Optimizing I-Cache Utilization for Database Systems using Call Graph Prefetching

Murali Annavaram, Jignesh M. Patel, Edward S. Davidson
Electrical Engineering and Computer Science Department
The University of Michigan, Ann Arbor
{annavara, jignesh, davidson}@eecs.umich.edu

Abstract

With the current technological trends of cheaper and larger memory, in the near future, most data sets in database servers will be resident in main memory. In this configuration, the performance bottleneck is likely to be the gap between the processing speed of the CPU and the memory access latency. To reduce the performance penalty associated with accessing main memory, modern processors buffer recently used instructions and data in processor caches that are much cheaper to access than main memory. Unfortunately, database applications have large instruction footprints and data sets and do not use these processor caches effectively. This poor cache utilization reduces the performance of the database system significantly. In this paper, we propose and evaluate a technique called Call Graph Prefetching (CGP) that allows a database system to effectively utilize the processor instruction cache. CGP profiles the original database binary using a sample workload, and produces a new binary that uses the I-cache more effectively. Our experiments show that CGP can speedup a database system by 42%. The strength of CGP is that it does not require recoding the database system, and also works with user-defined functions and data types in an ORDBMS, for which rewriting the database code may not be an option.

1 Introduction

In the last five decades, we have witnessed an explosive growth in computing technology with microprocessors becoming faster and cheaper, and memory becoming denser and cheaper. With new fabrication technologies on the horizon, this trend seems likely to continue for some time. Consequently, database servers in the near future will have large main-memory configuration, and most data sets will be resident in main memory [BBC+98]. Once a data set is in main memory, the performance of the database system is limited by how fast the memory sub-system can feed...
instructions and data to the processor. In most modern processors the time to access a block in main memory is nearly 100 cycles; on a 4-wide superscalar processor that can execute up to 4 instructions every cycle, 400 instructions can be executed in the time it takes to complete one memory access. To use the processor more effectively, a small set-associative cache that is accessible in a few processor cycles is used to buffer both instructions and data. When the processor needs either instructions or data it first looks up the appropriate cache (the instruction cache or the data cache), and on a cache miss it tries to find the memory block in the next level of the memory hierarchy.

On a data cache miss, instead of stalling while the data block is being fetched, an out-of-order processor will try to execute instructions that are not dependent on the data being fetched. However, since the lower levels of the memory hierarchy are significantly more expensive to access even an aggressive out-of-order processor cannot completely mask the cache miss latency. The resulting processor stalls can have a serious impact on the performance of the database system. Recent studies have shown that current database systems with their large code footprint and large data sets make poor utilization of both the data and the instruction cache [ADHW99, BRK98, LBE+98, NBC+94, SKN94]. Thus the key challenge in improving the performance of memory bound database systems is improving the utilization of these caches by reducing their cache miss rates.

In this paper, we propose and evaluate Call Graph Prefetching (CGP), a novel instruction prefetching technique that analyzes the call graph of a database system, and inserts prefetch instructions that reduce the number of instruction cache (I-cache) misses. CGP recognizes that function calls often cause I-cache misses due to control flow changes, and examines the call graph to decide if it should insert prefetch instruction to reduce these cache misses. To help make this decision, CGP produces profile information for the call graph by running the database binary with a sample workload that is provided by the database system. Although CGP uses a profile workload to determine
which functions to prefetch, CGP is highly effective even when the actual query workload differs significantly from the profile workload. This remarkable stability exists because database systems are built using a layered software architecture and queries often end up repeatedly calling the same set of functions in the database code. For example, regardless of whether a query has a join or a selection operation, both queries will call the same function in the storage manager to fetch tuples from a file. We evaluate the effectiveness of CGP using a database workload that consists of a subset of the Wisconsin benchmark and TPC-H queries. Our performance evaluations show that CGP can speedup a database system by 42% over a baseline system that is already optimized for I-cache performance.

We note that CGP does not require re-writing the source code of the database system, and also works with user-defined data types and functions in an ORDBMS, for which the source code may not be available.

Although both instruction and data cache misses can have a significant impact on the overall performance, this paper focuses only on the instruction cache misses. Instruction cache misses are harder to mask as they serialize the program execution by stalling the issuing of instructions in the processor pipeline until the cache miss is serviced. Nevertheless, we believe that data cache misses are also important, and any techniques for reducing data stalls will further improve the performance of the database system.

The rest of this paper is organized as follows. Section 2 describes previous work that is related to CGP. Section 3 presents an overview of CGP and describes the algorithm used to generate a new binary. Section 4 describes our evaluation methodology and the results of experiments that evaluate the effectiveness of CGP. Finally, the conclusions and future work are summarized in Section 5.
2 Related work and Terminology

Nyberg et al. [NBC+94] suggests that if data intensive applications use software assisted disk striping the performance bottleneck shifts from I/O response time to the CPU processing time. Boncz et al. [BRK98] also showed that the query execution time of data mining workloads with a large main memory buffer pool is memory bound rather than I/O bound. Shatdal et al. [SKN94] proposed cache-conscious performance tuning techniques for join and aggregation algorithms that improve the locality of the data accesses. These techniques reduce the data cache misses, where as CGP tries to reduce L-cache misses. CGP can work on top of these cache-conscious algorithms.

It is only recently that researchers have examined the performance impact of architectural features on DBMS [ADHW99, LBE+98, TLPZT97, EJK+96, RBH+95, CB94, MDB94]. Their results show that database applications have large instruction and data footprints and more distinctive branch behavior than the commonly used benchmarks in architectural studies (e.g. SPEC). Database applications have fewer loops and suffer from frequent context switches, causing significant increases in the instruction cache miss rates [Fra94]. Lo et al. [LBE+98] also showed that in OLTP workloads, the instruction cache miss rate is nearly three times the data cache miss rate. Ailamaki et al. [ADHW99] analyzed three commercial DBMSs on a Xeon processor and showed that TPC-D queries spend 20% of their execution time on branch misprediction stalls and 20% on L1 instruction cache miss stalls (even though the Xeon processor uses special instruction prefetching hardware).

Researchers have proposed several schemes to improve instruction cache performance. Pettis and Hansen [PH90] proposed a code layout algorithm which uses profile guided feedback information to layout the basic blocks that are on the most commonly occurring control flow path contiguously. Romer et al. [RVL+97] implemented Pettis and Hansen’s code layout algorithm using Etch tool and showed performance improvements for Win32 binaries.
In this paper we used OM [SW92] which implements modified Pettis and Hansen’s algorithm to do feedback directed code layout. This algorithm is discussed further in Section 4.2. Our results show that OM improves the performance of a highly optimized binary (C++ -O5 optimization level) by 48%. Using CGP in addition to OM further improves the performance by 42%, yielding a 2.1X speedup over the O5 optimized binary.

In spite of the overwhelming proof that database systems suffer from high instruction cache miss rates, very little research has focused on techniques to reduce this penalty. To the best of our knowledge, the cooperative prefetching work by Luk and Mowry [LM98] is the only study that addresses this problem. Luk and Mowry use compiler and hardware support to insert prefetch instructions before branches to prefetch the branch targets. In addition, they require special hardware filters to dynamically reduce prefetch traffic. Their work requires special hardware support while CGP does not require any hardware.

2.1 Terminology

Before we begin our discussion, we briefly introduce some prefetching terminology. Prefetch Accuracy is the ratio of the number of cache lines prefetched into the cache to the number of cache lines used. A prefetched cache line is useless if the cache line is either never used or replaced before it is used. Prefetch Accuracy should be high so that prefetch requests bring in useful instructions and not waste the limited memory bus bandwidth bringing in useless cache lines. Prefetch latency is the time from when a request for a cache line is issued to the time when the line is brought into the cache. Ideally a prefetch should be issued early enough so that the prefetched item arrives in the cache just before it is needed. If the prefetches are issued too early, prefetched items will be replaced before they actually get used.
3 Call Graph Prefetching (CGP)

DBMSs are commonly built using a layered software architecture with the storage manager being the bottom most layer. (In some specific implementations the storage manager itself may be organized in layers.) Relational operators that implement algorithms for join, aggregation etc., are typically built on top of the storage manager. The query optimizer and the query scheduler are then built on top of the operator layer. Each layer in this modular architecture provides a set of well-defined entry points and hides the implementation details to improve the portability and maintainability of the entire software. Each of these entry points makes a sequence of function calls that are transparent to its caller. Often this sequence of function calls can be predicted with great accuracy. CGP makes use of this predictability and prefetches instructions from a procedure that is most likely to be executed next.

We introduce CGP with the following pedagogical example. Figure 1 shows a segment of a call graph for adding a record to a file in the SHORE [CDF+94] storage manager. SHORE is a storage manager that provides storage volumes, B+-trees, R*-trees, concurrency control and transaction management. In this example, Create_rec calls Find_page_in_buffer to check if the relation into which the record is being added is already pinned in the buffer pool. If the page is not already in the buffer pool the Getpage_from_disk function is invoked to bring the page from the disk into the main memory buffer pool. This page is then locked using the Lock_page routine and updated using the Update_page.

The Create_rec function is the entry point provided by the storage manager to create a record, and can be invoked by a number of relational operators, including an insert operator, a bulk load operator, a join operator (to create temporary partitions or sorted runs), and an aggregate operator. Although it is difficult to predict calls to Create_rec, once it is invoked Find_page_in_buffer is always
the next function to be executed. With a large buffer pool size the page that is being updated will usually be pinned. Hence Lock_page and Update_page will often be the next set of functions that are invoked. CGP capitalizes on this predictability by first inserting prefetch instructions into Create_rec for prefetching the instructions needed for executing Find_page_in_buffer, and then inserting prefetch instructions to prefetch Update_page.

Figure 1: Example of a Call Graph for Create_rec function

3.1 Call Graph Prefetching Algorithm

We now describe the algorithm used for CGP. CGP algorithm reads the DBMS binary and builds a directed call graph. Each function in the binary corresponds to a node in the graph, and a directed edge is drawn from a caller function to the callee function with an initial edge label as 0. Next CGP uses a sample database workload and runs the DBMS binary to warmup the buffer pool. Then it reruns the workload and annotates the call graph as follows: Each edge in the call graph is labeled with a sequence number that records the order in which the caller has called the callee functions during the first invocation of the caller function.

It is possible that multiple invocations of the same function can trigger a different sequence of function calls each time. However, CGP labels the call graph using the information gathered
during the first invocation of the function. Although there are alternative approaches for labeling the edges, such as using the most frequent sequence of invocations, the approach used by CGP simplifies the edge labeling process. In future we plan to evaluate some of the alternative schemes.

Figure 2 shows the labeled directed call graph for the example described in Figure 1. Initially all the outgoing edges are labeled as 0. During the profile workload execution when CreateRec function is first invoked it makes a call to Find_page_in_buffer followed by a call to Update_page. The outgoing edges are labeled 1 and 2 to reflect the order in which the function calls were made. When Find_page_in_buffer is first invoked it makes a call to Lock_page and hence the corresponding outgoing edge is labeled 1. Find_page_in_buffer did not call Getpage_from_disk and hence the label for the corresponding outgoing edge retains the initial value of 0.

![Directed Call Graph with the edge labels from the profile execution.](image)

CGP algorithm uses the labeled call graph to insert prefetch instructions into the binary. For every node in the call graph CGP inserts instructions to prefetch the callee functions that are connected with an edge label greater than 0. The callee functions are prefetched in the ascending order of the edge labels. CGP inserts instructions to prefetch the first callee function in the first basic block of the caller. Prefetch instructions are inserted for the second callee functions immediately after the call to the first callee function and this process is repeated recursively.

This algorithm inserts prefetch instruction to prefetch the first \( N \) cache lines of a function. Since only the first \( N \) cache lines of a callee function are prefetched in the caller function the rest of the callee function is prefetched in the callee function itself by inserting instructions to prefetch the next \( N \) cache lines at equal intervals.
Since the CGP algorithm uses profile information, prefetching an entire function may waste processor resources if the prefetched function is not invoked during the actual query execution. Moreover, prefetching a large function into the instruction cache can pollute the cache by replacing an existing cache line that may be needed sooner than the prefetched line. Hence the prefetch algorithm only prefetches a few \( N \) cache lines, where \( N \) is a parameter that is based on the cache size, line size and the I-cache miss latency.

![Diagram](image)

**Figure 3:** *Create_rec* function after applying CGP algorithm. The new prefetch instructions are indicated by *.

Figure 3 shows the code generated after applying CGP algorithm to the call graph of *Create_rec*. Prefetch instructions are inserted at the beginning of *Create_rec* to prefetch *Find_page_in_buffer*. Since *Update_page* is the next function called after returning from *Find_page_in_buffer*, a prefetch instruction is inserted for this function after the call to *Find_page_in_buffer*. Instruction are inserted in *Find_page_in_buffer* to prefetch *Lock_page*. *Getpage_from_disk* is not prefetched because the edge label is 0.

Since only the first \( N \) cache lines of *Find_page_in_buffer* are prefetched from *Create_rec*, prefetch instructions are also inserted at equal intervals in *Find_page_in_buffer* to prefetch the next \( N \) cache lines. Both *Lock_page* and *Update_page* are small functions and the first \( N \) cache lines prefetched
from their caller functions will be sufficient to prefetch the entire function; hence, no instructions are inserted to prefetch the next $N$ cache lines in these functions.

### 3.2 Considerations for CGP Implementation

Since CGP inserts prefetch instructions into the binary, it requires that the instruction set include an opcode to prefetch instructions into the instruction cache. Some of the current architectures, such as HP-PA8000 and SPARC-V9 already provide these special instructions for prefetching into the instruction cache. For example, SPARC-V9 provides the pseudo instruction `iprefetch address` to prefetch the instruction code from the given address. Other processors in the near future are likely to follow this lead.

New binaries of the DBMS can be generated in the background by running the profile workload on the sampled input set. The decision to exchange an existing DBMS binary for a new binary can be made based on how different the current query workload is compared to the profile workload. Most processors provide hardware monitors for non-intrusively measuring the I-cache miss rates [htl]. Using these hardware monitors, I-cache miss rates observed during the profile workload execution can be recorded. During the actual workload execution the recorded information can be compared with information from the hardware monitors to determine if the cache performance of the current workload is much worse than the profile workload. If so, the decision can be made to generate a new binary and exchange it for the currently running DBMS.

### 4 Performance Evaluation

In this section we first describe the methodology that we used to evaluate the effectiveness of CGP and then present various experiments that were carried out to evaluate CGP.
4.1 Methodology

To evaluate the effectiveness of CGP we implemented a subset of the relational operators on top of the SHORE storage manager [CDF+94]. SHORE provides storage volumes, files of untyped objects, B+-trees, and R*-trees. SHORE also provides full concurrency control and recovery with two-phase locking and write-ahead logging. We implemented the following relational operators on top of SHORE: select, indexed select, gruce join, nested loops join, indexed nested loop join and hash-based aggregate. Each SQL query was transformed into a query plan using these operators. The relational operators and the underlying storage manager were compiled on an Alpha 21264 processor running OSF Version 4.0F. We used Compaq C++ compiler, version 6.2, with -O5 -ipo -inline speed, optimization flags turned on. This binary with a sample workload, described below, was fed into CGP which inserted the prefetch instructions. We ran the new database binary on top of a simulated processor that implements instruction prefetching. The simulator used for this purpose was the SimpleScalar simulator [BA97], which is a detailed cycle-level processor simulator. The simulator also gives us the ability to analyze the instruction cache behavior and also shows the performance impact of future processor designs on database systems.

The microarchitectural parameters that we used for this performance evaluation are shown in Table 1. This processor can execute up to 4 instructions every cycle similar to a Pentium Xeon processor.

<table>
<thead>
<tr>
<th>Level 1 Instruction cache</th>
<th>16 KB, 2-way associative, 32 byte line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 2 Instruction cache</td>
<td>512 KB, 2-way associative, 32 byte line</td>
</tr>
<tr>
<td>Level 1 cache hit latency</td>
<td>1 cycle</td>
</tr>
<tr>
<td>Level 2 cache hit latency</td>
<td>16 cycles</td>
</tr>
<tr>
<td>Mem access latency</td>
<td>40 cycles</td>
</tr>
</tbody>
</table>

Table 1: Microarchitectural parameters for current processor configuration

To evaluate the performance of CGP we used a database workload that consisted of eight queries
from the Wisconsin benchmark [BDT83], and five queries from the TPC-H [Cou99] benchmark. The Wisconsin benchmark queries that we used are: queries 1 through 7 and query 9. Queries 1 through 7 are 1% and 10% range selection queries (with and without indices) and query 9 is two-way join query. The TPC-H queries that we used are: queries 1, 2, 3, 5 and 6. These queries include complex queries with aggregations and many joins, and also includes a simple nested query (query 2).

We selected queries from two different benchmarks to demonstrate how CGP adapts to changing and mixed workloads. For the profile workload that is provided as input to CGP, we only used three queries from the Wisconsin benchmark. These include query 1 (sequential scan), query 5 (non-clustered index select) and query 9 (two-way join). These queries were chosen as they include the query operations that are frequently used by the of Wisconsin benchmark queries. Note that none of these queries include any aggregations or joins of more than two relations, which is the common in the TPC-H queries that we selected. Along with the profile queries, we also generated a profile workload for the Wisconsin benchmark with just 2200 tuples. In an actual implementation the profile queries can be generated by the database system by looking at its query log and picking queries that include the most frequently used query operations (like hash-join, or indexed-select). The sample dataset for the workload can be generated using known sampling techniques [GM98, BDF+97, LN95]. As the performance results in this section show, CGP is effective even if the actual workload differs significantly from the profile workload. This property of CGP makes is easier for the database system to produce a sample workload.

The results in this section evaluate the effectiveness of CGP for four different workload configurations. These workloads are:

1. *Wisc-prof* has the same queries and data set as the profile workload.
2. *Wisc-large-1* consists of the same three queries used in the profile workload, except that the queries were run on a larger 22,000 tuple Wisconsin database (10,000 tuples in TenK1 and TenK2 relations each, and 1,000 tuples each in the two OneK relations). The total size of the database including the indices is 10MB.

3. *Wisc-large-2* consists of the eight Wisconsin queries running on a 10MB database.

4. *Wisc+tpch* consists of all the eight Wisconsin queries and the five TPC-H queries running concurrently on a total database of size 40MB. In this workload the size of the TPC-H dataset was 30MB (scale factor of 0.01).

The queries in each workload were all executed concurrently, each query running as a separate thread on the database server. We used a small database size (40MB) to allow the SimpleScalar simulation to complete in a reasonable time. Even with this small database the total number of instructions simulated in *Tpch+Wisc* is about 3.7 billion. Increasing the size of the data set only increases the number of instructions executed in each function but does not significantly alter the types and sequence of functions calls that are made. Consequently, CGP is fairly independent of the database size that is used. To verify this claim, we simulated the eight Wisconsin queries on a 100MB data set using CGP and saw similar improvements as in 10MB data set.

### 4.2 Feedback Directed Code Layout with OM

Before presenting the results for CGP, we briefly discuss the feedback directed code layout optimization of OM that reduces I-cache misses by increasing spatial locality. Since CGP also targets I-cache misses, we applied CGP to an OM optimized binary to see how much additional benefit CGP can provide.

The OM [SW92] tool on Alpha processors implements a modified version of the Pettis and Hansen [PH90]
profile directed code layout algorithm for reducing instruction cache misses. OM performs the following two levels of optimizations: In the first level, the basic blocks within a function are rearranged such that conditional branches are most likely not to be taken. This optimization generates extended basic blocks with fewer taken branches. OM uses profile information to determine the likely outcome of the conditional branches. This technique reduces the number of taken branches, which in turn creates extended basic blocks. Consequently, the number of instructions used in each cache line increases, which reduces the I-cache misses. The second level of optimization rearranges functions using a closest-is-best strategy. If one function calls another function frequently, the two functions are allocated close to one another in memory so as to improve the spatial locality. Since OM is a link time optimizer, it has the ability to rearrange functions that are spread across multiple files as well as statically linked library routines.

The profile information needed for OM optimizations was generated by running two workloads, Wisc-prof and Wisc+tpch. Both these workloads were run separately and the profile information of both the runs were merged to generate the required feedback file used by OM. The OM optimizations were applied to an O5 optimized binary. OM's ability to analyze object level code opens up new opportunities for redoing some of the traditional compiler optimizations, such as inter-procedural dead code elimination and loop-invariant code motion, that could not be performed effectively at compile time even with O5 optimizations. Such optimizations reduced the dynamic instruction count of OM code by 12% relative to O5 optimized code.

4.3 Experiment#1 : Effectiveness of OM and CGP

In this section we present the performance improvements that resulted from the reduction in the dynamic instruction count due to OM optimizations. We also present the additional performance improvements resulting from applying CGP to an OM optimized binary.
Figure 4 shows the execution cycles needed for running the four workloads using the O5 optimized binary, O5+OM optimized binary and the binary generated after running through the CGP algorithm. We selected three different values for N, the number of cache lines prefetched at each prefetch insertion point in CGP. These values are: 2, 4 and ALL (prefetch the entire function). The corresponding labels in the graphs are $CGP_2$, $CGP_4$, $CGP_ALL$

The figure shows that on average OM optimizations result in a 48% speedup over O5 optimized code. $CGP_2$ achieves 32% further speedup over OM and $CGP_4$ achieves 42% speedup. The incremental improvement due to prefetching the entire function is not as significant compared to the improvements achieved by prefetching just 2 or 4 lines of the function.

CGP consistently improves the performance of the database system across all the four workloads. These performance improvements are significant not only for the first workload $wisc-prof$, where the workload exactly matches the profile workload, but also for the $tpch+wisc$ workload in which the mix of queries and the data sets differ significantly from the profile workload.

We now explain why CGP improves the performance significantly over OM optimizations. The $closest-is-best$ strategy used by OM for code layout is not very effective for functions that are frequently called from many different places in the code. For instance procedures such as $lock_record()$ can be invoked by several functions in the database system and OM’s $closest-is-best$ strategy places $lock_record()$ close to only a few of its callers by replicating $lock_record()$. Aggressive function replication can cause significant code bloat which can adversely affect the I-cache performance. Hence the best code layout achieved by OM will be sub-optimal. CGP can exploit such sub-optimal choices made by OM and can improve the performance. For instance, CGP can prefetch $lock_record()$ from those functions that invoke $lock_record()$ but were not chosen by OM as candidates for $closest-is-best$ strategy placement.
In Figure 4, the last bar in each workload shows the execution cycles required when there is an infinite sized I-cache, where all I-cache accesses require only one clock cycle. The infinite sized I-cache represents a hypothetical scenario and is used to quantify the theoretical best-case scenario.

![Graph showing execution cycles](image)

**Figure 4**: Execution cycles taken by the original database binary and the binary generated by CGP. The execution time is the number of cycles times the processor clock frequency (approx 500MHz on current processors).

Figure 5 plots the number of I-cache misses in the original database binary and the modified binary. The number of I-cache misses decrease by 40% due to OM optimizations but using CGP further reduces the I-cache misses by 83% resulting in significant performance improvements.

![Graph showing cache misses](image)

**Figure 5**: Cache misses in the original database binary and the binary generated by CGP. Infinite I-cache will suffer zero cache misses and is not shown in this graph.
4.4 Experiment#2: I-cache performance in Future processors

The previous section shows that CGP performs well on configurations that match the processors that exist in typical computer systems today. However, processors in the future will have larger caches and will have wider out-of-order execution cores. The question that immediately arises is: Will future processors with larger I-caches and wider out-of-order execution cores make CGP redundant by eliminating the I-cache access bottleneck? In this section, we try to answer this question.

<table>
<thead>
<tr>
<th></th>
<th>Level 1 Instruction cache</th>
<th>Level 2 Instruction cache</th>
<th>Level 1 cache hit latency</th>
<th>Level 2 cache hit latency</th>
<th>Mem access latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32 KB, 2-way, 32byte line</td>
<td>2 MB, 4-way, 64byte line</td>
<td>1 cycle</td>
<td>40 cycles</td>
<td>100 cycles</td>
</tr>
</tbody>
</table>

Table 2: Microarchitectural parameters for future processor configuration

We simulated a very aggressive out-of-order processor which can execute up to 8 instructions every cycle. Table 2 shows the relevant microarchitectural parameters for this study. Comparing this configuration with the current configuration show in Table 1, the I-cache size is now doubled. This larger cache should reduce the number of I-cache misses. Note, however that in the future configuration the Level 2 cache hit latency and the memory access latency is much greater. This is expected because processors keep getting faster as per Moore’s Law, but the speed of the memory bus that connects the memory to the processor does not increase at the same rate. In the last three years, processors have quadrupled their clock cycles, whereas the bus speed has only doubled. Consequently, even though future processor will have fewer I-cache misses, the penalty for a cache miss will be much higher.

Figure 6 shows the I-cache misses of the original binary and the modified binary running on the future configuration. Comparing Figure 5 and Figure 6 the number of I-cache misses decrease
significantly in the *future* configuration because of the larger I-cache size.

![I-cache misses in future processor configuration.](image)

Figure 6: I-cache misses in future processor configuration.

Figure 7 shows the execution cycles required to complete the four workloads in the *future* configuration. First we will compare the execution cycles of the infinite I-cache configuration in Figures 4 and 7. Comparing these figures, one can observe that the execution cycles of the infinite I-cache configuration decreases in the *future* configuration. The total number of processor cycles is reduced because the wider issue width of the processor allows the processor to execute more instructions concurrently. However, without the infinite I-cache, surprisingly, the execution cycles of all the workloads increases in the *future* configuration! Even though there are fewer I-cache misses in future processors (compare Figures 5 and 6), the penalty of each cache miss is significantly higher (compare Tables 1 and 2). The higher cost of I-cache misses overshadows the reduction in the number of I-cache misses, which causes the execution cycles to increase in future configuration. Consequently, techniques like CGP that reduce the I-cache misses will be even more critical to the performance of database systems in future processors.

4.5 **Experiment#3 : Impact of the hardware support for CGP**

In this experiment, we explore the possibility of using hardware techniques to improve the performance of CGP even further. With the technological trend towards increasing the number of
transistors on a chip, processor designers are looking for better ways of using the “real estate” on the processor. General-purpose processors (Pentium, PowerPC etc.,) are typically designed with workloads in mind that are expected to be market dominant. Data intensive workloads, which significantly include database workloads, are increasingly dominant today, rather than the more compute intensive applications that influenced the design of general purpose microprocessors in the recent past. This shift that has occurred, and been increasingly recognized by the microprocessor industry over the past decade, gives the database research community great leverage to dictate what the next generation general purpose computers must provide in order to satisfy this community. This sections looks at one potential use for this transistor real-estate that can help CGP to improve the performance of a database system beyond the improvements presented in the previous sections.

The CGP algorithm, described Section 3.1, uses profile workload execution to label the edges of the call graph to determine which functions to prefetch. If the actual workload has a different call graph profile then CGP may not be effective. In this section, we present some preliminary results that show how special hardware support can be used to dynamically select the functions that are prefetched during query execution.
The sequence of function calls invoked by a function $F$ are stored in a special hardware table called 
*Call Graph History Table (CGHT)*. When function $F$ is invoked the corresponding entry in CGHT is accessed to determine the function call sequence invoked by $F$ during its last execution. Then, the hardware dynamically Prefetches functions according to the last calling sequence. This simple scheme predicts that the control flow of the current invocation of function $F$ will be the same as the last invocation of this function. Figure 8 shows a simplified hardware structure for CGHT. In this figure, each row contains a sequence of function addresses that were invoked when a function was last executed.

<table>
<thead>
<tr>
<th>Func_name</th>
<th>Sequence of functions last called</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Create_rec</td>
<td>Find_page_in_Buffer Update_page</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 8: Call Graph History Table Structure

Figure 9 compares the execution cycles for the database workloads running on the current configuration with and without additional hardware support for CGP. The additional hardware support to dynamically alter the functions Prefetched will increase the performance by 10%; 10% is usually considered a significant performance improvement by the microprocessor designers to consider using the additional transistors to achieve this improvement. These are only preliminary results shown to demonstrate the importance of utilizing the additional transistors to improve the performance of the database systems.
5 Conclusions and Future work

This paper proposes and evaluates a technique called Call Graph Prefetching (CGP) that increases the performance of database systems by improving their I-cache utilization. With data sets that are mostly main memory resident, CGP can improve the performance of a database system by 42%. The strength of CGP is that it does not require recoding the database system or require access to the actual source code for the database system. To use CGP, the database system must provide a simple workload, which is then used by CGP to analyze the behavior of the database system’s call graph. Based on this analysis, CGP inserts prefetch instructions in the database binary to reduce the I-cache misses. Since database systems are built using layered software architecture techniques, many functions in the database code are called repeatedly during query evaluation. These functions that are called frequently tend to be fairly independent of the actual database workload. This implies that the database system has great flexibility in choosing the sample workload that it provides to CGP.

This paper also shows that CGP will be even more effective in future processors that have larger I-caches but suffer larger cache miss penalties. We also present preliminary results that show how
special hardware can be used to allow CGP to dynamically adapt to changing database workloads.

CGP does not require this specialized hardware but improves the performance of the database system even further if it has the specialized hardware.

Microprocessor designers today are paying closer attention to data intensive applications in helping them choose the features that they put into the next generation of general-purpose commodity microprocessors. We believe that the database community could play a crucial role in influencing the features that can be put in the future microprocessors. The Call Graph History Table is one example that can be used with CGP to improve the performance of database systems. Conversely, as shown in this and other recent papers, as database systems become increasingly compute bound, paying closer attention to utilizing the hardware effectively can pay rich dividends.

References


