizes. The MPI of ODB is twice as large as that of FP, and the MPI of FP is in turn twice as large as INT.

The conclusion to make from this data is that both primary caches (but especially the instruction cache) will be stressed much more heavily in ODB than in a machine targeted at SPEC cache working sets.

5.2. L1 Associativity

Figure 9 shows the effect of I and D-cache associativity on the MPI of FP, INT, and ODB benchmarks. All results presented in this graph are for a 16 KB cache with 64 byte cache lines.

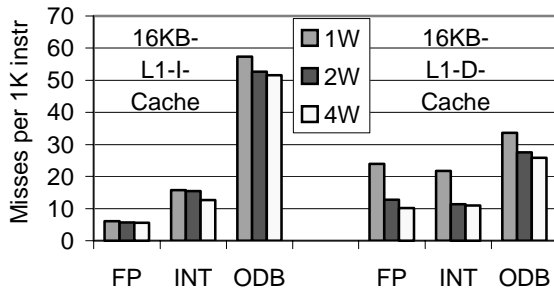


Figure 9 Effect of Associativity on MPI

Comparing results from Figure 8 and Figure 9, it can be seen that increasing cache size reduces I-cache misses, but increasing associativity marginally reduces I-cache misses. Since ODB is compiled using profile guided code placement optimization, the number of instruction cache conflict misses are reduced by mapping conflicting code blocks to different cache lines. As a result, more than 99% of the cache misses in ODB are due to capacity misses, consequently increasing associativity does not decrease MPI significantly.

Increasing the associativity from 1-way to 2-way reduces the D-cache MPI of FP and INT by nearly 47%. However, ODB MPI decreases by only 17%. Increasing the set-associativity of a given cache significantly reduces conflict misses. Comparing Figure 8 and Figure 9 suggest that while associativity is helpful to ODB, capacity is the primary concern.

5.3. L2 Cache Size

Figure 10 shows the MPI of the three benchmarks for different sizes of unified L2 caches. For these results, a

16KB, 4-way, 64 byte line L1 cache is used. The L2 cache is 16-way with 64 byte cache lines. While the INT and FP benchmarks in general benefit from larger cache sizes, some individual components working sets are mostly cached within the ranges shown. ODB suffers nearly an order of magnitude higher MPI than INT and even FP due to its large instruction and data footprint. For ODB even an 8MB cache has an MPI of 1.3 and 70% of the cache misses are capacity misses. (Note: As discussed in Section 2.1., the ODB benchmark is intentionally scaled to an in-memory setup. Hence, the data presented should not be taken as evidence that an 8MB cache is sufficient to fully cache the working set of a DBMS.)

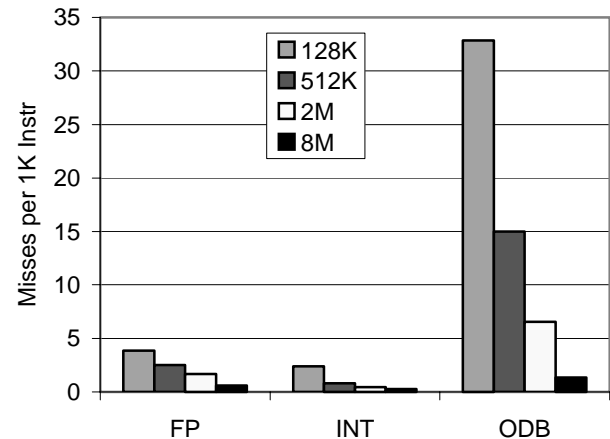


Figure 10 Effect of L2 Cache Size on MPI

Overall, a processor that is well designed to handle the caching for ODB will certainly also handle SPEC well, but the reverse is not true.

6. Load Value Locality

The predictability of loads and value prediction is introduced in [12] and presents interesting possibilities across architectures. It represents a way to reduce stalls due to memory latency in a manner that is complementary to the existing cache hierarchy. Of course, since the mispredict penalty is not insignificant, most implementations have complex prediction confidence mechanisms to select the correct value [22], as well as reduce the number of mispredicted values. Selective or focused value prediction [4] improves on confidence by concentrating value prediction on instructions that matter, using a notion of confidence or criticality. This study looks at the limits of predictability of a subset of instructions that are program bottlenecks via a very simple identification mechanism: cache misses.

Similar to [9], a 4K entry last value predictor table with a history of 1 and of 16 is used to explore value locality. Since this is a limit study, a perfect chooser is used

for the case of history=16, such that if any stored value is correct it is considered a correct prediction.

Locality is measured as a percent of correct predictions. The predictor is trained on every data access. Data is presented on the predictability of misses at each level of the cache hierarchy (using a 16KB L1D, 256KB L2, and a 3MB L3). Value Predictions are made only for load instructions.

Figure 11 shows the average and normalized average predictability for FP, INT, and ODB. Columns are grouped by the cache level at which the prediction is made, and include the predictability for a history=1 and history=16 (perfect selection). The normalized average represents the overall average weighted by the number of predictions performed per benchmark as a fraction of the total number of predictions of all benchmarks within a group.

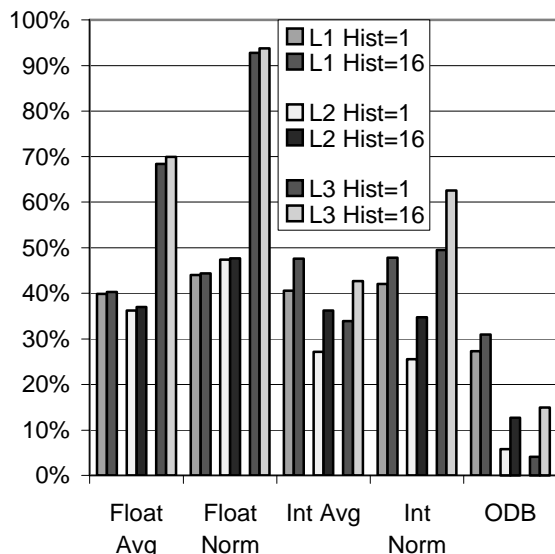


Figure 11 Load Value Predictability

This data clearly indicates that locality for SPEC and ODB, on IPF, varies greatly across benchmarks. Obviously, any value prediction mechanism for IPF will require significant constraints placed on it in order to prevent the mispredict penalty from dominating the performance of the mechanism. As shown in Table 9 of the Appendix, it is interesting to note that although some benchmarks have abysmal locality, they tend to also have fewer predictions allowing for a relatively high locality when normalized over the entire suite of benchmarks.

Although the locality of ODB for the history=1 (last value) predictor is low, the higher locality extracted by the perfect history=16 predictor implies that a predictor less trivial than simply last value would likely perform much better. Further improvements may also be found when considering the addition of stride detection into the

locality algorithm [22]. Finally, given the large number of memory transactions induced by TPC, even low locality may be worth leveraging.

7. Conclusions

This work examines the code mix, instruction supply, data supply, and value locality of an Oracle based OLTP workload and the SPEC benchmarks. Also a few lesser known observations are explored including the effect IPF has on program characteristics, such as the inflated NOP count, decreased memory operations, and large basic block size. Benchmark characteristics are compared and analyzed in an effort to understand how they each may impact microprocessor design. As expected there are many similarities between ODB and integer benchmarks, such as code mix and basic block size.

More interesting are the differences between these benchmarks. It is discovered that while INT and ODB appear to be similar, the ODB benchmark contains significantly more unique branches, yet they are more easily predicted even with a simple bimodal predictor at 2x the instructions per mispredict. The memory hierarchy is a major differentiator between SPEC and ODB. While floating point requires a larger capacity than integer, ODB miss rates dominate both. Finally, load value prediction of ODB appears to be even more challenging than SPEC.

It is clear that a diverse set of benchmarks is required for the evaluation of a general purpose enterprise microprocessor, and that SPEC and ODB are complimentary in many ways.

8. References

- [1] A. Ailamaki, D. DeWitt, M. Hill, and D. Wood, "DBMSs on a Modern Processor: Where Does Time Go?", In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 266-277, September 1999.
- [2] M. Annavaram, T. Diep, and J. Shen, "Branch Behavior of a Commercial OLTP Workload on Intel IA32 Processors", In *Proceedings of the International Conference on Computer Design*, pages 242-248, September 2002.
- [3] M. Annavaram, J. Patel, E. Davidson, "Call Graph Prefetching for Database Applications", In *Proceedings of the 7th Annual International Symposium on High Performance Computer Architecture*, pages 281-290, January 2001.
- [4] B. Calder, G. Reinmann, D. Tullsen, "Selective Value Prediction", In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 64-74, May 1999.
- [5] Y. Choi, A. Knies, L. Gerke, and T. Ngai, "The Impact of If-Conversion and Branch Prediction on Program Execution on the Intel Itanium Processor", In *Proceedings of the 34th Annual*

International Symposium on Microarchitecture, pages. 182-191, December 2001.

[6] Z. Cvetanovic and D. Bhandarkar, "Characterization of Alpha-Axp Performance using TP and SPEC Workloads", In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 60-70, April 1994.

[7] K. Driesen and U. Holzle, "The Cascaded Predictor: Economical and Adaptive Branch Target Prediction", In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 167-178, June 1998.

[8] M. Franklin, W. Alexander, R. Jauhari, A. Maynard, B. Olszewski, "Commercial Workload Performance in the IBM Power2 RISC System/6000 processor", In *IBM Journal of Research and Development Vol. 38 No. 5*, pages 555-561, 1994.

[9] Intel Corporation. Intel Itanium Architecture Software Developer's Manual. Available from <http://developer.intel.com/design/Itanium2/manuals/>.

[10] D. Jimenez and C. Lin, "Dynamic Branch Prediction with Perceptrons", In *Proceedings of the 7th Annual International Symposium on High Performance Computer Architecture*, pages 197-206, January 2001.

[11] K. Keeton, D. Patterson, Y. He, R. Raphael, and W. Baker, "Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads", In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 15-26, June 1998.

[12] M. Lipasti, C. Wilkerson, and J. Shen, "Value Locality and Load Value Prediction", In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138-147, October 1996.

[13] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform", In *Computer Vol: 35 Issue: 2*, pages 50-58, February 2002.

[14] A. Maynard, C. Donnelly, B. Olszewski, "Contrasting Characteristics and Cache Performance of Technical and

Multi-user Commercial Workloads", In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145-156, October 1994.

[15] J. McCormick and A. Knies, "A Brief Analysis of the SPEC CPU2000 Benchmarks on the Intel Itanium2 Processor", *HotChips 14*, August 2002.

[16] S. McFarling, "Combining Branch Predictors", *WRL Technical Note TN-36*, Digital Equipment Corporation, June 1993.

[17] M. Rosenblum, E. Bugnion, S. Herrod, E. Witchel, and A. Gupta, "The Impact of Architectural Trends on Operating System Performance", In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 285-298, December 1995.

[18] D. Soltis, M. Gibson, and C. McNairy, "Itanium2 Processor Microarchitecture Overview", *HotChips 14*, August 2002.

[19] SPEC, "SPEC CPU2000", <http://www.spec.org/osg/cpu2000/>, January 2000.

[20] M. Serrano and Y. Wu, "Memory Performance Analysis of SPEC2000C for the Intel Itanium Processor", In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[21] J. Smith, "A study of branch prediction strategies", In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 135-148, May 1981.

[22] K. Wang and M. Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors", In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages. 281-290, December 1997.

[23] T. Yeh and Y. Patt, "Alternative implementations of two-level adaptive branch prediction", In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124-134, May 1992.

9. Appendix

Table 4 Instruction Mix for INT

	bzip2	crafty	eon	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr	mean
ALU	55%	59%	35%	44%	50%	55%	49%	46%	40%	48%	48%	40%	48%
MEM	20%	14%	23%	20%	16%	18%	23%	18%	21%	18%	21%	16%	19%
BR	5%	8%	8%	12%	13%	9%	8%	12%	14%	8%	16%	9%	10%
FPU	0%	1%	10%	1%	0%	0%	0%	1%	0%	5%	1%	7%	2%
NOP	20%	17%	25%	23%	20%	18%	19%	24%	24%	21%	15%	28%	21%

Table 5 Instruction Mix for FP

	ampp	applu	apsi	art	equake	fma3d	lucas	mesa	mgrid	sixtrack	swim	wupwise	mean
ALU	24%	11%	17%	13%	21%	18%	29%	32%	12%	35%	16%	23%	21%
MEM	13%	13%	21%	29%	29%	18%	11%	13%	14%	13%	19%	8%	17%
BR	4%	0%	3%	6%	2%	1%	1%	7%	1%	9%	1%	3%	3%
FPU	20%	30%	23%	16%	21%	23%	27%	21%	30%	12%	28%	22%	23%
NOP	39%	45%	36%	36%	27%	41%	31%	28%	44%	30%	36%	44%	36%

Table 6 Instruction Supply per Branch and Taken Branch for INT

	bzip2	crafty	eon	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr	mean
SPB	20.36	12.45	12.70	8.58	7.51	11.23	11.90	8.34	7.24	12.98	6.44	10.96	9.90
SPTB	23.43	37.70	22.18	27.52	15.70	20.23	19.15	18.20	18.59	20.58	38.86	37.79	22.84
IPB	15.92	9.25	9.44	6.55	5.78	8.88	8.59	6.20	5.37	8.73	5.40	7.31	7.43
IPTB	18.32	28.00	16.49	21.02	12.10	15.99	13.82	13.53	13.78	13.84	32.55	25.23	17.02

Table 7 Instruction Supply per Branch and Taken Branch for FP

	ampp	applu	apsi	art	equake	fma3d	lucas	mesa	mgrid	sixtrack	swim	wupwise	mean
SPB	26.24	248.59	37.73	17.58	53.02	84.95	101.15	14.93	140.24	11.22	78.30	33.41	32.26
SPTB	32.44	276.74	43.54	18.89	61.76	171.33	108.66	37.46	144.92	23.28	79.14	67.15	48.85
IPB	14.88	133.93	19.91	10.86	34.20	49.61	59.14	10.54	75.69	6.95	49.89	17.43	19.58
IPTB	18.39	149.10	22.98	11.67	39.83	100.04	63.54	26.45	78.21	14.41	50.42	35.03	29.30

Table 8 Useful Instructions/Mispredicted Branch for Predictor Size for INT

bimodal predictor	bzip2	crafty	eon	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr	Hmean
256 entry	418.66	76.83	188.14	106.90	70.12	89.25	57.96	57.38	74.52	108.89	347.36	73.67	92.83
512 entry	418.73	80.03	238.83	110.01	74.70	89.22	58.01	57.98	77.32	117.45	376.47	73.20	95.99
1024 entry	418.73	81.05	273.18	110.06	77.55	89.22	58.01	58.34	77.45	119.02	398.88	72.70	97.14
2048 entry	418.73	82.12	275.17	113.41	79.10	89.22	58.01	58.36	78.16	119.37	405.80	72.70	97.85
4096 entry	418.73	82.97	280.08	113.45	79.82	89.22	58.01	58.38	78.57	119.62	407.91	72.70	98.18
8192 entry	418.73	83.56	280.08	113.45	80.13	89.22	58.01	58.37	78.57	119.62	409.18	72.70	98.29
16384 entry	418.73	83.99	280.08	113.45	80.28	89.22	58.01	58.38	78.57	119.62	410.83	72.70	98.37
32768 entry	418.73	83.99	280.08	113.45	80.30	89.22	58.01	58.38	78.57	119.62	410.88	72.70	98.37
65536 entry	418.73	83.99	280.08	112.66	80.32	89.22	58.01	58.38	78.57	119.62	410.94	72.70	98.33
unique t-br	137	1719	528	928	15651	256	81	2595	207	1933	325	602	319

Table 9 Locality of load values

	L1			L2			L3		
	Hist=1	Hist=16	# pred	Hist=1	Hist=16	# pred	Hist=1	Hist=16	# pred
ampp	18.3%	18.5%	3.9E+7	1.5%	2.0%	1.2E+7	4.3%	8.3%	2.3E+4
applu	31.6%	33.8%	3.2E+7	27.8%	31.1%	9.5E+6	72.6%	84.5%	1.3E+6
apsi	53.5%	53.5%	6.5E+7	44.8%	44.9%	7.0E+6	75.0%	75.0%	2.6E+3
art	64.7%	64.7%	1.5E+8	54.2%	54.3%	1.5E+8	0.2%	0.2%	1.3E+3
equake	57.0%	57.8%	9.2E+7	40.9%	41.4%	3.8E+7	99.9%	99.9%	4.9E+6
fma3d	23.9%	24.3%	9.1E+7	26.2%	28.7%	5.4E+6	80.0%	80.9%	6.6E+2
lucas	34.8%	34.8%	5.4E+7	61.6%	61.8%	9.0E+5	91.3%	91.3%	3.7E+3
mesa	47.3%	47.3%	5.7E+7	17.5%	17.5%	3.8E+6	80.3%	80.3%	5.7E+5
mgrid	47.6%	47.6%	4.0E+7	51.9%	52.0%	1.2E+7	94.3%	94.5%	2.6E+6
sixtrack	43.6%	43.8%	3.0E+7	17.5%	19.6%	6.8E+5	38.6%	39.8%	5.4E+4
swim	32.3%	33.3%	7.9E+7	53.1%	53.4%	2.3E+7	93.2%	93.5%	7.8E+6
wupwise	24.0%	24.0%	3.3E+7	37.4%	37.4%	2.9E+6	90.3%	90.3%	1.2E+6
bzip2	89.9%	90.2%	6.2E+7	50.1%	52.6%	4.3E+6	54.2%	55.6%	2.9E+5
crafty	46.7%	51.2%	1.4E+8	17.8%	45.3%	2.6E+6	17.1%	52.0%	1.3E+6
eon	36.9%	43.3%	2.5E+7	22.7%	31.1%	1.8E+5	22.1%	33.7%	8.6E+1
gap	42.2%	44.7%	5.6E+7	19.5%	23.5%	4.6E+6	60.5%	62.7%	1.4E+6
gcc2	44.2%	45.4%	1.2E+8	38.5%	49.6%	2.8E+6	58.1%	65.1%	2.0E+5
gzip	33.1%	38.0%	3.6E+7	4.5%	6.9%	5.2E+5	2.1%	3.0%	7.3E+4
parser	46.1%	49.3%	7.3E+7	28.3%	32.0%	8.8E+6	49.3%	52.6%	8.2E+5
perlbnk	15.8%	44.8%	2.9E+7	41.2%	55.7%	9.6E+3	8.3%	18.6%	3.5E+2
mcf	32.6%	41.0%	1.0E+8	28.3%	37.5%	4.7E+7	87.5%	92.6%	2.9E+6
vortex2k	41.4%	45.7%	5.7E+7	43.0%	51.2%	1.2E+6	15.1%	21.9%	1.5E+5
vpr	33.4%	41.8%	4.6E+7	17.8%	22.9%	2.1E+6	5.7%	7.8%	5.4E+3
twolf	24.9%	35.4%	8.9E+7	14.5%	26.3%	2.2E+7	26.4%	46.8%	3.3E+6
ODB	27.3%	30.9%	1.1E+8	5.7%	12.7%	1.6E+7	4.1%	15.0%	3.2E+6