3 PLATFORM, TOOLS, AND BENCHMARKS

This chapter describes the different benchmarks we used to evaluate our proposed optimizations, as well as the simulation and performance measuring tools used across the work.

We have used three different benchmarks to evaluate our proposals: the SPEC integer benchmarks, and two different database workloads, a decision support system (DSS), and an on-line transaction processing system (OLTP). This section describes each setup in detail, including the special characteristics of each set.

The use of code layout optimizations implies the use of profile information, and the generation of two sets of executables: the unoptimized set, and the optimized set. This chapter also describes the tools used to obtain the profile data, and the process followed to obtain the optimized code sets.

Finally, this chapter includes a detailed description of the different simulation environments used and those we developed, the parameters explored, and the metrics obtained. We also specify which simulators were used for each benchmark, as some had special requirements. In this section, we also include a description of the real machine run measurements we performed.

3.1 Benchmarks

This section provides a brief description of the different benchmark sets used in this work. We have tried to use a wide variety of workloads, to exercise every possible situation in which our proposals may be useful. In particular, we were very interested in commercial workloads. Database application represent the most important market segment for high-performance servers, and their importance is constantly growing.

In addition to commercial applications such as decision support systems (DSS) and on-line transaction processing (OLTP), we tested our proposals using a wide variety of applications, such as those represented by the SPEC integer benchmarks. We did not use any floating-point intensive

application in our study because the performance bottleneck for these workloads is not in the fetch engine.

3.1.1 SPEC int 95

The SPEC'95 benchmark suite [16], released on August 1995[1] by the Standard Performance Evaluation Corp. (SPEC) represents the latest version of the worldwide standard for measuring and comparing computer performance across different hardware platforms. SPEC'95 was developed by SPEC's Open Systems Group (OSG), which includes more than 30 leading computer vendors, systems integrators, publishers and consultants throughout the world.

SPEC'95 comprises two sets (or suites) of benchmarks: CINT95 for compute-intensive integer performance and CFP95 for compute-intensive floating point performance. The two suites provide component-level benchmarks that measure the performance of the computer's processor, memory architecture and compiler. SPEC benchmarks are selected from existing application and benchmark source code running across multiple platforms. Each benchmark is tested on different platforms to obtain fair performance results across competing hardware and software systems.

SPEC'95 is the third major version of the SPEC benchmark suites, which in 1989 became the first widely accepted standard for comparing compute intensive performance across various architectures. The new release replaced SPEC'92. Performance results from SPEC'95 cannot be compared to those from SPEC'92, since new benchmarks have been added and existing ones changed.

Improvements to the suites include longer run times and larger problems for benchmarks, more application diversity, greater ease of use, and standard development platforms that will allow SPEC to produce additional releases for other operating systems.

go	Board game simulator
m88ksim	Motorola 88K simulator
gcc	C/C++ compiler
compress	Lempel-Zip encoder/decoder
li	Lisp interpreter
ijpeg	JPEG compressor/decompressor
perl	The Perl language interpreter
vortex	Simple database application

Table 3.1. Description of the SPEC'95 integer benchmarks.

Table 3.1 shows a brief description of the SPEC'95 integer programs. We did not use *go* because our profiling tool (*pixie*) could not obtain profile data for it, due to some unexplained program errors. The lack of profile data prevented us from generating an optimized code layout, which excluded *go* from our benchmark set. We also excluded *compress* due to the small size of its code. It fits completely in a very small instruction cache, and concentrates most of its execution is a small loop with very few branches, which leaves no room for code layout optimizations to have any effect.

We generated the baseline code for our work using Compaq's C compiler, using -O2 optimization level. The simulation methodology has varied across the work, depending on the speed of the

[1]The SPEC'95 benchmark suite was declared obsolete as of July 2000, replaced by the SPEC 2000 suite.

simulator we used for each set of results. It is discussed in detail in Section 3.3, where we describe the different simulators we used.

3.1.2 PostgreSQL and TPC-D

PostgreSQL is a public domain, client/server database developed at the University of California-Berkeley, very popular among the Linux community, with a large number of users. PostgreSQL has a client/server structure and compromises three types of processes; clients, backends and a postmaster. Each client communicates with one backend that is created by the postmaster the first time the client queries the database. The postmaster is also in charge of the initialization of the database.

We use PostgreSQL as the execution engine for the Transaction Processing Performance Council benchmark for Decision Support Systems (TPC-D) [90]. DSS workloads imply large data sets and complex, read-only queries that access a significant portion of this data.

TPC-D has been described in the recent literature on the topic [5, 91] and for this reason we do not give a detailed account of it. At a glance, the TPC-D benchmark defines a database consisting of 8 tables, and a set of 17 read-only queries and 2 update queries. It is worth noting that the TPC-D benchmark is just a data set and the queries on this data; it is not an executable. The tables in the database are generated randomly, and their size is determined by a Scale Factor. The benchmark specification defines the database scale factor of 1 corresponding to a 1GB database. There are no restrictions regarding the indexes that can be used for the database.

We set up the Alpha version of the Postgres 6.3.2 database compiled with the $-O3$ optimization flags and Compaq's compiler. A TPC-D database is generated with a scale factor of 0.1 (100MB of data). With the generated data we build two separate databases, one having Btree indexes and the other having Hash indexes. Both databases have unique indexes on all tables for the primary key attributes (those attributes that identify a tuple) and multiple entry indexes for the foreign key attributes (those attributes which reference tuples on other tables). More indexes could be created to accelerate the query execution, as the standard does not restrict those, but we were more interested in the database behavior than in the performance of the benchmark.

Query	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Training Set	-	-	X	X	X	X	-	-	X	-	-	-	-	-	X	-	-
Test Set	-	X	X	X	-	X	-	-	-	-	X	X	X	X	X	-	X

Table 3.2. TPC-D queries used to obtain the profile information (training) and to obtain performance results (test).

We run the TPC-D queries on the PostgreSQL database management system assuming the data is disk resident, not memory resident. We do not perform any warm up to load the database in memory, so TPC-D data must be loaded for every new query. The set of queries used to obtain the profile information and the ones used to evaluate the performance of our method are shown in Table 3.2. The Training set is executed on the Btree indexed database only, and the Test set is executed on both the Btree and the Hash indexed databases.

3.1.3 Oracle and TPC-B

Our OLTP workload runs on top of the Oracle 7.3.2 DBMS. The Oracle 7.3.2 DBMS runs on both uniprocessor and multiprocessor shared memory machines, and recent benchmark results demonstrate that the software scales well on current SMP machines [17].

Figure 3.1 illustrates the different components of Oracle. The server executes as a collection of Unix processes that share a common large shared memory segment, called the System Global Area (SGA). Oracle has two types of processes: daemons, and servers. The daemons run in the background and handle miscellaneous housekeeping tasks such as checking for failures and deadlocks, evicting dirty database blocks to disk, and writing redo logs. The server processes are the ones that actually execute database transactions and account for most of the processing. Both daemons and server processes share the same code, but have their own private data segment called the Program Global area (PGA). Daemons and servers primarily use the SGA for communication and synchronization, but also use signals to wake up blocked processes.

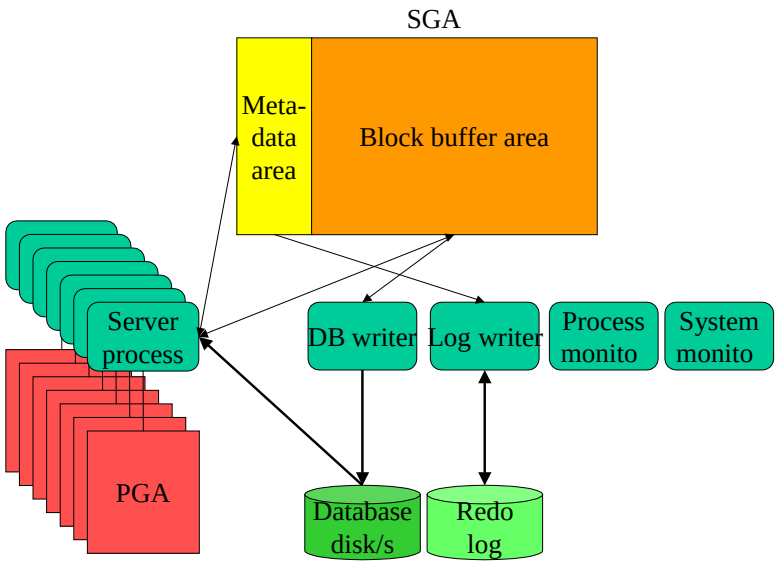

Figure 3.1. Major components of the Oracle 7.3.2 server.

The SGA is roughly composed of two regions, namely the block buffer and meta-data areas. The block buffer is used for caching the most recently used database disk blocks in main memory, and typically accounts for 80% of the SGA. The meta-data area contains directory information used to access the block buffers, and space for miscellaneous buffers and synchronization data structures.

As other database engines, Oracle maintains two important types of persistent data structures: the database tables and the redo log. The latter keeps a compressed log of committed transactions, and is used to restore the state of the database in case of failure. Committing only the log to disk (instead of the actual data) allows for faster transaction commits and for a more efficient use of disk bandwidth.

Our OLTP workload is modeled after the TPC-B benchmark [89]. This benchmark models a banking system, with each transaction simulating a balance update to a randomly chosen account. the account balance update involves updates to four tables: the branch table, the teller table, the

account table itself, and the history table. Each transaction has fairly small computation needs, but the workload is I/O intensive because of the four table updates per transaction. Fortunately, even in a full-size TPC-B database (a fer hundred GB), the history, branch, and teller tables can be cached in physical memory, and frequent disk accesses are limited to the account table and redo log.

We use Oracle in a dedicated mode for this workload, where each client process has a dedicated server process for serving its transactions. The communication between client and server occurs through a Unix pipe. Each transaction consists of five queries, followed by a commit request. To guarantee the durability of a committed transaction, commit requests block the server until the redo log has been written. To hide the I/O latencies, including the latency of log writes, OLTP runs are usually configured with multiple server processes per processor. We have used 8 server processes per processor.

Although TPC-C has superseded TPC-B as the official OLTP benchmark [88], we have chosen to model TPC-B because it is simpler to set up and run, and yet exhibits similar behavior to TPC-C as far as processor and memory are concerned [86].

3.2 Optimized code generation

In most of our work we use two different sets of applications: the baseline set, and the code layout optimized set. Generating a layout optimized version of an application is done in two steps. First, it is necessary to obtain adequate profile information to process the code and generate the desired basic block ordering information. Then we can use a binary optimizer to read the baseline application, and apply the transformations required to generate the code layout desired. This section describes both the process we followed to generate our optimized benchmark set, and the tools we used to do it.

3.2.1 Profiling tools

During this work we have used two different code layout optimizers: the software trace cache (described in Chapter 4), and SPIKE [13]. Each requires a different set of profile data to process the original binary.

The software trace cache requires the profile data to have an exact count of how many times each basic block transition was followed. That is, for each pair of basic blocks A and B, the number of times that block B was executed after block A. From this edge count we can easily derive the number of times each basic block was executed.

We obtain this profile data using ATOM [83]. We instrument the baseline application so that each time a basic block is executed, a profiling routine is called with the basic block identifier as argument. The routine keeps track of which was the previously executed block, and maintains a structure with all edges seen to that point.

On the other hand, SPIKE requires a basic block execution count profile in the *pixie* format. Pixie is one of the ATOM tools, and basically counts how many times a basic block was executed. As an alternate source for profile information, it is possible to feed SPIKE with profile data obtained with DCPI [1].

DCPI is a continuous profiling infrastructure which monitors the system at random intervals, and builds its profile database using a sampling of the actual execution. It allows the gathering of

profile information in a non-intrusive way, as it does not require any modification of the application to analyze, and has a very low execution overhead. However, the sampling nature of the profile obtained, makes it inaccurate if the profiling period was not long enough. We experimented with both profile sets (pixie and DCPI), and used the one which provided the best results. In our case, that was the pixie profile.

3.2.2 Code optimizers

Figure 3.2 shows the process we used to generate the optimized application codes. The first optimizer we used implements the software trace cache algorithm, as described in Chapter 4. It reads the profile data generated by ATOM (see previous section), and generates a file containing a PC translation table.

It is important to note that we did not actually generate an optimized code for the software trace cache. The result of running the STC algorithm is an address translation table, which maps each instruction in the original application to its new location, as calculated by the algorithm. We then use this PC translation table to feed the simulator with translated addresses. The use of a PC translation table does not account for code transformations that may be required in the actual executable, such as branches that should be added or removed. However, Pettis and Hansen report that these instructions represent only 2–3% of all the executed instructions [59]. We also used the same technique to generate optimized code layouts for the Pettis & Hansen algorithm [59], and the Torrellas *et al.* [87] algorithm.

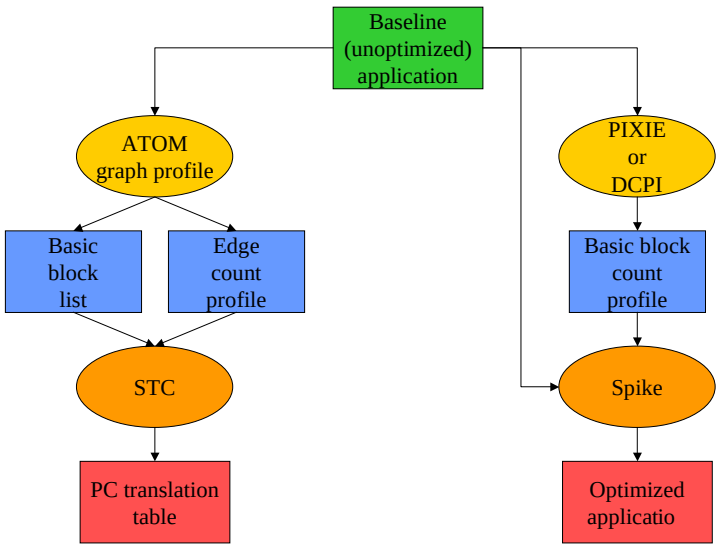

Figure 3.2. Profiling of the baseline application and generation of the optimized code layouts.

The second optimizer used is a variation of Compaq's SPIKE [13], extended to include the fine-grain subroutine splitting and the CFA optimizations used in the STC. The output of SPIKE is a valid executable application which can be run directly on the simulator, or even on a real machine. The main difference between the STC optimizations and SPIKE is that SPIKE does not inline subroutines (or chains), which reduces the spatial locality, and introduces additional control transfers (subroutine calls and returns) that would have been avoided in the STC.

The results shown in Figure 4.4, Sections 4.2.2 and 4.2.3, and Chapters 5 and 6 were generated using the STC layout. The remaining results, in particular all Oracle/TPC-B results in Chapter 4, were obtained with applications optimized with Spike.

3.3 Simulators

3.3.1 Fetch engine simulation

The first stages of this work were evaluated using a fetch engine simulator which implements the trace cache mechanism as described in Figure 2.16. We simulate in detail the branch prediction mechanism (return address stack (RAS), muti-ported BTB, and multiple branch predictor), the instruction cache, and the trace cache. We use an adapted version of the gshare branch predictor as a multiple branch predictor. We augment the PHT so that it contains three separate 2-bit counters instead of one. Each counter is used as a separate branch prediction.

We read two consecutive lines from the instruction cache, to ensure that a full width of instructions can be provided. If any of the two lines is not present in the L1 cache, the fetch stage stalls until both lines are available.

The branch prediction mechanism uses the fetch address to provide 16 consecutive entries from the BTB, which allow the engine to identify branch instructions. The RAS is used to predict the target address of return instructions, and the multiple branch predictions are used for the first three conditional branches encountered. Using all this information we generate in instruction mask which selects only the instructions until the first taken branch from the instruction cache.

The trace cache is accessed using only the fetch address. If an appropriate trace is found, it is still necessary to check if it matches the path predicted by the multiple branch predictor. If it matches, we provide the instructions stored in the trace cache, and use its prediction as the next fetch address. If the trace is not present, or does not match the predicted path, fetch proceeds from the instruction cache, and the next fetch address is taken from the branch prediction mechanism.

Table 3.3 shows the default setup for the fetch engine simulator. Unless otherwise stated, these are the values used in all results using this simulator.

Instruction cache	32KB, 64B lines, direct mapped 64KB, 64B lines, 2-way set associative Fetch 2 lines per cycle 6-cycle miss penalty
Trace cache	32 to 2048 entries, 2-way set associative
gshare	Adapted to 3 predictions per cycle 12 bit history, 4K-entry PHT 12-cycle misprediction penalty
BTB	512 entries, 2-way set associative 12-cycle misprediction penalty
RAS	Perfect

Table 3.3. Default simulator setup for the isolated fetch engine simulator.

In addition to measuring individual component statistics, such as instruction cache hit rate, trace cache hit rate, or branch prediction accuracy, this simulator generated two different fetch performance metrics:

Fetched instructions per access (FIPA): Measures the effective fetch width of the fetch engine. The FIPA is the average number of correct path instructions that the fetch engine provides each time it is accessed, regardless of how many cycles it takes to fetch the instructions. This metric does not account for cycles wasted fetching down a wrong speculative path, not stall cycles due to instruction cache misses, it is a raw fetch width metric.

Fetched instructions per cycle (FIPC): Measures the isolated fetch performance of the engine. The FIPC is the average number of correct path instructions that the fetch engine provides each cycle. Unless otherwise stated, we assume that the engine stall for 6 cycles on each L1 instruction cache miss, and that 12 cycles are wasted fetching from a wrong path for each branch misprediction.

This simulator was used to generate the results in Figures 4.4 and 4.12, and Chapter 5.

3.3.2 The Simplescalar toolset

In some parts of our work we have used the SimpleScalar 3.0a tool set [7]. SimpleScalar is advertised to perform fast, flexible, and accurate simulation of processors implementing a MIPS-like architecture. The tool set takes an executable file, and simulates its execution on one of several processor simulators.

The tool set includes simulators to simply test the correct execution of the program, simulate the branch prediction mechanism, the memory hierarchy, or a fully detailed out of order superscalar processor. The simulation speed depends on the detail level of the simulation, with the functional simulator being the fastest, and the out of order model the slowest.

The out of order model simulates a 6 stage pipeline with fetch, decode/rename, issue/schedule, execute, writeback, and commit stages using a centralized structure (the reservation stations) to guarantee the correct execution semantics. The simulator is execution-driven (as opposed to trace-driven), and is able to accurately simulate instructions on a wrong speculative path.

The default processor setup we used resembles the configuration of the Alpha 21264 processor, and is summarized in Table 3.4.

L1 cache	64KB, 32B lines, 2-way associative
L2 cache	1MB, 32B lines, 4-way associative 15-cycle latency
Memory	120-cycle latency
BTB	4K-entry, 4-way associative
gshare	4K-entry, 12-bit history
RAS	64-entry
width	4-wide fetch, issue, commit

Table 3.4. Default setup for our SimpleScalar simulator.

The simulator provides very detailed metrics on most structures found in the simulated superscalar processor, but the metric we are most interested in is the overall processor performance measured in instructions per cycle (IPC).

We used the simplescalar out of order model to generate the results shown in Figure 4.22.

3.3.3 Branch predictor simulation

In addition to a complete fetch engine simulator, we required a fast and detailed branch predictor simulator to evaluate the impact of code layout optimizations and the usage of profile data on the branch prediction mechanism.

The branch predictor simulator we used is derived from the SimpleScalar tool set [7], which we extended to implement not only two-level adaptive branch predictors but also static branch prediction strategies, dealiased branch predictors, and to incorporate the profile data that was obtained to generate the optimized code layouts.

Table 3.5 shows the default setup for the branch predictor simulator. Unless otherwise stated, these are the values used in all results using this simulator.

BHR	Global register, 10–16 bits
PHT	$2^{BHRsize}$ entries 3 tables for gskew
BTB	4K-entry, 4-way associative
Bias table	uses the BTB (agree only)
Selector	$1/3$ of the total size (bimode only)
RAS	64-entry

Table 3.5. Default simulator setup for the branch predictor simulator.

In addition to the branch prediction accuracy, this predictor was extended to classify all executed branches by their actual behavior, to measure prediction table interference, and classify the conflicts among positive (the conflict causes a correct prediction), neutral (the conflict does not cause the prediction to change, but may cause a non-saturated counter to saturate), and negative (the conflict causes a misprediction), and to provide statistics about the distribution of data on the second level table.

All results in Section 4.2.3 and Chapter 6 where generated using this simulator.

3.3.4 SimOS

In order to model a multiprocessor OLTP workload we used SimOS [73]. SimOS is a full system simulation environment initially developed at Stanford University to study MIPS-based multiprocessors. We used the Alpha port of SimOS developed at Compaq's Western Research Lab, which models the hardware components of Alpha-based multiprocessors (processors, MMU, caches, disks, console) in enough detail to run unmodified Compaq's system software and the Oracle DBMS.

SimOS supports multiple levels of simulation detail, and allows the user to choose the most appropriate trade-off between simulation speed and detail. The fastest simulation mode uses on-the-fly binary translation to position the workload in a steady state. at that point, the simulator takes a state checkpoint, and the it is possible to switch to a more detailed (and slow) simulation mode.

The ability of SimOS to run both application and system code is essential to simulate all the rich interactions between operating system and application code in OLTP workloads.

Our simulations run from a checkpoint that is taken when the workload is already in its steady state, and run for 500 transactions (after a warm-up period) on a simulated 4-processor Alpha

system. The basic SimOS-Alpha simulations use a 1 GHz single-issue pipelined processor model with 64KB 2-way instruction and data caches (64-byte line size), and a 1.5MB 6-way unified L2 cache. The memory latencies assume aggressive chip-level integration [4]: 12ns L2 hit, 80ns for local memory, and 150-200ns for 2-hop and 3-hop remote misses. This data is summarized in Table 3.6.

L1 cache	64KB, 64B lines, 2-way instruction and data
L2 cache	1.5MB, 128B lines, 6-way unified 12ns latency
Memory	80ns local access 150ns 2-hop remote access 200ns 3-hop remote access
CPU	1-way, in-order, 1GHz clock freq. 4 CPU multiprocessor system
Oracle	8 server processes per CPU 500 transactions per server

Table 3.6. Default SimOS simulations setup.

We used SimOS to monitor the memory hierarchy of a multiprocessor system running an OLTP workload. We obtained results regarding L1 instruction caches (data and instructions), the shared L2 cache (but distinguishing data misses from instruction misses), and TLB (both data and instructions).

In addition to direct monitoring experiments, we also used SimOS to obtain instruction traces from our OLTP workload, containing information about the interaction between application and system code. We then analyzed these traces using a very detailed instruction cache simulator which provides data about temporal and spatial locality characteristics of the trace examined.

The results in Figure 4.21 were obtained using SimOS, and the analysis of instruction cache impact in Section 4.2.1 were obtained by analyzing the traces obtained.

3.3.5 Real machine runs

In addition to using SimOS to simulate a multiprocessor system running an OLTP workload, we used Spike to obtain an optimized version of the Oracle DBMS, and examined its behavior on a real multiprocessor system.

The setup examined on the real systems is exactly the same as the one examined under SimOS. We tested two different Alpha platforms: a 21164 and a 21264, both shared memory multiprocessors.

After code layout optimizations, the improved efficiency of the workload often causes it to be more I/O bound. This leads to higher idle time, making elapsed execution time comparisons meaningless. In practice, the workload can be re-tuned to eliminate the excess idle time. However, such re-tuning would further modify the behavior of the optimized runs with respect to the unoptimized runs, making it difficult to isolate the effects of our optimizations. Therefore, we chose to use non-idle execution cycles instead of elapsed execution time as our performance metric. We used DCPI and the Alpha performance counters to measure this non-idle execution time.

The results for our real machine runs are shown in Figure 4.23. The performance results obtained compare correctly to those experienced in full-size audit runs of TPC-C.

3.3.6 Ideal pipeline simulator

After all the analysis of the impact of code layout optimizations, we propose a new fetch architecture which aims at minimum complexity, while maintaining a high fetch performance. We evaluate our fetch architecture in comparison to a classic BTB architecture, and the trace cache architecture.

We simulate the processor front-end in detail, and allow wrong-path instruction fetch to allow wrong speculative predictor updates, and the possible interference and prefetching effects on the instruction cache.

The processor back-end simulates a perfect in-order pipeline, with a perfect L1 data cache. Branch mispredictions are detected at the commit stage (last pipeline stage), and non-speculative branch predictor updates and fetch redirection also happen there. That is, the processor performance is limited only by the fetch stage.

	small predictor	large predictor
BTB	512 entries, 4-way	2048 entries, 4-way
gshare	14 bits, 16K entries	16 bits, 64K entries
stream pred.	1024 entries, 4-way	4096 entries, 4-way
trace pred.	1024 entries, 4-way	4096 entries, 4-way
RAS	64 entries	
issue width	2, 4, and 8	
pipe depth	8 stages	16 stages
FTQ	4 entries	
inst. cache	64KB, 2-way	
trace. cache	32KB, 2-way	
line size	4x pipe width	
L2 latency	15 cycles	

Table 3.7. Detailed simulator setup for the complete processor simulator.

Table 3.7 shows the values used in the processor simulation. Most of the setups correspond to the fetch engine, which was simulated in greater detail. The BTB architecture uses only the BTB and gshare predictors, the stream architecture uses only the stream predictor, but the trace cache uses both the trace predictor and the BTB.

We simulate two different processors: one with a short pipeline (8 stages) and a small branch predictor (6KB of storage), and one with a deep pipeline (16 stages) and a large branch predictor (40KB). For both processors, we simulate fetch and issue widths from 2 to 8, with cache line sizes from 8 to 32 instructions (32 to 128 bytes).

The trace cache processor uses a 32KB trace cache (instruction storage), and a 32KB instruction cache, both 2-way set associative. We tested multiple combinations of instruction and trace cache sizes (big trace cache with small instruction cache and vice versa), and selected the optimum one. We use optimized codes for all setups.

This simulator allows us to compare the performance of the three fetch architectures in terms of processor IPC, as well as measuring the individual fetch performance factors (memory latency, fetch width, and prediction accuracy).

3.4 Final remarks

We have used a large variety of workloads and simulators in this work. The initial work was done using a simple fetch engine simulator that we developed ourselves, and the TPC-D workload only. Later on we included the SPEC benchmarks. Then SPIKE was made available to us, and we could use it to generate optimized applications which could be run on a real machine, which changed our simulation methodology. At the same time, we had the chance to work with Oracle and TPC-B, which further extended our benchmark set[2]. Finally, new and more complete simulators were developed as they were needed to obtain more detailed results, and better knowledge about the internal behavior of optimized applications.

However, we do not believe that having used different benchmarks, and different simulators across this work invalidates our conclusions in any way, but quite the opposite. We have obtained similar and consistent results using different applications and different workloads, which makes us feel confident about the validity of our experiments. Furthermore, our simulation results were validated by multiple real machine runs, and even audit size runs of commercial workloads. We believe that far from making our results dubious, the consistency of the results across multiple platforms and benchmarks makes our conclusions more solid.

[2]Many thanks to Luiz Barroso and Kourosh Gharachorloo, as well as the whole Western Research Lab. team for the unique opportunity granted to us to interact with them, and access their tools and expertise on the subject.