Optimization of Instruction Fetch for Decision Support Workloads*

Alex Ramirez, Josep Ll. Larriba-Pey,
Carlos Navarro, Xavi Serrano, Mateo Valero
Universitat Politecnica de Catalunya
Jordi Girona 1–3, D6
08034 Barcelona (Spain)
{aramirez,larri,cnavarro,mateo}@ac.upc.es

Josep Torrellas
3314 Digital Computer Laboratory
1304 West Springfield Avenue
Urbana, IL 61801 (USA)
torrella@cs.uiuc.edu

Abstract

Instruction fetch bandwidth is feared to be a major limiting factor to the performance of future wide-issue aggressive superscalars.

In this paper, we focus on Database applications running Decision Support workloads. We characterize the locality patterns of ia database kernel and find frequently executed paths. Using this information, we propose an algorithm to lay out the basic blocks for improved I-fetch.

Our results show a miss reduction of 60-98% for realistic I-cache sizes and a doubling of the number of instructions executed between taken branches. As a consequence, we increase the fetch bandwith provided by an aggressive sequential fetch unit from 5.8 for the original code to 10.6 using our proposed layout. Our software scheme combines well with hardware schemes like a Trace Cache providing up to 12.1 instruction per cycle, suggesting that commercial workloads may be amenable to the aggressive I-fetch of future superscalars.

1 Introduction

Future wide-issue superscalars are expected to demand a high instruction bandwidth to satisfy their execution requirements. This will put pressure on the fetch unit and has raised concerns that instruction fetch bandwidth may be a major limiting factor to the performance of aggressive processors. Consequently, it is crucial to develop techniques to increase the number of useful instructions per cycle provided to the processor.

The number of useful instructions per cycle provided by the fetch unit is broadly determined by three factors: the branch prediction accuracy, the cache hit rate and the number of instructions provided by the fetch unit for each access. Clearly, many things can go wrong. Branch mispredictions cause the fetch engine to provide wrong-path instructions to the processor. Instruction cache misses stall the fetch engine, interrupting the supply of instructions to the processor. Finally, the execution of non-contiguous basic blocks prevents the fetch unit from providing a full width of instructions.

Much work has been done in the past to address these problems. Branch effects have been addressed with techniques to improve the branch prediction accuracy [10] and to predict multiple branches per cycle [20, 24]. Instruction cache misses have been addressed with software and hardware techniques. Software solutions include code reordering based on procedure placement [7, 6] or basic block mapping, either procedure oriented [16] or using a global scope [8, 21]. Hardware solutions include set associative caches, hardware prefetching, victim caches and other techniques. Finally, the number of instructions provided by the fetch unit each cycle can also be improved with software or hardware techniques. Software solutions include trace scheduling [4], and superblock scheduling [9]. Hardware solutions include branch address caches [24], collapsing buffers [2] and trace caches [5, 19].

While all these techniques have vastly improved the performance of superscalar I-fetch units, they have been largely focused and evaluated on engineering workloads. Unfortunately, there is growing evidence that popular commercial workloads provide a more challenging environment to aggressive instruction fetching.

Indeed, recent studies of database workload performance on current processors have given useful insight [1, 12, 14, 15, 18, 23]. These studies show that commercial work-

*This reserach has been supported by CICYT grant TIC-0511-98 (all UPC authors), the Generalitat de Catalunya grants ACI 97-26 (Josep Ll. Larriba-Pey and Josep Torrellas) and 1998FI-003060-26 (Alex Ramirez), the Commission for Cultural, Educational and Scientific Exchange between the United States of America and Spain (Josep Ll. Larriba-Pey, Josep Torrellas and Mateo Valero), the Spanish Ministry of Education grant PN98 43443683-1 (Carlos Navarro), NSF grant MIP-9619351 (Josep Torrellas), and CEPBA.

loads do not behave like other scientific and engineering codes. They execute fewer loops and have many procedure calls. This leads to large instruction footprints. The analysis, however, is not detailed enough to understand how to optimize them for improved I-fetch engine performance.

The work in this paper focuses on this issue. We proceed in three steps. First, we characterize the locality patterns of database kernel code and find frequently executed paths. The database kernel used is PostgreSQL [13]. Our data shows that there is significant locality and that the execution patterns are quite deterministic.

Second, we use this information to propose an algorithm to reorder the layout of the basic blocks in the database kernel for improved I-fetch. Finally, we evaluate our scheme via simulations. Our results show a miss reduction of 60-98% for realistic instruction cache sizes and a doubling of the number of instructions executed between taken branches to over 22. As a consequence, a 16 instruction wide sequential fetch unit using a perfect branch predictor increases the fetch bandwidth from 5.6 to 10.6 instructions per cycle when using our proposed code layout.

The software scheme that we propose combines well with hardware schemes like a Trace Cache. The fetch bandwith for a 256 entry Trace Cache improves from 8.6 to 12.1 when combined with our software approach. This suggests that commercial workloads may be amenable to the aggressive instruction fetch of future superscalars.

This paper is structured as follows. In Section 2, we give a detailed account of the internals of a database management system and compare PostgreSQL to it. In Section 4, we analyze the locality and determinism of the database execution. In Section 5, we describe the basic block reordering method that we propose. In Section 6 we give details on related work. In Section 7 we evaluate the performance of our method and compare it to other hardware and software techniques. Finally, in Section 8 we conclude and present guidelines for future work.

2 Database Management Systems

Database Management Systems are organized in different software modules. Those modules correspond to different functionalities to run queries, maintain the Database tables or use statistics on the Database data among others. Our interest focuses on those modules that take charge of running relational queries which are the most time consuming part of a RDBMS.

In order to run a relational query, it is necessary to perform a number of steps as shown in Figure 1. The query is specified by the user in a declarative language that determines what the user wants to know about the Database data. Nowadays, the Structured Query Language (SQL) is the standard declarative language for relational Databases [3].

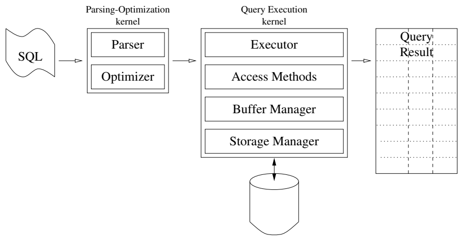

Figure 1. Steps required for the execution of an SQL query and all the RDBMS modules involved.

The SQL query is translated into an execution plan that will be processed by the Query Execution kernel. The query execution plan has the form of a tree, with nodes representing the different operations.

The task of the Parsing-optimization kernel is to check the grammatical correctness of the SQL expression and to generate the best execution plan for the given query in a specific computer and Database. While the importance of the Parsing-optimization module is paramount to generate a plan that executes fastest on a specific computer, the time employed to run it can be considered small compared to the total time spent in executing the query.

2.1 The Query Execution Kernel of a RDBMS

The Executor of a RDBMS (Figure 1) contains the routines that implement basic operations like Sequential Scan, Index Scan, Nested-Loop Join, Hash Join, Merge Join, Sort, Aggregate and Group. It also contains the routines that schedule the execution of those basic operations as described by the execution plan.

Just below the Executor, are the lower modules of the DBMS, the Access Methods, the Buffer Manager and the Storage Manager. This modular structure hides the different semantic data levels from the Executor. Now, we describe those semantic data levels and the communication data structures for the lower modules of the DBMS.

The tables of a Database are stored as files following a given logic structure. The Storage Manager is responsible for both managing those files and accessing them to provide file blocks to the Buffer Manager.

The Buffer Manager is responsible for managing the blocks stored in memory similarly to the way the OS Virtual Memory Manager does. The Buffer Manager provides memory blocks to the Access Methods Module.

The Access Methods of a RDBMS provide tuples to the Executor module. Depending on the organization of each table, the Access Methods will traverse the required index structures and plain database tables stored in the blocks managed by the Buffer Manager. Each DBMS will implement different Access Methods for its own index and database table structures.

2.2 PostgreSQL

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.

The structure of the server backend of PostgreSQL corresponds to that of a general DBMS that we explained before, shown in Figure 1.

The execution of a query in PostgreSQL is performed in a pipelined fashion. This means that each operation passes the result tuples to the parent operation in the execution plan as soon as they are generated, instead of processing their whole input and generating the full result set. This explains the lack of loops and the long code sequences found in the PostgreSQL and other DBMS kernels [15].

2.3 DSS Workloads and TPC-D

In this paper we use the Transaction Processing Performance Council benchmark for Decision Support Systems (TPC-D) [22] as a workload for our experiments. 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 [1, 23] 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 indices that can be used for the database.

3 Database Setup

We set up the Alpha version of the Postgres 6.3.2 database on Digital Unix v4.0 compiled with the $-O3$ optimization flags and the *cc* 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 indices and the other having Hash indices. Both databases have unique indices on all tables for the primary key attributes (those attributes that identify a tuple) and multiple entry indices for the foreign key attributes (those attributes which reference tuples on other tables).

4 Analysis of the Instruction Reference Patterns

Next, we examine the instruction reference patterns of the database focusing on locality issues by counting the number of times each basic block is executed, and recording all basic block transitions.

We select our Training set based on the importance of the Qualify and Scan operations, and the large number of misses attributed to the Access Methods and Buffer Manager modules [17].

The Training set used to obtain the profile information consists of executing queries 3, 4, 5, 6 and 9 on the Btree indexed database only. This set also includes queries with and without an extensive use of the Aggregate, Group and Sort operations, because they need all their children's results to be executed, which stops the normal pipelined execution of queries in the PostgreSQL database, and implies the storage of large temporary results. Furthermore, these operations store and access the temporary data without going through the Access Methods, which makes them somehow unique.

4.1 Reference locality

The data in Table 1 illustrates an important characteristic of the database code. Only 12.7% of the static instructions were referenced for an execution of the Training set, which means that the database contains large sections of code which are rarely accessed.

	Total	Executed	Percent
Procedures	6.813	1.340	19.7%
Basic blocks	127.426	15.415	12.1%
Instructions	593.884	75.183	12.7%

Table 1. Total number of static program elements, and the fracion of them that are actually used.

Figure 2 plots the percentage of the dynamic basic block references captured by a given number of static basic blocks. We observe that the 1000 most popular basic blocks (0.7% of the total count) accumulate 90% of the dynamic references, and that 2500 blocks gather as much as 99%.

This large concentration of the basic block references implies a large potential for exploiting locality. To further explore temporal locality, we counted the number of instructions that were executed between two consecutive invocations of a basic block.

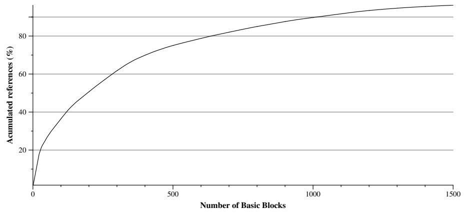

Figure 2. Percentage of total basic block references for a number of basic blocks.

If we consider the subset of the most popular basic blocks which concentrate 75% of the dynamic basic block references, we observe that they have a probability of 33% of being re-executed in less than 250 instructions, and as much as 19% of being referenced twice in less than 100 instructions.

4.2 Execution determinism

Next, we study how deterministic are the sequences of executed basic blocks.

We classify basic blocks in one of four kinds attending at how they affect the program flow. Fall-through basic blocks do not end with a branch instruction, so execution always continues on the next basic block. Branch basic blocks end with a conditional or unconditional branch. Subroutine call basic blocks end with a subroutine invocation or indirect jump, and may have many successors. Return basic blocks have many possible successors, as a subroutine may be referenced from several places.

BB Type	Static	Dynamic	Predictable
Fall-through	24.4%	22.4%	100%
Branch	42.4%	50.2%	59%
Subroutine call	8%	13.7%	100%
Subroutine return	25.2%	13.7%	100%

Table 2. Percentage of basic blocks executed by type, both static and dynamic. Percentage of the dynamic number that behaves in a fixed way.

Most basic block transitions have a very high, or very low probability of being executed. Looking at the numbers in Table 2, fall-through basic blocks, subroutine calls and returns compromise 50% of the dynamic basic blocks, and these usually have a fixed target. Also, 59% of the branch basic blocks (30% of the total basic blocks) tend to behave in a fixed way, either always taken or always not taken. Overall, 80% of the basic block transitions are predictable, which means that the executed sequences of basic blocks are fairly deterministic.

We have shown that there is a large concentration of references in a small set of very popular basic blocks and that there is substantial temporal locality to be exploited, as the most popular blocks are referenced every few instructions. Also, the execution paths are quite deterministic, allowing us to exploit spatial locality by mapping basic blocks executed sequentially in consecutive memory locations. This allows us to build basic block traces at compile time, arranging them carefully in memory to avoid conflict misses.

5 Method description

Instrumenting the database and running the Training set, we obtained a directed control flow graph with weighted edges.

To improve the fetch bandwidth, we build our basic block sequences placing in consecutive memory positions those basic blocks executed sequentially, maximizing the number of instructions executed between taken branches.

Also, to reduce the instruction cache miss rate, we will map the most frequently executed code sequences in a reserved area of the cache and other popular sequences close to other equally popular ones, reducing interference among them.

5.1 Seed selection

We first tried an automatic seed selection (*auto* selection). The list of seeds contains the entry points of all functions, in decreasing order of popularity. This selection tries to expose the maximum temporal locality, building first the sequences for the most popular functions.

The second seed selection (*ops* selection) was based on our knowledge of the database structure. We limited the list of seeds to the entry points of the Executor operations. This seed selection will obtain longer sequences than the first one, as most functions will be included halfway through the main sequence, as they are referenced by it. However, some important basic blocks may be left out, as they will be unreachable from the selected seeds, or some intermediate basic block will not pass the Exec or Branch thresholds.

Also, the sequences built this way will have lower temporal locality as they include less frequently referenced basic blocks surrounding the most popular ones.

5.2 Sequence building

Using the weighted graph obtained running the training set, and starting from the selected seeds, we implement a greedy algorithm to build our basic block traces targeting an increase in the code sequentiality. Given a basic block, the algorithm follows the most frequently executed path out of it. This implies visiting a subroutine called by the basic block, or following the control transfer with the highest probability of being used. All the other valid transitions from the basic block are noted for future examination.

For this algorithm we use two parameters called *Exec Threshold*, and *Branch Threshold*. The trace building algorithm stops when all the successor basic blocks have been visited or have a weight lower than the Exec Threshold, or all the outgoing arcs have a branch probability less than the Branch Threshold. In that case, we start again from the next acceptable transition, as we noted before, building secondary execution paths for the same seed. Once all basic blocks reachable from the given seed have been included in the main or secondary sequences, we proceed to the next seed.

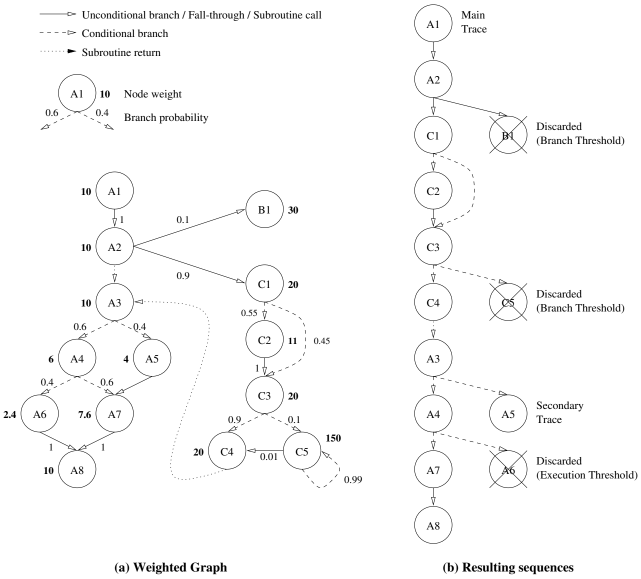

Figure 3. Trace building example.

Figure 3.a shows an example of the weighted graph and Figure 3.b shows the resulting sequences. We use an ExecThresh of 4 and a BranchThresh of 0.4. Starting from seed $A1$ and following the most likely outgoing edge from each basic block we build the sequence $A1 \rightarrow A8$ (Figure 3.b). The transitions to $B1$ and $C5$ are discarded due to the Branch Threshold. We noted that the transition from $A3$ to $A5$ is a valid transition, so we start a secondary trace with $A5$, but all its successors have been already visited, so the sequence ends there. We do not start a secondary trace from $A6$ because it has a weight lower than the Exec Threshold.

5.3 Sequence Mapping

Figure 4 shows the sequence mapping scheme. We define a logical array of caches, equal in size and address alignment to the physical cache. The sequences found in the first pass of the algorithm described in Section 5.2 are mapped from the start of the logical cache array. Then, we place the rest of the sequences in order, one pass at a time, keeping the area used by the sequences in the first pass free of code in all logical caches. This way, the first sequences will not be replaced from the cache by any other code, and so will be free of interference. We call this area the Conflict Free Area (CFA), and derives directly form the *SelfConf-Free* area proposed in [21].

The size of this CFA is determined by the Exec and Branch Thresholds used for the first pass of our sequence building algorithm.

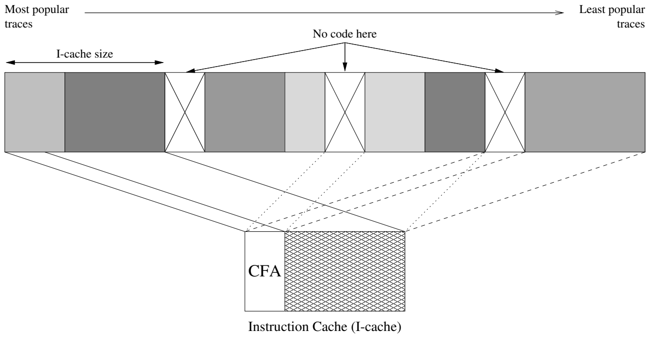

Figure 4. Sequence mapping into a direct mapped cache

When all the sequences have been mapped in the cache, we map all the remaining basic blocks in order, this time filling the entire address space. This rarely executed code is expected not to produce many conflicts with the sequences placed in the CFA.

6 Related Work

The use of profile-directed code reordering to reduce the instruction cache miss rate is not new. Hwu and Chang [8] use function inline expansion, and group into traces those basic blocks which tend to execute sequentially. Then, they map these traces in the cache so that functions which are executed close to each other are placed in the same page.

Pettis & Hansen [16] propose a reordering of the procedures in a program and the basic blocks within each procedure. Their aim is to minimize the conflicts between the most frequently used functions, placing functions which reference each other close in memory. They also reorder the basic blocks in a procedure, moving unused basic blocks to the bottom of the function code or even splitting the procedures in two, moving away the unused basic blocks. Their algorithm does not consider the target cache geometry.

Gloy *et al.* [6] extend the Pettis & Hansen algorithm at the procedure level to consider the temporal relationship between procedures in addition to the target cache information and the size of each procedure. Hashemi *et al* [7] and Kalamaitianos *et al* [11] use a cache line coloring algorithm inspired in the register coloring technique to map procedures so that the resulting number of conflicts is minimized.

Torrellas *et al* [21] propose a basic block reordering algorithm for Operating System code, running on a very conservative vector processor. They map the code in the form of sequences of basic blocks spanning several functions, and keep a Conflict Free Area for the most frequently referenced basic blocks.

Our target goes beyond an instruction cache miss rate reduction. Targeting an increase in the number of sequential instructions executed, we combine the different approaches and map out basic block traces as a unit, keeping whole sequences in the *conflict free area* instead of preserving individual basic blocks in it.

Techniques like trace scheduling [4] and superblock scheduling [9], use static branch prediction to define logical groups of basic blocks, enlarging the scope of the compiler to schedule instruction issue by moving individual instructions across basic blocks.

For more aggressive processors, the width of instructions provided to the processor becomes an important issue. Techniques like the Branch Address Cache [24], the Collapsing Buffer [2] and the Trace Cache [19, 5] approach the problem of fetching multiple, non-contiguous basic blocks each cycle. Both the Branch Address Cache and the Collapsing Buffer access non-consecutive cache lines from an interleaved i-cache each cycle and then merge the required instructions from each accessed line.

The Trace Cache does not require fetching of non-consecutive basic blocks from the i-cache. This is done by storing the dynamically constructed sequences of instructions in a special purpose *trace cache*. If a fetch request corresponds to one of the sequences stored in the trace cache, this sequence is passed to the decode unit, effectively fetching multiple basic blocks in a single cycle. On a trace cache miss, fetching proceeds from the conventional i-cache and a new sequence is created and stored in the trace cache.

Our approach targets an increase in the number of instructions provided by the instruction cache, reducing the need of the trace cache, and providing a better back-up mechanism in case of a trace cache miss.

7 Method Evaluation

As we said before, our method is based on profile information. In order to evaluate our technique we used a different set of queries for the Test and Training sets. The Test set used to obtain the simulation results consists of queries 2, 3, 4, 6, 11, 12, 13, 14, 15 and 17, executed on both the Btree and the Hash indexed databases (see Section 3). For both the Training and the Test sets, all queries were run to completion.

We compared the results of both our automatic code layout (*auto* layout) and the experience based layout (*ops* layout) with those obtained by the method proposed by Pettis & Hansen (*P&H* layout) and the work of Torrellas *et al* (*Torr* layout).

7.1 Simulation setup

We did not generate a new executable with the proposed code layouts. Instead, we generated a new address for each basic block, feeding the simulators with this faked address instead of the original PC. The code was not modified in any way, so all basic blocks have the same size for all the examined code layouts.

The fetch unit used in our simulations corresponds to the SEQ.3 fetch unit described in [19]. This fetch unit accesses two consecutive cache lines, and provides the instructions from the fetch address up to the first taken branch, or up to a maximum of three branches, or 16 instructions, whichever comes first.

We used perfect branch prediction to examine the performance limit of the examined techniques, avoiding interference due to branch and target mispredictions.

7.2 Instruction cache miss rate

Table 3 shows the miss rate for the sequential fetch unit and the different code layouts. We present results for different i-cache and CFA sizes. We also evaluated a 2-way set associative cache and the addition of a 16-line fully-associative victim cache. The numbers given are in terms of i-cache misses per instruction executed.

i-cache	Code layout					Cache	
/ CFA	orig	P&H	Torr	auto	ops	2-way	victim
8/2	6.5	3.0	2.3	2.2	2.1	6.1	5.6
8/4	–	–	2.9	4.2	2.9	–	–
8/6	–	–	3.1	2.3	5.2	–	–
16/4	4.0	1.1	0.9	0.8	0.7	2.6	3.4
16/8	–	–	0.7	0.8	0.6	–	–
16/12	–	–	0.8	0.8	1.0	–	–
32/4	2.7	0.3	0.2	0.3	0.2	1.2	1.6
32/8	–	–	0.2	0.4	0.2	–	–
32/16	–	–	0.3	0.2	0.1	–	–
32/24	–	–	0.2	0.3	0.2	–	–
64/8	1.4	0.09	0.05	0.07	0.04	0.3	0.4
64/16	–	–	0.14	0.08	0.05	–	–
64/24	–	–	0.02	0.03	0.03	–	–

Table 3. Instruction cache miss rate for the different i-cache and CFA sizes examined.

Our proposed layouts obtain similar results to the *Torr* layout, and always outperform the *P&H* layout. All the code layouts obtained better results than both the 2-way associative cache and the victim cache.

From the results shown, it is clear that the most important factor regarding the miss rate reduction offered by our technique is the size of the CFA.

Intuitively, an increase in the CFA size causes both positive and negative effects. On the one hand, a larger CFA shields more routines from self-interference, and, as a result, eliminates misses in those routines. On the other hand, however, less area is left for the rest of the routines, which will suffer more conflict misses. Once the CFA size reaches a certain value, the second effect will dominate. Also, once the CFA is able to satisfy most of the references, there is little point in further increasing it, as we will be taking out cache area that may be better used to avoid conflict misses in other code.

7.3 Fetch bandwidth

We compared fetch bandwith results with those of the basic Trace Cache described in [19]. We simulated a direct mapped Trace Cache of 256 entries (16KB). All kinds of branch instructions were counted against the 3 branch limit, including unconditional branches and subroutine calls and returns. We did not stop instruction fetch on indirect jumps as was done in [5, 19].

I-cache	Code layout					Trace cache	
/ CFA	orig	P&H	Torr	auto	ops	16KB	16KB+ops
Ideal	7.6	9.6	8.5–9.9	9.9	10.7	10.3	12.2
8/2	3.1	5.2	5.6	6.0	6.2	5.1	8.4
8/4	–	–	5.0	5.3	6.6	–	8.7
8/6	–	–	4.9	5.8	5.6	–	8.1
16/4	4.0	7.3	7.4	8.1	8.8	6.2	10.3
16/8	–	–	7.4	8.1	9.0	–	10.4
16/12	–	–	7.3	7.9	8.1	–	10.2
32/4	4.7	8.8	8.9	9.2	10.0	7.2	11.5
32/8	–	–	8.4	8.8	10.1	–	11.5
32/16	–	–	8.0	9.3	10.3	–	11.8
32/24	–	–	8.2	9.2	10.1	–	11.6
64/8	5.8	9.3	8.8	9.8	10.6	8.6	12.0
64/16	–	–	8.4	9.7	10.5	–	12.1
64/24	–	–	8.5	9.8	10.6	–	12.1

Table 4. Fetch bandwidth in instructions per cycle. The i-cache miss penalty is 5 cycles.

Table 4 shows the number of instructions per cycle provided by each setup.

The *Ideal* line corresponds to a perfect instruction cache. For the realistic i-cache setups, we applied a fixed miss penalty of 5 cycles. We did not count any miss penalty on a trace cache hit.

Looking at the *Ideal* fetch bandwidth provided by each technique, we observe that the *P&H* layout is very close to the *auto* reordering and the 16KB Trace Cache. The *ops* reordering has more potential bandwidth than any other technique except the combination of itself with the Trace Cache.

The method proposed by Torrellas *et al.* shows a variable behavior. The larger the CFA, the more basic blocks are included in the CFA. These basic blocks have been pulled out of their sequences to be included in the CFA, which breaks the sequential execution jumping in and out of the CFA.

Once the average number of cycles taken by each fetch request is considered, both the *auto* and the *ops* layouts become clearly better than any other technique. The Trace Cache could not remember all the executed sequences, and had to resort to sequential fetching 52% of the times, and the fetch unit could provide few instructions due to the lack of sequentiality in the original code.

Our proposed layout could use the whole memory space as a *software trace cache* to capture the most frequently executed sequences of code, and could provide more instructions per cycle using the statically stored traces. When using the *ops* layout, the fetch engine could provide more instructions per cycle, even on a Trace Cache miss, both due to the increased sequentiality of the code and to the reduced i-cache miss rate.

It is worth noting that a lower miss rate alone does not mean a higher fetch bandwidth, as can be observed comparing the results of the P&H and the Torr layouts for the larger caches. In order to improve the fetch bandwidth, both the i-cache miss rate and the sequentiality of code must be taken into account.

8 Conclusions and Future Work

Instruction fetch bandwidth is feared to be a major limiting factor to the performance of future wide-issue aggressive superscalars.

In this paper, we focus on Database applications running Decision Support workloads. We characterize the locality patterns of database kernel code and find frequently executed paths. Using this information, we propose an algorithm to lay out the basic blocks of the database kernel for improved fetch bandwith. This is achieved by both a reduction of the i-cache miss rate and an increase in the number of instructions executed between taken branches.

Our results show a miss reduction of 60-98% for realistic i-cache sizes, obtaining miss rates under 0.05% with a 64KB direct mapped cache. The proposed code layout also increases the number of instructions executed between taken branches from the 8.9 of the original code to 22.4. With this, a 16 instruction wide fetch unit could provide 10.6 instructions per cycle using our proposed code layout.

Improving only the i-cache miss rate or the potential fetch bandwidth provided is not enough. Both factors must be taken into account to provide optimal results.

A Trace Cache alone could not hold all the executed sequences, while our technique used all the memory space as a Software Trace Cache to statically store the most frequently executed traces. As a consequence, while the Trace Cache alone could only provide 8.6 inctructions per cycle, a combination of the Software-Hardware Trace Caches increased the result to 12.1.

We have shown that large first level i-caches can capture the working set of large applications like a DBMS. It is worth studying if the controlled use of code expanding techniques like function inlining and code replication can increase the potential fetch bandwidth provided by a sequential fetch unit while keeping the miss rate under control.

In the near future we plan to extend the proposed algorithm to automatize the process of selecting the thresholds and the seeds while obtaining results closer to the knowledge-based selection. Also, we will examine the effect of our technique on the IPC for a wider range of applications like OLTP workloads and the SPEC benckmark.

References

[1] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. *Proceedings of the 16th Annual Intl. Symposium on Computer Architecture*, pages 3–14, June 1998.

[2] T. Conte, K. Menezes, P. Mills, and B. Patell. Optimization of instruction fetch mechanism for high issue rates. *Proceedings of the 22th Annual Intl. Symposium on Computer Architecture*, pages 333–344, June 1995.

[3] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Benjamin Cummings, second edition, 1994.

[4] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[5] D. H. Friendly, S. J. Patel, and Y. N. Patt. Alternative fetch and issue techniques from the trace cache mechanism. *Proceedings of the 30th Anual ACM/IEEE Intl. Symposium on Microarchitecture*, Dec. 1997.

[6] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder. Procedure placement using temporal ordering information. *Proceedings of the 30th Anual ACM/IEEE Intl. Symposium on Microarchitecture*, pages 303–313, Dec. 1997.

[7] A. H. Hashemi, D. R. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. *Proc. ACM SIGPLAN'97 Conf. on Programming Languaje Design and Implementation*, pages 171–182, June 1997.

[8] W.-M. Hwu and P. P. Chang. Achieving high instruction cache performance with an optimizing compiler. *Proceedings of the 16th Annual Intl. Symposium on Computer Architecture*, pages 242–251, June 1989.

[9] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Water, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. K. G. E. Haab, J. G. Hold, and D. M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *Journal on Supercomputing*, (7):9–50, 1993.

[10] T. Juan, S. Sanjeevan, and J. J. Navarro. Dynamic history-length fitting: A thrid level of adaptivity for branch prediction. *Proceedings of the 25th Annual Intl. Symposium on Computer Architecture*, pages 155–166, June 1998.

[11] J. Kalamaitianos and D. R. Kaeli. Temporal-based procedure reordering for improved instruction cache performance. *Proceedings of the 4th Intl. Conference on High Performance Computer Architecture*, Feb. 1998.

[12] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance characterization of a quad pentium pro smp using oltp workloads. *Proceedings of the 25th Annual Intl. Symposium on Computer Architecture*, pages 15–26, June 1998.

[13] M. S. G. Kemnitz. The postgres next generation database management system. *Communications of the ACM*, Oct. 1991.

[14] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. *Proceedings of the 25th Annual Intl. Symposium on Computer Architecture*, pages 39–50, June 1998.

[15] A. M. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. *Proceedings of the 6th Intl. Conference on Architectural Support for Programming Languajes and Operating Systems*, pages 145–156, Oct. 1994.

[16] K. Pettis and R. C. Hansen. Profile guided code positioning. *Proc. ACM SIGPLAN'99 Conf. on Programming Languaje Design and Implementation*, pages 16–27, June 1990.

[17] A. Ramffez, J. L. Larriba-Pey, C. Navarro, X. Serrano, J. Torrellas, and M. Valero. Code reordering of decision support systems for optimized instruction fetch. Technical Report UPC-DAC-1998-56, Universitat Politecnica de Catalunya, Dec. 1998.

[18] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of database workloads on shared-memory systems with out of order processors. *Proceedings of the 8th Intl. Conference on Architectural Support for Programming Languajes and Operating Systems*, Oct. 1998.

[19] E. Rottenberg, S. Benett, and J. E. Smith. Trace cache: a low latency aprroach to high bandwith instruction fetching. *Proceedings of the 29th Anual ACM/IEEE Intl. Symposium on Microarchitecture*, pages 24–34, Dec. 1996.

[20] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block ahead branch predictors. *Proceedings of the 7th Intl. Conference on Architectural Support for Programming Languajes and Operating Systems*, Oct. 1996.

[21] J. Torrellas, C. Xia, and R. Daigle. Optimizing instruction cache performance for operating system intensive workloads. *Proceedings of the 1st Intl. Conference on High Performance Computer Architecture*, pages 360–369, Jan. 1995.

[22] T. P. P. C. (TPC). Tpc benchmark d (decision support). Standard Specification, Revision 1. 2. 3, 1993–1997.

[23] P. Trancoso, J. L. Larriba-Pey, Z. Zhang, and J. Torrellas. The memory performance of dss commercial workloads in shared-memory multiprocessors. *Proceedings of the 3rd Intl. Conference on High Performance Computer Architecture*, Feb. 1997.

[24] T. Y. Yeh, D. T. Marr, and Y. N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. *Proceedings of the 7th Intl. Conference on Supercomputing*, pages 67–76, July 1993.