Profiling the Predator Query Execution Engine

Bjarke Buur Mortensen

Abstract

Predator is a freely available database system. This paper studies the performance of the Predator query execution engine using high-level and low-level profiling. We describe the aspects of running database systems on modern processors and the tools available for investigating how these aspects influence performance. We follow the methodology of previous work in database systems profiling. Our results confirm previous findings and we investigate further into the causes of these findings by looking at how various parameters of variation affect performance of query execution.

1 Introduction

The Predator Database System is an open source database system which is freely available. It was developed by Praveen Seshadri at Cornell University and is now maintained at University of Copenhagen.

The goal of this project is to study the performance of the Predator query execution engine. Even if Predator has not been designed for performance, but for extensibility and flexibility, its architecture is representative of current database systems: a query execution engine is used as a virtual machine to evaluate compiled queries.

Because we have the source code for Predator we can study performance using both high level and low level profiling. *High level profiling* aims at mapping performance bottlenecks to source code organisation. This is done by constructing flat program profiles, i.e. listings that show the individual execution times for functions in the program. More sophisticated profiling comes from *program paths* [1], records of the dynamic order of a program's blocks or functions as the program runs.

Low-level profiling has been studied in the context of database systems by Ailamaki [2]. It focuses on processor and memory behaviour as the program executes. Low-level profiling provides insight into how well the system utilises the hardware it runs on, and can thus help identify limitations in the implementation of critical parts of the code.

Applying both high level and low level perspectives in the study of performance is important in order to get a complete understanding of a system's characteristics. Consider for instance a very performance-conscious, low overhead software design: it will not yield the desired performance if the compiled code and data structures are bad at utilising the platform on which the system runs. Likewise, a system with well-tuned critical functions will perform poorly if the system design induces unnecessary overhead.

In this project we study high level and low level profiling of the Predator query execution engine using experiments similar to those conducted by [1] and

[2]. The basic motivation for the experiments is to find out how the execution engine can be improved. We use profiling to understand the details of query execution. In particular, it is interesting to investigate what impact the flexibility and extensibility design decisions have had on performance. We focus on identifying inefficiencies at the call graph level: what are the program paths travelled too often and which program paths are unnecessarily long? A long term goal is to study how beneficial it would be to specialise the query execution engine for a given query. Our hope is that such a specialisation can lead to more efficient call graphs.

We have two goals for our low-level study. First, we investigate how several parameters of variation influence query execution at the low level. We then map these findings to source code organisation. Second, we compare our results to those of Ailamaki [2], studying whether Predator shows similar limitations in hardware utilisation.

Predator is a complex system with a large code base. As an added benefit, our profiling efforts will make it easier to understand the coupling between the components of the system, thus making it easier for us to maintain and extend Predator.

We make the following contributions:

- We develop an experiment framework that allows for many different kinds of profiling. It will allow people working on Predator to easily profile the changes they make. This framework is now available online [3].
- We combine the profiling tools to study the performance of the Predator query execution engine
- We find that Predator, like other database systems, rarely spend more than 40 percent of total query time doing actual computation. The rest is spent stalling in the processor for various reasons.
- We identify the causes of these stalls as coming from choice of algorithm, critical parts of code or software design issues.

The rest of the report is organised as follows. In section 2 we present the background for our experiments. We explain the approaches taken by [1] and [2] for the high level and low-level profiling, respectively. Furthermore we describe the tools available for performing the experiments. Section 3 details the experiment setup. In section 4 we analyse the results of our experiments. We pick out four different issues pertaining to query execution. In each case we interpret the results of low-level profiling and map these results to high level issues where applicable. Section 5 concludes and gives pointers to future work.

2 Background

This section describes the background for our experiments. We explain the different aspects of high level and low level profiling and the methods used in the project. We also describe the tools used in the experiments.

2.1 High Level Profiling

The most common type of profiling produces flat profiles. Flat profiles list the execution times for each function called together with the number of times a function was called. Flat profiles are ideal for identifying *hot spots* in the code, i.e. functions that are candidates for optimisation. When a critical function has been identified, further analysis can be made using code coverage tools that report how many times statements in the code are reached.

The reason flat profiling is such an effective tool, is that most programs follow the 80-20 rule: 80 percent of a programs execution takes place in 20 percent of the code [1]. Performance gains in these 20 percent will have a high impact on the total running time of the program.

While finding hot spots in the code is important, it does not always suffice. First, it might be the case that the hot spots, i.e. the critical functions, are already well-optimised and cannot be improved upon. Second, the program might not follow the 80-20 rule, in which case there are no easily identifiable functions to optimise[1]. Last, a program can exhibit performance problems even if the flat profile does not point to any hot spots. This is due to high level design decisions such as function relationships and indirections that lead to too many function calls.

These cases require one to look at program execution at a higher level to discover performance issues that are not inferable from the flat profile.

To approach analysis at a higher level, we can use the concept of program paths. Program paths record the dynamic order of program blocks or functions as the program executes. Ball and Larus [1] introduced them at the intraprocedural level, whereas Larus [4] extended the concept to *whole program paths*, i.e. paths that cover the entire dynamic flow of a program execution. A simple kind of whole program path is a call graph, i.e. a graph showing the relationship between functions as they are called during execution. This definition of paths is well suited to our needs, since we are interested in reasoning about the paths from a design point of view. The call graph can tell us which functions a specific function calls, and how many times it calls them. This gives us information to reason about design problems, e.g. indirections, redundancy and other inefficiencies. It can help us identify *hot paths* in the program, i.e. paths that are candidates for optimisation. Optimisation can be approached by investigating whether these paths or parts of them are unnecessarily long or travelled more than needed.

It is possible to produce both static and dynamic call graphs. For complex applications, static call graphs tend to become very large as the number of possible paths is very high. Dynamic graphs are easier to analyse, because they report only the functions that were actually called at run-time. Furthermore, dynamic graphs can be compared between different runs of the program, for instance in an effort to answer why one run is significantly slower than the other.

In this project we use dynamic call graphs to understand and illustrate Predator's high level design.

[1] Confusingly, this kind of flat profile is said to be 'flat', as opposed to the typical 'steep' profile.

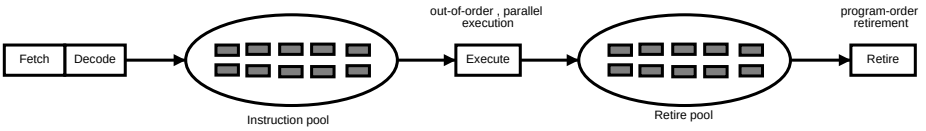


Figure 1: Out-of-order pipeline (adapted from [7])

2.1.1 Tools

The tool *gprof* [5] that comes with the popular *gcc* compiler can produce flat profiles as well as call graphs for executables that have been instrumented during compilation. The number of function calls reported for each function are exact, but the run-time results are obtained through sampling, and are thus subject to statistical inaccuracy.

We use gprof to construct the call graphs. We then use Graphviz [6] to visualise the call graphs by converting the call graph data into a format Graphviz can understand. Graphviz does automatic graph layout, which is advantageous because call graphs typically contain many nodes and edges.

2.2 Low Level Profiling

Modern CPUs are superpipelined and superscalar [7, 8]. Superpipelined processors allow for high clock rates because the traditional pipeline steps are further subdivided. The Pentium II processor used in our experiments has 14 pipeline stages whereas the newer Pentium 4 has 20 stages (see e.g. [9]). Superscalar processors have multiple execution units to allow instructions to be executed in parallel. Typical processors are capable of executing 3-8 instructions per cycle [10].

Several factors prevent the processor from running at full speed. Superscalar execution requires that the instructions executed in parallel are independent of each other. For instance, two instructions i and j executed in parallel cannot use as source the output of each other. This is typically the case when e.g. instruction i loads data from the memory system into a register and instruction j uses that register in an arithmetic operation. Instruction j cannot be executed until i has fetched the data. This problem is particularly acute when the load instruction causes cache misses, because of the so-called memory gap: the processor can execute instructions much faster than the memory system can deliver them.

To hide some of the stall costs a processor uses out-of-order execution (see figure 1). Out-of-order execution means that the processor fetches and decodes instructions and places them in an instruction pool. Special hardware then examines the instructions, and executes those that have no outstanding dependencies, allowing the processor to do more useful work. When the instruction has been executed it is placed in a retire buffer. From there instructions are retired in program-order.

Branch instructions change the program flow, but the decision of whether to follow a branch or not is not known until the branch condition has been evaluated a couple of stages into the pipeline. To prevent this from stalling the pipeline, modern processors use speculative execution. Branch prediction hardware guesses whether a branch is taken or not and the processor fetches and

executes the next block of instructions speculatively according to the prediction. This means that the processor does not need to stall when it encounters a conditional branch instruction, and that the penalty of branch instructions is limited to the case when conditional branches are mispredicted by the hardware. However, mispredicting a branch means that the instructions executed in advance have to be discarded. Since the pipeline is deep and the processor can issue several instructions per clock cycle, many instructions will have been issued before the branch condition is actually evaluated. In case of misprediction, the instructions executed in advance prevent useful instructions from executing. This makes branch mispredictions expensive.

Most modern CPUs allow programs to access special hardware performance counters. These counters are registers that can be configured using special instructions to measure hardware events. The events are for instance number of branch instructions executed, number of requests to memory, and number of instructions retired. Using these events the low level behaviour of program code can be characterised [11, 12]. For instance, the ratio between number of branch instructions and mispredicted branch instructions characterises how well a program exploits the branch prediction hardware.

2.2.1 A model for understanding hardware behaviour of database workloads.

Another way to use the performance counters is to break down total execution time into its low level constituents. This is done for the SPEC95 benchmark suite in [12] and more systematically for database management systems in the recent work by Ailamaki [2]. Ailamaki shows that database systems do not scale well to the performance offered by modern CPUs. More than half of the time is spent stalling when executing simple queries. The model she used to obtain these results breaks total running time into computation time and stall time. The stall time is further divided into components based on the reason for the stall. Total execution time T_Q is decomposed into the following components:

$$T_Q = T_C + T_M + T_B + T_R - T_{OVL}$$

T_C is the actual computation time; T_M corresponds to stalls related to cache misses; T_B iscorresponds to stalls related to branch mispredictions and T_R corresponds to resource stall time. Resource stalls are stalls caused by unavailability of resources in the processor, such as functional units and buffer slots. Some of the stalls are overlapped by the processor, which is why an additional component T_{OVL} is subtracted to obtain the total time.

Table 1 shows how Ailamaki's model is further divided into components that correspond to hardware events that can be measured using the hardware performance counters. The numbers obtained from the hardware counters are not directly comparable. For some events the actual stall time in clock cycles is returned whereas others return the number of events. For the latter, the event count has to be converted to cycles by multiplying the event count with an architecture specific cycle penalty. We explain the events and how to obtain them briefly. A more thorough explanation can be found in [2].

Computation time (T_C) Computation time is the time spent doing useful computation. The Pentium II converts instructions into so-called *µops*

Component		Description	Measurement method
T_C		Computation time	$0.3333 * \#\mu ops\, retired$
T_M	T_{L1D}	Level 1 data cache stalls	$\#misses * 4\, cycles$
	T_{L1I}	Level 1 instruction cache stalls	$\#misses * 4\, cycles$
	T_{L2D}	Level 2 data cache stalls	$\#misses * 72\, cycles$
	T_{L2I}	Level 2 instruction cache stalls	$\#misses * 72\, cycles$
	T_{DTLB}	DTLB stalls	Not measured
	T_{ITLB}	ITLB stalls	$\#misses * 32\, cycles$
T_B		Branch misprediction stalls	$\#branch\, mispredictions\, retired * 17\, cycles$
T_R	T_{FU}	Functional unit stalls	actual stall time
	T_{DEP}	Dependency stalls	actual stall time
	T_{ILD}	Instruction-length decoder stalls	actual stall time
T_{OVL}		Overlap time	not measured

Table 1: Execution time components and measurement method on a Pentium II processor (adapted from Ailamaki [2]).

(micro operations) before executing them. The Pentium II can execute up to 5 *µops* each cycle. We calculate T_C using a pessimistic measure of 3 *µops* per cycle times the number of *µops* retired. This means that possibly fewer cycles are spent doing computations [13].

Memory stalls (T_M) The memory stalls are divided according to whether they are caused by instruction fetch or data load. Furthermore, we categorise the stalls by whether they occur in the small and fast *level 1 cache* or the larger and slower *level 2 cache*. The penalty of 4 cycles for level 1 cache misses is from [2]. The 72 cycles penalty for level 2 cache misses is the measured memory latency on our test platform. We measured this value using the *lmbench* benchmark suite [14]. ITLB and DTLB (Instruction or Data Translation Lookaside Buffer) are caches used for translating virtual addresses into physical ones. An ITLB miss costs 32 cycles. On the Pentium platform it is not possible to measure DTLB misses, which means that cycles spent due to this stall condition are not included in the results.

Branch misprediction stalls (T_B) Branch misprediction stall time is computed from the number of retired mispredicted branch instructions. The penalty is 17 cycles.

Resource stalls (T_R) Resource stalls are divided into stalls caused by functional unit contention (T_{FU}), stalls caused by instruction dependencies (T_{DEP}), and instruction length decoder stalls (T_{ILD}). T_{ILD} accounts for stalls that occur when the Pentium II translates instructions into *µops*. For all three events the hardware measures the actual stall time.

Overlap time (T_{OVL}) As detailed above modern processors hide some of the cost of stalls by speculative execution and out-of-order scheduling. Whereas we account for the speculative execution by only attributing stall penalties to mispredicted branches, we have no way to measure the influence of out-of-order execution on the execution time. This means that the time

we attribute to stall components that can be overlapped, notably data cache misses, will be upper bounds.

We use the model to investigate whether Predator shows similar behaviour as the DBMSs investigated by Ailamaki. Whereas Ailamaki was interested in finding the general execution time break down for database systems, we can go further with Predator since the source code is available. We investigate the mapping between stall time and source code to find pieces of code that make poor use of the hardware.

2.2.2 Tools

There are numerous tools available for accessing performance counters on Linux platforms, which we use as software platform for our experiments. We use the Intel Pentium II processor as our hardware platform. An easy tool to use is Rabbit [15]. Even though the Pentium II only has two performance counters available, Rabbit can measure an arbitrary number of events in one run of the program by multiplexing the counters. However, Rabbit suffers a number of caveats. First, multiplexing makes the event counts obtained less accurate. Second, Rabbit collects counts on a system basis. This means that other processes' performance will pollute the results.

Ailamaki used the cross platform tool PAPI, the Performance API [16, 17] for the parts of her experiments performed on the Linux platform. It is available for many different variants of Unix and for Windows and for many different hardware architectures. It uses the native performance counters of the architecture on which it runs but presents a uniform API across platforms, making it easy to move experiments between platforms. PAPI counts events on a per process basis. On the Linux platform, this feature requires a kernel patch. Furthermore, PAPI is a library, so it requires that the program being examined is instrumented to set up and tear down performance measurement. For our remake of Ailamaki's experiments we use PAPI because of its precision and its ability to collect profiling data on a per-process basis.

OProfile is a tool for doing system-wide profiling [18]. It uses platform specific performance counters to implement a low-overhead continuous profiler. It runs on Linux systems using either Intel x86 or Athlon processors. It does not require any kernel patch but relies on a loadable kernel module. Once OProfile has been started it monitors all system activity. As events occur they are attributed to the currently active process using the overflow capabilities of the performance counters. When profiling is finished, various kinds of post-processing can be made. A percentwise distribution of events over all the running processes can be generated. A detailed profile of a specific program can also be produced. This profile can be thought of as a flat profile like the ones gprof produces. However, instead of showing running time breakdown, it shows the hardware event occurrence breakdown. In addition, if programs have been compiled with debugging information, a tool can generate annotated source code, where events are attributed to a specific source code line. This feature is not accurate; compiler optimisations such as inlining make it difficult to attribute events accurately to source code lines.

Component		Native hardware event	PAPI equivalent
T_C		UOPS_RETIRED	N/A
T_M	T_{L1D}	DCU_LINES_IN	PAPI_L1_DCM
	T_{L1I}	L2_IFETCH	PAPI_L1_ICM
	T_{L2D}	L2_LINES_IN - BUS_TRAN_IFETCH	PAPI_L2_TCM - PAPI_L2_ICM
	T_{L2I}	BUS_TRAN_IFETCH	PAPI_L2_ICM
	T_{DTLB}	N/A	N/A
	T_{ITLB}	ITLB_MISS	PAPI_TLB
T_B		BR_MIS_PRED_RETIRED	PAPI_BR_MSP
T_R	T_{FU}	RESOURCE_STALLS	PAPI_RES_STL
	T_{DEP}	PARTIAL_RAT_STALLS	N/A
	T_{ILD}	ILD_STALL	N/A
T_{OVL}		N/A	N/A

Table 2: Mapping stall time components to native performance counter events and the equivalent PAPI events.

2.2.3 Mapping of PAPI counters to native performance counters

Table 2 shows how the running time components are mapped to the Pentium II's hardware events. T_{DTLB} does not have a corresponding hardware event and is thus not measured. Furthermore some events are not available directly in PAPI as a predefined event. Fortunately, PAPI supports measuring such native events by specifying their native event number. Also note that there is no hardware event for measuring level 2 data cache misses. Instead we derive the count by subtracting the level 2 instruction cache misses from the total level 2 cache misses.

During our work with PAPI we discovered some inconsistencies regarding level 2 cache misses. PAPI reported a higher number of level 2 instruction cache misses than total level 2 cache misses! It turned out that PAPI mapped the native Pentium II event `IFU_IFETCH_MISS` to `PAPI_L2_ICM`, i.e. level 2 instruction cache misses. We conducted a series of tests that indicated that the correct native event to use is `BUS_TRAN_IFETCH`, as reported in table 2. This mapping has now been adopted by PAPI. However, the Pentium II Software Developer's Manual [19] is ambiguous in its wording on the meaning of the different counters. Intel confirmed that we are actually using the correct mapping [20].

3 Experiment Setup

This section describes the experiment setup used in the project. We describe the hardware used for the experiments (section 3.1), give details about the software and experiment framework (section 3.2), and describe the workload used (section 3.3).

3.1 Hardware

For our experiments, we have used a system with a 450 MHz Pentium II processor and 256 MB ram.

For the low-level experiments the choice of hardware is important. The Pentium II is the same processor used by Ailamaki. This gives us the benefit that we can adopt the model used by Ailamaki directly. This is fortunate, since the hardware event penalties detailed in table 1 are not well documented in the literature.

The Pentium II is not a new processor. The newest processors are now clocked at well over 2 GHz and the pipeline depth has also increased steadily. Thus, it is reasonable to expect that the impact of stall conditions discovered by studying the Pentium II will be even more significant when studying the newest processors available. However improvements in processor implementation might mask or even revert some of these trends. For instance, the newest processor in the Pentium series, the Pentium 4, has a significantly larger branch prediction buffer and a new kind of level 1 instruction cache, called a trace cache (for details see e.g. [21]). Thus it would be interesting as future work to redo the experiments on a Pentium 4 or a similar new processor.

3.2 Experiment framework

We have developed an elaborate experiment framework for Predator. The framework provides ways to easily run the different tests used in this project (gprof, PAPI, OProfile). It is implemented through the use of Makefiles and a single perl-script, that starts and stops the server and makes a client program issue queries to the server. Adding a new kind of workload to the test-suite amounts to creating a new directory, creating queries for setting up and tearing down the tables and indices used in the workload and then placing the workload queries in separate files in the directory. The experiment framework is available on the Predator web site [3], so people working with Predator have an easy way of profiling the changes they make. It is our hope that the insight produced this way will go back into improving Predator.

Running different experiments requires that the Predator server is compiled with the right options. For OProfile to do annotated source output, the executable needs to have debugging information built in. To produce gprof profiles and call graphs the executable must be instrumented to trace function calls at compile time. Finally, to use PAPI, the Predator executable must be instrumented to start and stop counting. To allow maximum flexibility, we have implemented this as two commands, *papi_start* and *papi_stop*, that can be issued in the Predator server at the desired time. The framework issues these commands just before and just after the query is executed, respectively.

The Predator source code configuration provides parameters to manage all these different options as well as options for controlling compiler optimisations.

Our experiments have been carried out on Debian Linux system using version 2.4.17 of the Linux kernel. The kernel has been patched with the Perfctr performance counter patch that PAPI requires. The compiler used is gcc-2.95-4. Both PAPI and OProfile are still projects in development, so both programs have been obtained from the projects' version control systems.

3.3 Workload

For our experiments, we follow the lead of Ailamaki and choose a workload of range selection queries and table equijoins on a memory-resident database. We

do this to leave out the costs of concurrency control and the I/O subsystem (Predator uses the Shore storage manager [22] for this) thereby focusing on the pure CPU and memory performance of the execution engine. Furthermore, when we inspect the code more closely, we focus mainly on CPU performance. We have added a CPU intensive query from the FinTime benchmark suite [23] to our workload. The workload queries are listed in appendix A. Queries A.1, A.2 and A.3 are from Ailamaki. Query A.4 is from FinTime.

The FinTime query A.4 uses the tables prescribed by the the FinTime benchmark suite. The tables R and S used in queries A.1, A.2 and A.3 both have three integer fields, $a1$, $a2$ and $a3$. R_w_idx is identical to R, but has a non-clustered index on the field $a2$. Ailamaki added padding fields to the tables to make the record size 100-bytes. This record size will reveal more about data cache performance. Our experiments have primarily focused on finding mappings from low level performance to source code organisation. Thus we have chosen to ignore the padding fields. In addition, our table R contains only $120,000$ records, whereas Ailamaki's R table contains 1.2 million records.

To ensure that the data involved in the queries is resident in memory we rely on Shore's caching abilities. We start the Predator server and issue the queries multiple times before starting measurements. This puts the relevant tables and indices in memory, and warms up the caches.

4 Experimental Results

This section analyses our experimental results. We have identified a number of issues that can be compared across queries. We compare files scans with index scans (section 4.1), the impact of selectivity in range selections (section 4.2), the behaviour of different join methods (section 4.3) and the impact of selection criteria on performance (section 4.4). In section 4.5 we extract some general trends from the comparative analysis.

In the following sections, we use these abbreviations:

SRS sequential range selection (query A.1)

IRS indexed range selection (query A.2)

SJ sequential join (query A.3)

FTA FinTime aggregate (query A.4)

4.1 Sequential scans vs. index scans

Figure 2 shows the execution time breakdown (a) and the number of instructions executed per record (b) for SRS and IRS with 10% selectivity. IRS is approximately 2 times slower than SRS.

IRS's execution time is dominated by memory stalls and resource stalls. Only 25% of the CPU time is spent on actual computation. 25% of total execution is attributed to level 2 data cache stalls (T_{L2D}). Likewise 28% comes from resource stalls caused by contention for functional units (T_{FU}). Using OProfile, we can

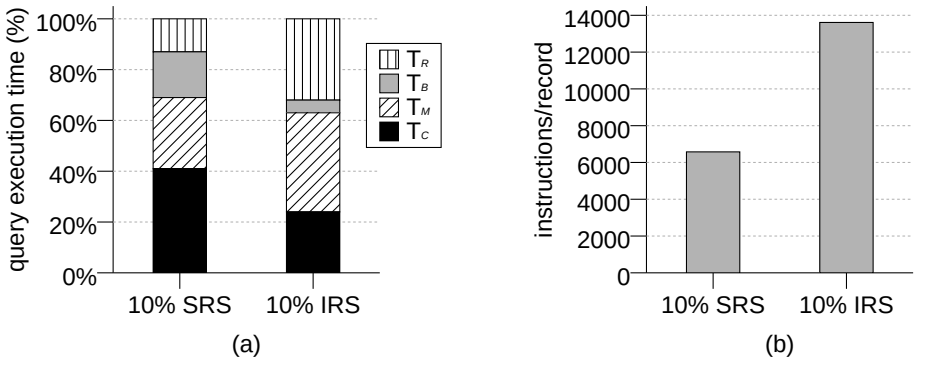


Figure 2: Range selection using 10 % selectivity. (a) Execution time breakdown. (b) Number of instructions executed per record. For SRS 79% of T_M comes from level 1 instruction cache misses. For IRS 64% of T_M comes from level 2 data cache misses.

see that the majority of stalls for both level 2 cache misses[2] and functional unit contention occurs in the same five functions. Even though the correlation between cache misses and functional unit stalls is not one-to-one, we believe that it is caused by a contention for the processor's load/store unit. See appendix B.1 for the OProfile output.

That IRS is dominated by data cache misses is due to the random distribution of the range attribute $a2$. Since the index is non-clustered the records pointed to by the index will be randomly distributed throughout the table. With a ten percent selectivity we must expect to retrieve many cache lines at random and return to them when they might have been evicted from the cache. This is a smaller scale instance of the performance degradation that is common when swapping many pages between disk and main memory due to capacity constraints in main memory. It intuitively supports the notion that the processor cache is to primary memory what primary memory is to external memory. SRS's performance, data cache-wise, is much better. Since the table is scanned sequentially, each record is brought into cache exactly once.

Looking at the SRS query we see that the branch stall time component account for approximately 18% of SRS's execution time. Since we have a 10% selectivity, we expect that the branch misses occur when a record is selected for inclusion in the aggregate calculation (the uncommon case). However, OProfile tells us that selection criteria evaluation (`ExprEval` and related functions) are only a partial cause of mispredictions. Most branch mispredictions are caused by calls to functions related to buffer management and locking. These are also part of the critical path that is evaluated for each record. This seems to support the findings of [2] that even for simple workloads, the number of branches on the critical path are so high that the size of the branch prediction buffer is insufficient to store the results of all branches. The processor therefore resolves

[2] We cannot measure level 2 data cache misses separately, so we have to look at total level 2 cache misses instead. However, level 2 instruction cache misses for IRS are only 1% of the total level 2 cache misses, so the OProfile output for total level 2 cache misses must be nearly identical to level 2 data cache misses.

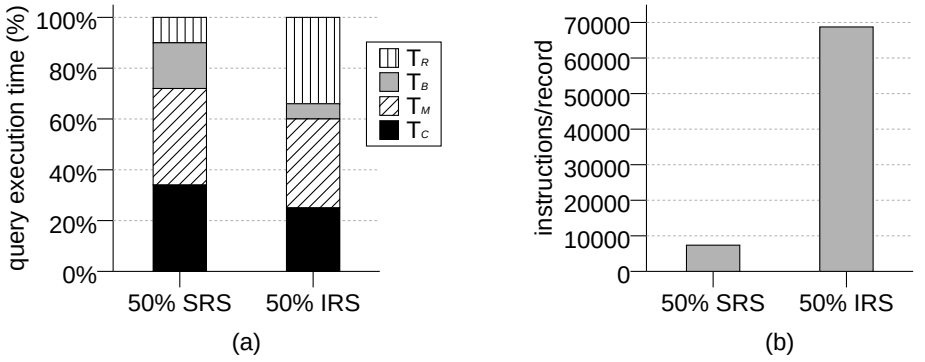


Figure 3: Range selection using 50 % selectivity. (a) Execution time breakdown. (b) Number of instructions executed per record.

to static prediction which is correct only half the time on average.

The other major stall factor for SRS is level 1 instruction cache misses. These account for 23% of the total running time. As shown by [2], there is a strong correlation between branch mispredictions and instruction misses, because mispredictions changes the instruction flow. Appendix B.2 shows the functions that cause most branch mispredictions and most level 1 instruction misses. Although there is not a one-to-one correlation, the profile confirms Ailamaki's results.

From figure 2 (b) we see that the number of instructions executed is twice as many for IRS than for SRS. Looking at the instruction execution profile (appendix B.3), we see a recurrence of some of the functions that causes data cache misses and resource stalls. This means that efforts to reduce the running time of indexed access should simultaneously focus on improving the cache performance and the instruction count for these functions.

4.2 Impact of selectivity

This experiment tests the impact of selectivity on performance. We run the same queries as in the previous experiment but change the range of the selection so that 50% of the records match. For both SRS and IRS, figure 3 (a) shows that the relative breakdown of execution time is nearly identical to the case with 10% selectivity. Not surprisingly, the performance of SRS 50% is only slightly slower than SRS 10%. The same number of records are scanned (i.e. all of them) which means that the number of data cache misses is unchanged. The extra execution time is reflected in a slightly higher number of instructions executed per record. More work has to be done when more records are part of the final result.

A more interesting observation is that the branch misprediction rate is unchanged (it still accounts for 18% of total running time). When selectivity is 50%, the branch that decides whether a record is included in the result or not will be taken exactly half the time. This is a so-called hard-to-predict branch [10] that will be mispredicted on average half of the time, causing an increase in branch mispredictions. However, as described in section 4.1 above, it is likely that the branch prediction buffer is insufficient to store all branches on the critical path, so the branch prediction hardware is already exhibiting worst-case

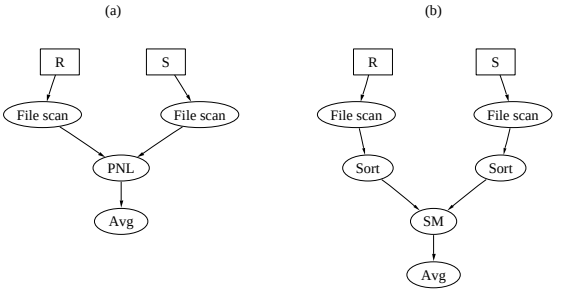


Figure 4: Query execution plans for SJ (a) using page-nested loop and (b) using sort/merge join. PNL: page nested loop; SM: sort/merge

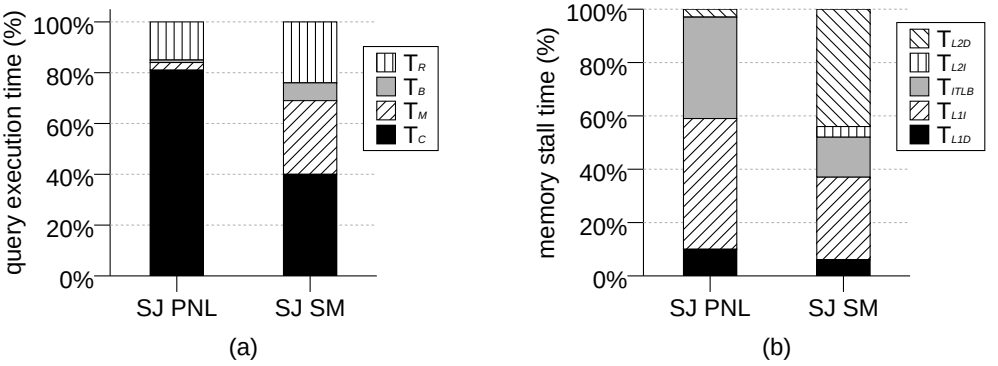


Figure 5: Sequential join. (a) Execution time breakdown. (b) memory stall breakdown

behaviour.

As expected the running time suffers when using an index to select 50% of the records. The running time is 5 times slower than for 10% IRS. A larger part of the index is scanned, which generates more random accesses to the table. This creates more data misses and functional unit stalls. However relative to the other stall components, data cache misses have not increased. The reason is again the extensive use of the hash table lookup function, which is responsible for more than half of the instructions retired.

4.3 Join methods

This experiment compares the behaviour of two different join methods. We do this by executing the same query twice, in each case instructing the query optimizer to use a specific join method. The query we use is SJ. The two methods we use is page-nested loop (PNL) and sort-merge join (SM). Because PNL is very inefficient for large joins, the experiments were carried out using scaled down versions of R and S. We used tables one tenth of the original sizes, i.e. R contained 12000 records and S contained 4000.

Figure 4 shows the query execution plans for SJ using (a) page-nested loop and (b) sort/merge join. The PNL algorithm iterates over the outer table R, each time retrieving a page of records. For each page of records, PNL iterates over the records of the inner table S and evaluates the join to find matching tuples. SM first sorts the two tables of the join based on the join attributes, and then merges the sorted tables in one pass. Sort-merge join is much more efficient than page-nested loop. It takes approx. 1.7 seconds to execute the query using SM whereas PNL uses approx. 73 seconds.

Figure 5 shows the execution time breakdown (a) and the memory stall components in detail (b). The difference in runnning times is due to the difference in algorithm, so the graphs cannot be used to explain that. Instead, they show how the choice of algorithm affects low-level behaviour.

The most apparent observation about PNL is that more than 80% of the time is spent doing computation. Looking closer at where time is spent we see that the 10 most active functions all have to do with record fetching and expression evaluation (see appendix B.4). When we examine the memory stall breakdown (figure b), PNL is dominated by level 1 instruction cache misses (49%) and instruction translation lookaside buffer misses (38%). This obviously indicates that the system is using a lot of instructions to execute the query. More importantly, the number of ITLB misses is higher than for the other queries. An ITLB miss occurs when a translation from virtual memory address to physical memory address is not cached in the ITLB. This indicates that more instructions are executed and also that the instructions cover a larger portion of the code base. This is either because the system accesses more pages of code (code coverage), or because of bad code layout, i.e. code used by PNL is distributed over unnecessarily many pages.

The sort-merge join algorithm shows similar low-level behaviour as the range selection queries we have studied in sections 4.1 and 4.2. This means that approximately 40% of the time is spent doing useful computation. The functions accounting for most of the time are sorting or buffer managment (see appendix B.4).

Memory stall time for SJ is dominated by level 2 data cache stalls. This is due to the random table access patterns created by sorting the tables R and S. Overall, SM has more level 2 instruction cache misses than PNL[3], but significantly fewer level 1 instruction cache misses. This might at first seem contradictory, but it makes sense if you consider level 2 instruction cache misses to be indicative of the amount of code that has to be covered to execute the query, and level 1 instruction cache misses to be indicative of the number of instructions needed to execute it. For SM, a higher number of level 2 instruction cache misses points to the two phase nature of the sort/merge algorithm: code for both sorting and merging has to be covered. The low number of level 1 instruction cache misses compared to PNL (approx. one tenth in absolute numbers) is due mostly to the more efficient algorithm. SM has another advantage over PNL. Since the merge step cannot be activated until the tables are completely sorted, and since the sort and merge steps seen in isolation are simpler than page nested loop (requires less code). the amount of code active at any given time is smaller. SM therefore benefits from temporal locality in its instruction access patterns.

[3]For PNL it is less than 1% of the memory stall time, and cannot be seen on the graph.

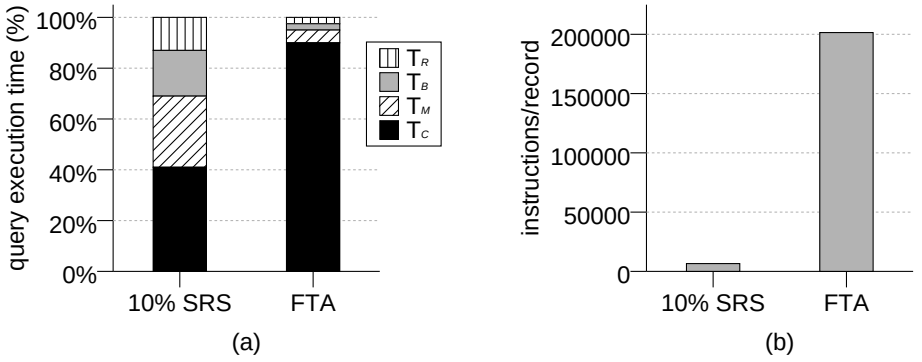


Figure 6: Comparing 10% SRS and FTA. (a) Execution time breakdown. (b) Number of instructions executed per record.

This kind of experiment is useful for showing how choosing different join methods impact performance. Even though in this particular case the choice is clear, such experiments can provide insight into the validity of the query optimiser's choice of execution plan.

4.4 Selection criteria

The purpose of this comparison is to examine how selection criteria affect performance. We do this by comparing two sequential selection queries, SRS and FTA. FTA is a modified version of query 1 of the Historical Market Information part of FinTime (see [23]). In order for us to more easily compare to SRS, we have removed the group by clause and only have one aggregation. Furthermore, the table generation software that comes with FinTime outputs rows ordered by date. We have permuted the records of the table because we want the distribution of the range attributes to be random as it is for SRS. This way the only thing that separates SRS from FTA is the selection criteria.

Figure 6 shows the execution time breakdown (a) and the number of instructions executed per record (b) for SRS and FTA. Figure (a) reveals that for FTA approximately 90% of the time is spent doing computation. This is contrary to the trend that the other selection queries show. However, FTA is almost 10 times slower than SRS, which for the most part is due to the number of instructions required to perform the more complex expression evaluation.

Most of this slowdown comes from the more complex expression evaluation required. Appendix C.1 and C.2 show the call graphs for the expression evaluation part of the execution path for SRS and FTA, respectively. Because the range selection attributes `Id` and `TradeDate` of FTA are of more complex types than the integer used by SRS, the expression evaluation requires some additional work to retrieve the values from storage (`XxxADTClass::TypeCopy` and related functions). Furthermore, examining whether the current record's `Id` is in the list of strings is done by iterating over the values of the list. For each list item the value is put into a generic type-buffer along with type information (by the function `XxxConstValue::ConstStringPlan::Evaluate`) and then checked for equality with the current record value by the function `ExprEval`.

For this particular case we have used a list of ten strings. The number of calls to `XxxConstValue::ConstStringPlan::Evaluate` and `ExprEval` corresponds roughly to the size of the table (120,000) times the number of items in the list (10) (we select 10 out of 120 different string values, so all items of the list will be considered for most records, namely those that do not match).

Even though the ten-fold increase in running time can mostly be attributed to the extra number of instructions required for selection evaluation, there is also an increase in absolute values of the stall time components compared to SRS. Level 2 data cache misses are increased by a factor of three and level 1 data cache misses by a factor of 10, because the FTA query fetches more data out of each row of the table. This is followed by an increase in functional unit stalls as we also observed in section 4.1.

The larger number of instructions executed causes more instruction cache misses at both level 1 (16% increase) and level 2 (33% increase). As we saw in the case for page-nested loop in section 4.3 the ITLB misses are also high here. Compared to SRS there are twice as many ITLB misses. Again, it indicates that we access many pages of instructions on the critical path.

4.5 General trends

Looking at our experiments in general, a number of trends are evident. First of all, our results show that, overall, Predator behaves like the database systems investigated by Ailamaki. Effective computation time accounts for less than half of the total execution time, except when we force the system to use suboptimal query plans.

Memory stall times range from 30 to 40 percent. This supports the findings reported by many studies (eg. [2, 10]) that the memory hierarchy is the single most important cause of bottlenecks on modern CPUs. When execution time is dominated by data cache misses, time spent on computation drops to around 25 percent of total execution time. This is the case for 10% IRS, 50% IRS, and SJ using sort-merge join. Furthermore, the clear correlation between data cache misses and functional unit contention stalls adds an indirect cost to data cache misses. This makes a strong case for the use of cache optimised data structures.

Compared to the database systems investigated by Ailamaki, Predator uses significantly more instructions per record to execute a query. Because the number of level 1 instruction cache misses is always high, even when the workload is dominated by data cache misses, it indicates that the amount of code that has to be covered on the critical path that is executed for each record, is large. We see this trend even for simple queries. For more complex queries, we see ITLB misses becoming a significant contributor to the memory stall time component, indicating that many pages of code has to be accessed when Predator executes. Another performance bottleneck is the number of branches executed on the critical path. Our results seem to indicate that the branch prediction hardware is insufficient to cope with the number of branches on the critical path.

This means that improving the query execution engine should be approached by reducing both the number of instructions and the number of branches on the critical paths of the execution engine. However, it should also be realised that database workloads are different from scientific workloads. This is indeed why they have not been able to scale as well to advances in hardware. Whereas scientific workloads typically apply a small body of code to large amounts of data,

commercial strength database systems are inherently more complex. Issues such as concurrency control and buffer management complicate the design and make it difficult to avoid stall time components altogether.

5 Conclusion

In this paper we have investigated the Predator query execution engine. We have developed an experiment framework that allows for many different kinds of profiling. We have combined these profiling techniques to gain insight into the Predator query execution engine. We have looked at four different aspects, in each case describing the low level behaviour. Where applicable, this insight has been mapped to high level issues. Our results show that the query execution engine behaviour is comparable to other database systems, in that actual computation time rarely accounts for more than 40 percent of total execution time. In Predator's case we find that the code size of the critical path that is executed for each record is causing many instruction cache misses, i.e. stall conditions that are hard to overlap. In addition, the critical path also makes poor use of the branch prediction hardware. When queries are executed that have many random accesses to data, the dominant component becomes level 2 data cache misses. Even in these cases, there is a constant overhead in the form of instruction misses and branch mispredictions, which means that actual computation time becomes lower. Our experiments have shown how low level profiling can give insight into performance characteristics and how these characteristics map to source code issues. To improve the Predator query execution engine we must develop our understanding of these mappings further.

5.1 Future Work

Our experiments point to several issues that can be investigated further.

As mentioned, the Pentium II is not a new processor. Redoing our experiments on e.g. the Pentium 4 would be interesting, because its trace cache and improved branch prediction hardware addresses some of the things we have found to be performance bottlenecks for Predator. Such a study would be able to tell if and how commercial type workloads such as database systems benefit from new hardware features.

Another issue that must be further investigated is the performance of indexed selections. We found that a few functions were responsible for a large part of computation time, resource stalls and cache misses. Optimising these functions or the use of them would bring immediate performance improvements.

For the case of branch mispredictions and instruction misses, the *gcc* compiler can do profile-based optimisations. Based on profiles of previous runs of the program, the compiler can lay out the code to better accommodate the typical pattern of use for the program. For branches, this involves writing branch code so that it fits to the static branch prediction algorithm of the processor. To reduce instruction misses the compiler can link the program in such a way that functions that are used together often are placed close together in the compiled code. This kind of optimisation is unobtrusive.

A more drastic approach is to investigate how different query representations can improve performance. For instance it should be investigated whether there

are parts of the code that is executed during query execution that can be moved to the query compilation stage.

A Workload queries

A.1 Sequential Range Selection

```
select avg(a3)
from R
where a2 < high and a2 > low
```

A.2 Indexed range selection

```
select avg(a3)
from R_w_idx
where a2 < high and a2 > low
```

A.3 Sequential Join

```
select avg(R.a3)
from R, S
where R.a2 = S.a1;
```

A.4 FinTime Aggregate

```
select Avg(ClosePrice)
from HistPrice hp
where Id in <string list>
and TradeDate > Date "2001-09-02"
and TradeDate < Date "2010-09-02";
```

B OProfile output

This appendix lists various OProfile post processing results. For clarity, certain details such as function addresses and function parameter lists have been removed.

B.1 Correlation between level 2 cache misses and functional unit stalls

The ten functions in 10% IRS attributing most to level 2 cache misses (`L2_LINES_IN`, left side) and functional unit stalls (`RESOURCE_STALLS`, right side).

```
30.293 hash_lru_t<...>::_replacement()
23.127 w_hash_t<>::lookup()
19.869 w_link_t::detach()
12.052 serial_t::operator=()
 3.583 hash_lru_t<...>::grab()
 1.954 zkeyed_p::rec()
 1.302 file_p::next_slot()
 0.977 btrec_t::set()
 0.651 smutex_t::release()
 0.651 btree_p::search()
```

```
21.870 w_hash_t<...>::lookup()
19.714 w_link_t::detach()
19.295 hash_lru_t<>::_replacement()
 9.096 serial_t::operator=()
 7.579 hash_lru_t<...>::grab()
 3.922 lid_m::lookup()
 2.518 w_link_t::attach()
 2.438 file_p::next_slot()
 1.899 lid_m::lid_entry_t::_init_id()
 1.681 zkeyed_p::rec()
```

B.2 Correlation between level 1 instruction cache misses and branch mispredictions.

The ten functions in 10% SRS attributing most to level 1 instruction cache misses (L2_IFETCH, left side) and branch mispredictions (BR_MSP_RETIRED, right side).

```
6.38642 pin_i::_pin()
6.30558 XxxShoreFileRelation::FScanCursorInfo::NextRecord()
5.57801 bf_m::_fix()
4.28456 bf_core_m::find()
4.28456 prologue_rc_t::prologue_rc_t()
4.28456 scan_file_i::_next()
4.28456 XxxStorageManager::GetObjectPreAlloc()
2.82943 pin_i::_repin()
2.82943 XxxShoreFileRelation::NextItem1()
2.74859 XxxRelBooleanPlan::ExprEval()
```

```
11.0048 XxxShoreFileRelation::FScanCursorInfo::NextRecord()
 7.6555 bf_core_m::find()
 7.6555 scan_file_i::_next()
 6.2201 bf_m::_fix()
 5.74163 pin_i::_repin()
 5.26316 XxxStorageManager::GetObjectPreAlloc()
 5.26316 XxxRelBooleanPlan::ExprEval()
 4.30622 pin_i::repin()
 3.34928 XxxShoreFileRelation::NextItem1()
 2.87081 XxxRelBooleanPlan::Evaluate()
```

B.3 Executed instructions for 10% IRS

The ten functions in 10% IRS executing most instructions (INST_RETIRED)

```
15.2384     hash_lru_t<...>::grab()
13.0088     hash_lru_t<...>::_replacement()
8.83203     lid_m::lookup()
8.22396     w_hash_t<...>::lookup()
6.39641     file_p::get_rec()
4.87456     file_p::next_slot()
3.65177     serial_t::operator=()
3.45572     lid_m::_add_to_cache()
2.9274      w_link_t::detach()
2.68815     smutex_t::release()
```

B.4 Executed instructions for PNL SJ and SM SJ

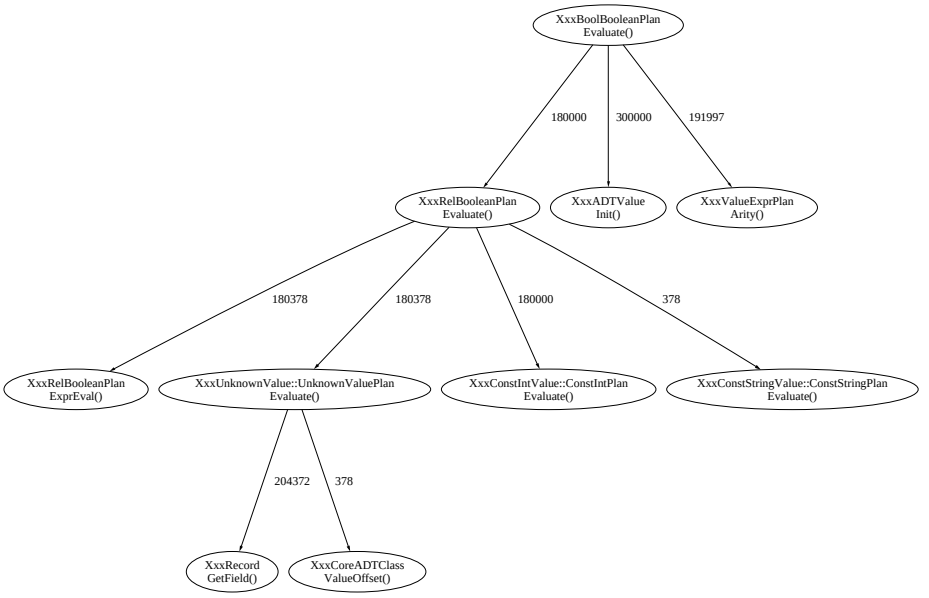
The ten functions in PNL SJ (left) and SM SJ (right) executing most instructions (INST_RETIRED)

```
16.421   XxxRecord::GetField()
12.7904  XxxRelBooleanPlan::Evaluate()
11.0822  RelPNLPlanOp::checkFinal()
10.9391  RelNLPlanOp::GetNextRecord()
10.0269  XxxBooleanExpressionPlan::MakeConjunction()
 9.09951 XxxUnknownValue::UnknownValuePlan::Evaluate()
 9.00509 RelPNLPlanOp::getNextItem()
 5.94438 XxxRelBooleanPlan::ExprEval()
 5.86745 RelPNLPlanOp::reclaimCurrent()
 3.53725 XxxBooleanExpressionPlan::MakeConjunction()
```

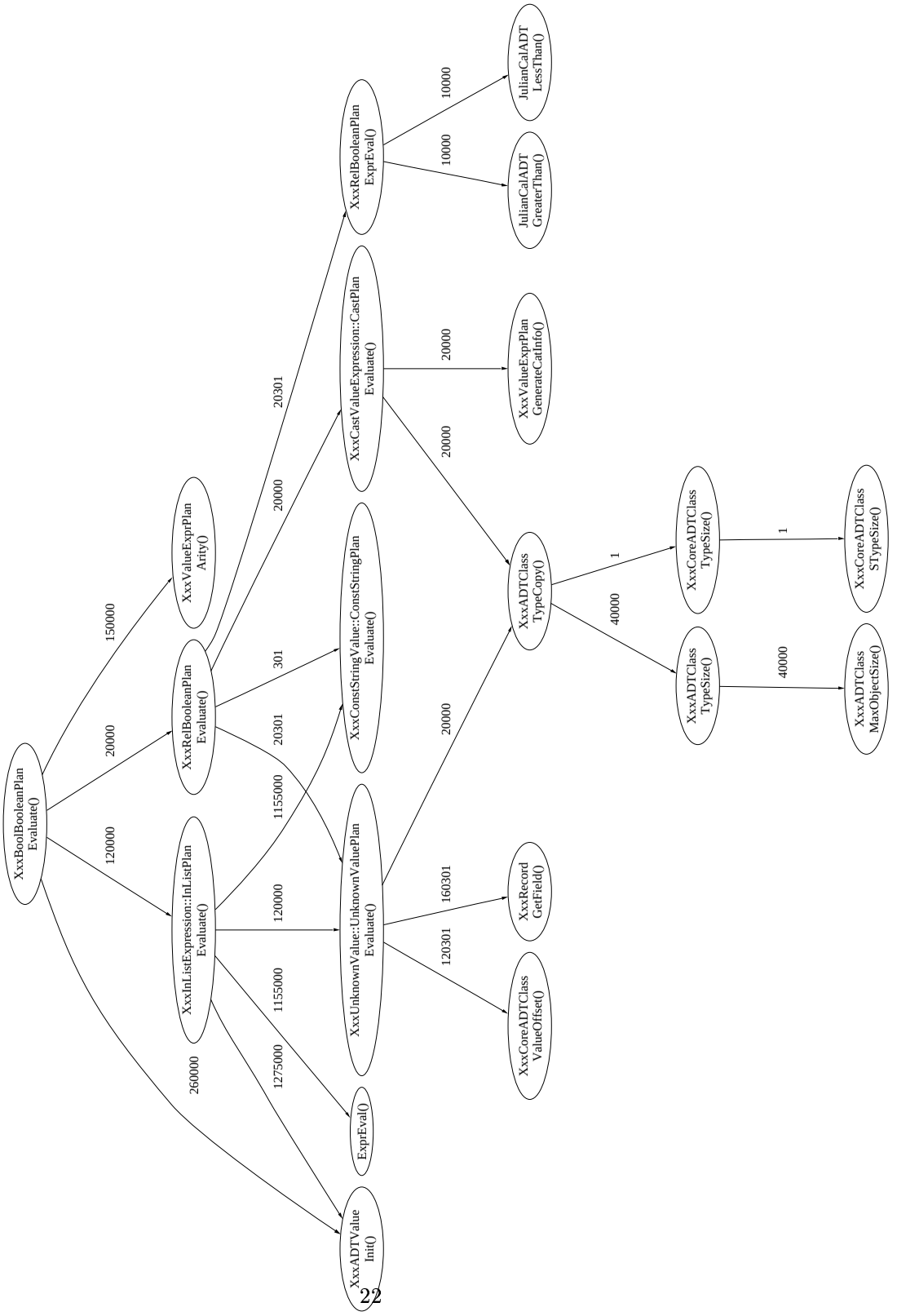
```
20.7669 run_mgr::_KeyCmp()
 5.25183 bf_m::is_bf_page()
 4.55581 pin_i::unpin()
 4.23943 pin_i::~pin_i()
 4.08757 skey_t::ptr()
 3.89775 pin_i::_init_constructor()
 1.97418 sort_keys_t::int4_cmp()
 1.78436 histoid_t::__find_page()
 1.72108 skey_t::contig_length()
 1.69577 run_mgr::_rec_in_run()
```

C Call Graphs

C.1 SRS Expression Evaluation



C.2 FTA Expression Evaluation



References

[1] Thomas Ball and James R. Larus. Using Paths to Measure, Explain, and Enhance Progran Behaviour. *IEEE Computer*, 33(7):57–65, July 2000.

[2] Anastassia Ailamaki. *Architecture-Conscious Database Systems.* PhD thesis, University of Wisconsin – Madison, 2000.

[3] Predator web site. URL `http://www.distlab.dk/predator`.

[4] James R. Larus. Whole Program Paths. In *Proceedings of the SIGPLAN 99 Conference on Programming Languages Design and Implementation*, 1999.

[5] Jay Fenlason and Richard Stallman. *GNU Gprof –The GNU Profiler.* URL `http:://www.gnu.org/manual/gprof`.

[6] Graphviz – open source graph drawing software. URL `http://www.research.att.com/sw/tools/graphviz`.

[7] Kevin Dowd and Charles Severance. *High Performance Computing.* O'Reilly, 2. edition, 1998.

[8] Mark Brehob, Travis Doom, Richard Enbody, William H. Moore, Sherry Q. Moore, Ron Sass, and Charles Severance. Beyond RISC – The Post-RISC Architecture. URL `http://www.netfact.com/crs/papers/postrisc2/`.

[9] Sandpile.org. URL `http://www.sandpile.org`.

[10] Sofus Mortensen. Refining the pure-C cost model. Master's thesis, University of Copenhagen, 2001.

[11] Yong Luo and Kirk W. Cameron. Instruction-level Characterization of Scientific Computing Applications Using Hardware Performance Counters. In *Proceedings of the Workload Characterization: Methodology and Case Studies*, 1998.

[12] Dileep Bhandarkar and Jason Ding. Performance Characterization of the Pentium Pro Processor. In *Proceedings of the Third International Symposium on High Performance Computer Architecture*, pages 288–297, feb 1997.

[13] Eletronic communication with A. Ailamaki, January 2002.

[14] LMbench - Tools for Performance Analysis. URL `http://www.bitmover.com/lmbench`.

[15] Rabbit – A Performance Counters Library for Intel/AMD Processors and Linux. URL `http://www.scl.ameslab.gov/Projects/Rabbit`.

[16] PAPI – Performance Application Programming Interface. URL `http://icl.cs.utk.edu/projects/papi`.

[17] Jack Dongarra, Kevin London, Shirley Moore, Phil Mucci, and Dan Terpstra. Using PAPI for Hardware Performance Monitoring on Linux Systems. In *Proceedings of the Conference on Linux Clusters: The HPC Revolution*, 2001.

[18] Oprofile. URL http://oprofile.sourceforge.net.

[19] *The IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*. Intel Corporation. URL http://developer.intel.com/design/pentium4/manuals/245472.htm.

[20] Eletronic communication with A. Glew, March 2002.

[21] Tom's Hardware Guide: Intel's new Pentium 4. URL http://www6.tomshardware.com/cpu/00q4/001120/index.html.

[22] Shore – A High-Performance, Scalable, Persistent Object Repository. URL http://www.cs.wisc.edu/shore.

[23] Kaippallimalil J. Jacob and Dennis Shasha. FinTime – a financial time series benchmark. URL http://www.cs.nyu.edu/cs/faculty/shasha/fintime.html.