Call Graph Prefetching for Database Applications

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 continuing technological trend of ever cheaper and larger memory, most data sets in database servers will soon be able to reside 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. Previous work has shown that database applications have large instruction and data footprints and hence do not use processor caches effectively. In this paper, we propose Call Graph Prefetching (CGP), a hardware technique that analyzes the call graph of a database system and prefetches instructions from the function that is deemed likely to be called next. CGP capitalizes on the highly predictable function call sequences that are typical of database systems. We evaluate the performance of CGP on sets of Wisconsin and TPC-H queries, as well as on CPU-2000 benchmarks. For most CPU-2000 applications the number of I-cache misses were very few even without any prefetching, obviating the need for CGP. Our database experiments show that CGP can improve the performance of a database system by 30% over a baseline system that uses the OM tool to layout the code so as to improve I-cache performance. CGP also achieved 7% higher performance than *next-N-line* prefetching on database applications.

1   Introduction

The increasing need to store and query large volumes of data have made database management systems (DBMSs) one of the most prominent applications on today's computer systems. DBMS performance in the past was bottlenecked by disk access latency which is six orders of magnitude slower than processor cycle times. But with the trend toward denser and cheaper memory, database servers in the near future will have large main-memory configurations, and many working sets will be resident in main memory [BBC+98]. Moreover techniques such as concurrent query execution, where a query that is waiting for a disk access is switched with another query that is ready for execution, can successfully mask the disk access latencies. Several commercial database systems already implement concurrent query execution along with asynchronous I/O to reduce the I/O bottleneck. Once the disk access latency is tolerated, or disk accesses are sufficiently

infrequent, the performance bottleneck shifts from I/O response time to the memory access time.

There is a growing gap between processor and memory speeds which can be reduced by the effective use of multi-level caches. But recent studies have shown that current database systems with their large code and data footprints suffer significantly from poor cache performance [ADHW99, BRK98, LBE+98, NBC+94, SKN94]. Thus the key challenge in improving the performance of memory bound database systems is to utilize caches effectively and reduce cache miss stalls.

In this paper, we propose *Call Graph Prefetching (CGP)*, a hardware instruction prefetching technique that analyzes the call graph of an application and prefetches instructions to reduce the instruction cache misses. Although CGP is a generic instruction prefetching scheme, it is particularly effective for DBMS because of the layered software design approach used by these systems. CGP uses a Call Graph History Cache (CGHC) to dynamically store sequences of functions invoked during program execution, and uses the stored history when choosing which functions to prefetch. We evaluate the effectiveness of CGP using a subset of CPU-2000 benchmarks and a database workload that consists of a subset of the Wisconsin queries [BDT83] and TPC-H [Cou99] queries.

Our performance evaluations show that most CPU-2000 benchmarks do not need any prefetching because of the very few I-cache misses suffered by these benchmarks. On the other hand, the database workloads do suffer a significant number of I-cache misses, and CGP improves their performance by 30% over a baseline system that has been tuned up by using the OM tool to layout the code for improved I-cache performance. CGP issues more useful prefetches than *next-N-line (NL)* prefetching, while keeping the prefetch traffic similar to NL. On the DBMS workloads CGP reduces the cache misses by 10% and improves the performance by 7% relative to NL.

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 program execution by stalling the issuing of instructions in the processor pipeline until the

cache miss is serviced. Nevertheless 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 discusses the architectural modifications needed for implementation. Section 4 describes the simulation environment and performance analysis tools that we used to assess the effectiveness of CGP. We also evaluate the performance of CGP when applied to CPU2000 benchmarks and show that although CGP can be applied to other application domains it is most effective for database systems. The results of the assessment are presented in Section 5, and we conclude in Section 6.

2 Related Work

Researchers have proposed several techniques to improve the I/O bottleneck of database systems. Nyberg et al. [NBC+94] suggested that if data intensive applications use software assisted disk striping, the performance bottleneck shifts from I/O response time to the memory access time. Boncz et al. [BRK98] 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 that improve the locality of the data accesses for *join* and *aggregation* algorithms. These techniques reduce the data cache misses, and are orthogonal to the goal of CGP which tries to reduce I-cache misses. CGP may be implemented on top of these cache-conscious algorithms.

It is only recently that researchers have examined the performance impact of architectural features on DBMSs [ADHW99, LBE+98, TLPZT97, EJK+96, RBH+95, CB94, MDB94]. Their results show that database applications have large instruction and data footprints and exhibit more unpredictable branch behavior than benchmarks that are commonly used 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 about 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). Surprisingly, their results showed that L1 data cache misses that hit in L2 were not a significant bottleneck, but L2 data cache misses reduced the performance by 20%.

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 contiguously the sequence of basic blocks that lie on the most commonly occurring control flow path. Romer *et al.* implemented the Pettis and Hansen code layout algorithm using the Etch [RVL$^+$97] tool and showed performance improvements for Win32 binaries. In this paper we used OM [SW92] which implements a modified Pettis and Hansen algorithm to do feedback-directed code layout. This algorithm is discussed further in Section 5.1. Our results show that OM improves the performance of a highly optimized binary (C++ -O5 optimization level) by 11%. Using CGP in addition to OM further improves the performance by 30%, for a combined 45% performance improvement. CGP alone, without OM, achieved a 40% performance improvement.

Next-N-line prefetching (NL) [Smi78] is another prefetching technique that is often used. In this technique when a line is being fetched by the CPU, the next N sequential lines are prefetched, unless they are already in cache. This scheme works in programs that execute long sequences of straight line code. We used NL prefetching for prefetching code within a function and used CGP for prefetching across function calls. We show that CGP takes good advantage of the nextline prefetching scheme and also outperforms a pure nextline scheme by 7%.

Researchers have proposed several techniques for non-sequential instruction prefetching [SH92, CLM97, LM98, RCA99]. Of these, the work that is closest to the research presented in this paper is that of Luk and Mowry [LM98]. They proposed cooperative prefetching where the compiler inserts prefetch instructions

to prefetch branch targets. Their approach, however, requires ISA extensions to add four new prefetch instructions: two to prefetch the targets of branches, one for indirect jumps and one for function returns. They use next-N-line prefetching for sequential accesses. Special hardware filters are used to reduce the prefetch traffic. Since they targeted their scheme to perform prefetching for a wide range of applications such as the SPEC95 benchmarks, SUIF compiler passes, and Postgres, they needed special ISA extensions as well as additional hardware support for prefetch filtering. By contrast, CGP is a simple hardware scheme that discovers and exploits predictable call behavior as found, for example, in database applications due to their layered software design. CGP uses NL prefetching to prefetch within a function boundary and can benefit from using the OM tool at link time to make NL more effective by reducing the number of taken branches which increases the sequentiality of the code. Hence using NL with OM can effectively prefetch the instructions within a function boundary and thereby will reduce the need for branch target prefetching within a function boundary. CGP is thus freed to focus on prefetching for function calls. Since CGP is implemented in hardware it permits running legacy code without modification or recompilation, which is particularly attractive for large software systems such as databases.

3   Call Graph Prefetching (CGP)

DBMSs are commonly built using a layered software architecture where each layer provides a set of well-defined entry points to the layers above it. Figure 1 shows the layers in a typical database system with the storage manager being the bottom-most layer. The storage manager provides basic file storage mechanisms (such as tables and indices), concurrency control and transaction management facilities. Relational operators that implement algorithms for join, aggregation etc., are typically built on top of the storage manager. The query scheduler, the query optimizer and the query parser 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 its internal implementation details so as to improve the portability and maintainability of the software. The sequence
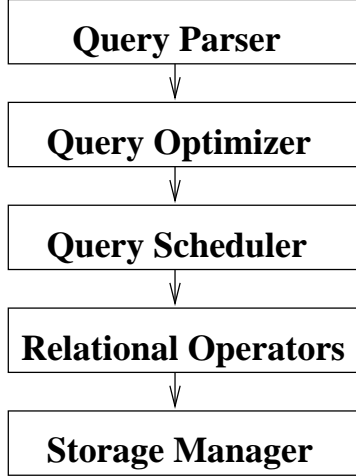
```
┌─────────────────────────┐
│      Query Parser       │
└─────────────────────────┘
             │
             ∨
┌─────────────────────────┐
│     Query Optimizer     │
└─────────────────────────┘
             │
             ∨
┌─────────────────────────┐
│     Query Scheduler     │
└─────────────────────────┘
             │
             ∨
┌─────────────────────────┐
│   Relational Operators  │
└─────────────────────────┘
             │
             ∨
┌─────────────────────────┐
│     Storage Manager     │
└─────────────────────────┘
```

Figure 1: Software layers in a typical database system

of function calls within each of these entry points is transparent to the layers above. Although such layered

code typically exhibits poor spatial and temporal locality, the function call sequences can often be predicted

with great accuracy. CGP exploits this predictability to prefetch instructions from the procedure that is

deemed most likely to be executed next.

3.1   A Simple Call Graph Example

We introduce CGP with the following pedagogical example. Figure 2 shows a segment of a call graph for

adding a record to a file in SHORE [CDF+94]. 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_pool to check if the relation into which the record is being added is already 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,

subsequently updated using *Update_page*, and finally unlocked using *Unlock_page*.

The *Create_rec* function is the entry point provided by the storage manager to create a record, and is

routinely invoked by a number of relational operators, including insert, bulk load, join (to create temporary

partitions or sorted runs), and aggregate. Although it is difficult to predict calls to *Create_rec*, once it is

invoked *Find_page_in_buffer_pool* is always the next function to be called. When a page is brought into the

memory buffer pool from the disk, DBMSs typically "pin" the page in the buffer pool to prevent the possibility

of its being replaced before it is used. Given a large buffer pool size and repeated calls to *Create_rec*, the page

that is being updated will usually be found pinned in the buffer. Hence *Getpage_from_disk* will usually not

be called and *Lock_page*, *Update_page* and *Unlock_page* will be the sequence of functions next invoked. CGP

capitalizes on this predictability by prefetching instructions needed for executing *Find_page_in_buffer_pool*

upon entering *Create_rec*, then prefetching instructions for *Lock_page* once *Find_page_in_buffer_pool* is entered,

and finally prefetching instructions from *Update_page* after returning from *Find_page_in_buffer_pool*, and for

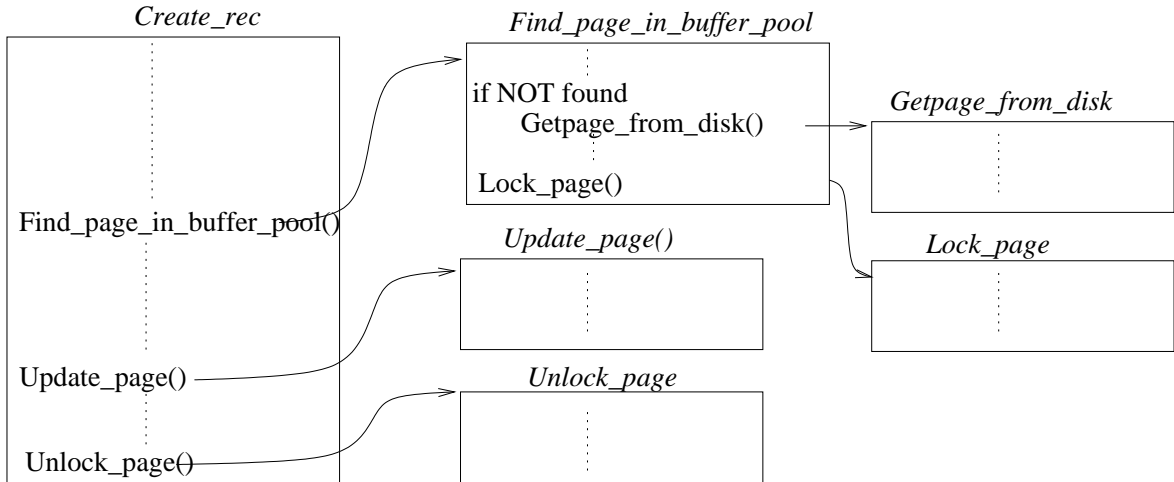*Unlock_page* upon returning from *Update_page*.

Figure 2: Call Graph for the *Create_rec* function

3.2   Exploiting Call Graph Information

The main hardware component of the CGP prefetcher is the Call Graph History Cache (CGHC) which

comprises a tag array and a data array as shown in Figure 3. Each entry in the tag array stores the starting

address of a function (F) and an index (I). The corresponding entry in the data array stores a sequence

of starting addresses corresponding to the sequence of functions that were called by F the last time that

function F was called. If F has not yet returned from its most recent call, this sequence may be partially

updated. For ease of explanation here and in Figure 3 we use the function name to represent the starting address of the function. Each entry in the data array, as implemented in our evaluations, can store up to 8 function addresses. If a function in the tag entry invokes more than 8 functions, only the first 8 functions invoked will be stored. As shown later in Section 5.3, a small direct mapped CGHC achieves nearly the same performance as an infinite sized CGHC and hence we chose to use a direct mapped CGHC instead of a set-associative CGHC.

Each *call* and each *return* instruction that is executed makes two accesses to CGHC. In both cases the first access to CGHC is made using target address of the *call* (or the *return*) to determine which function to prefetch next; the second access to CGHC is made using the starting address of the currently executing function to update the current function's index and calling sequence that is stored in CGHC. To quickly generate the target address of the *call* or the *return* instruction, the processor's branch predictor is used, instead of waiting for the target address computation which may take several cycles in the out-of-order processor pipeline. On a CGHC access, if there is no hit in the tag array, no prefetches are issued and a new tag array entry is created with the desired tag and an index value of 1. The corresponding data array entry is marked invalid, unless the CGHC miss occurs on the second (update) access for a *call* (say P calls F), in which case the first slot of the data array entry for P is set to F.

In general, the index value in the tag array entry for a function F, points to one of the functions in the data array entry for F. An index value of 1 selects the first function in the data array entry. Note that the index value is initialized to 1 whenever a new entry is created for F, and the index value is reset to 1 whenever F returns.

When the branch predictor predicts that P is calling F, the first (prefetch) access to the direct mapped CGHC tag array is made by using the lower order bits of the predicted target address, F, of the function call. If the address stored in the tag entry matches F, as the index value of a function being called should be 1, a prefetch is issued to the first function address that is stored in the corresponding data array

8

entry. The second function will be prefetched when the first function returns, the third when the second returns etc. The prefetcher thus predicts that the sequence of calls to be invoked by F will be the same as the last time F was executed. We chose to implement this prediction scheme because of the simplicity of its prefetch logic and the accuracy of this predictor for stable call sequences.
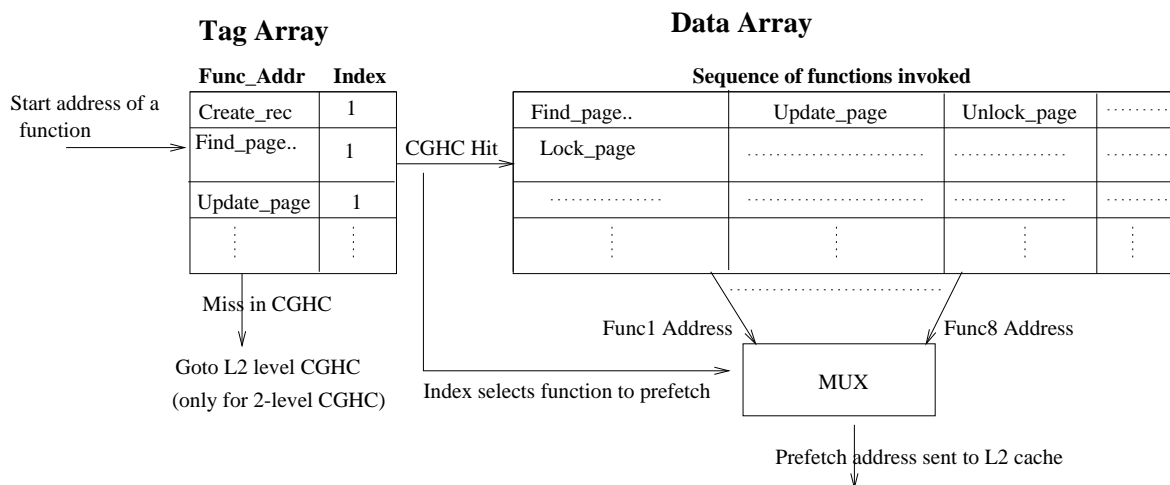
For the same *call* instruction (P calls F), the second (update) access to the CGHC tag array is made using the lower order bits of the starting address of the current function P. If the address stored in the tag entry matches P, then the index of that entry is used to select one of 8 slots of the corresponding data array, and F is stored in that slot. The index is incremented by 1 on each *call* update, up to a maximum value of 8.

On a *return* instruction, when the function F returns to function P, the lower order bits of the starting address of P are used for the first (prefetch) access the CGHC. On a tag hit, the corresponding data array entry is accessed to prefetch the function that is in the slot pointed to by the index value in the tag array entry.

A modified branch predictor is used to provide the starting address of P. On a *return* instruction, a conventional branch predictor only predicts the return address in P to which F returns, in particular it does not provide the starting address of P. Since the entries in the tag array store only starting addresses of functions, the target address of a *return* instruction cannot be directly used for a tag match in CGHC. To overcome this problem the processor always keeps track of the starting address of the function currently being executed. When a *call* instruction is encountered, the starting address of the caller function is pushed onto the branch predictor's return address stack structure along with the return address. On a *return* instruction, the modified branch predictor retrieves the return address as usual, and also gets the caller function starting address which is used to access the CGHC tag array.

On the same *return* instruction, the second (update) access to CGHC is made with the lower order bits of the starting address of the current returning function F. On a tag hit, the index value in the tag

9

array entry is reset to one.

Since CGP predicts that the sequence of function calls made by a caller will be the same as the last time that caller was executed, prefetching an entire function based on this prediction may waste processor resources if the prefetched function is not invoked during the actual execution. Moreover prefetching a large function into the instruction cache can pollute the cache by replacing existing cache lines that may be needed sooner than the prefetched lines. Hence the prefetch algorithm only prefetches N cache lines, where N is a parameter that can be based on the cache size, line size and the I-cache miss latency. Since only the first N cache lines of a callee function are prefetched from within the caller function, the rest of the callee function is prefetched after entering the callee function by using a simple NL prefetching scheme. We use the notation CGP_N to represent a CGP scheme that prefetches only N cache lines rather than an entire function on each prefetch request.

Figure 3: Call Graph History Cache. (state shown in CGHC occurs as *Lock_page* is being prefetched from *Find_page_in_buffer_pool*)

3.3   Design considerations

Operations that access and update the CGHC are not on the critical path of the processor pipeline and can be done in the background. In our implementation the CGHC is accessed and updated in different cycles to eliminate the need for having a dual-ported CGHC that can be accessed and updated in the same cycle.

The CGHC is accessed one cycle after the branch predictor predicts the target of a *call* or *return* instruction. Since the CGHC is a small direct mapped cache, the tag match of the target address is done in 1 cycle. On a hit in CGHC, a prefetch is issued in the next cycle. After the tag match and the decision whether to issue a prefetch, the CGHC is updated in the next clock cycle to reflect the call sequence history.

Our current CGP implementation prefetches instructions directly into the L1 I-cache. The traffic generated by the prefetches and the L1 cache misses are serviced by L2 in a FIFO order without giving any priority to the demand miss traffic. Although the lack of priority may increase the latency of the demand miss traffic, it simplifies the L2 accessing interface within the L1 cache.

4 Simulation Environment and Benchmarks

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 is a fully functional storage manager which has been used extensively in the database research community and is also used in some commercial database systems. SHORE provides storage volumes, files of untyped objects, B+ trees, and R* trees, 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, grace 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 compiled SHORE using the Compaq C++ compiler, version 6.2, with the -O5 -ifo -inline speed flags turned on.

We used the SimpleScalar simulator [BA97], for detailed cycle-level processor simulation. The microarchitectural parameters that we used for this performance evaluation are shown in Table 1.

Fetch, Decode & Issue Width	4
Inst Fetch & L/S Queue Size	16
Reservation stations	64
Functional Units	4add/2mult
Memory system ports to CPU	4
L1 I and D cache each	32KB,2-way,32byte
Unified L2 cache	1MB,4-way,32byte
L1 hit latency(cycles)	1
L2 hit latency(cycles)	16
Mem latency (cycles)	80
Branch Predictor	2-lev,2K-entry

Table 1: Microarchitectural Parameter Values

To evaluate the performance of CGP we used a database workload that consists of eight queries (1 through 7 and 9) from the Wisconsin benchmark [BDT83], and five queries (1,2,3,5, and 6) from the TPC-H benchmark [Cou99]. Wisconsin queries 1 through 7 are 1% and 10% range selection queries (with and without indices) and query 9 is two-way join query. The selected TPC-H queries include queries with aggregations and many joins, and also includes a simple nested query (query 2). The remaining TPC-H queries need more relational operators than what we have currently implemented and hence are not evaluated in this paper.

We selected queries from two different benchmarks, Wisconsin and TPC-H, to demonstrate how CGP performs with mixed workloads. The results in this section evaluate the effectiveness of CGP for four different database workloads. These workloads are:

1. *Wisc-prof*, a set of three queries from the Wisconsin benchmark: query 1 (sequential scan), query 5 (non-clustered index select) and query 9 (two-way join). These queries were chosen since they include query operations that are frequently used by the other Wisconsin benchmark queries. These selected queries were run on a database of 2100 tuples.

2. *Wisc-large-1* consists of the same three queries used in the *Wisc-prof* workload, except that the queries were run on a full 21,000 tuple Wisconsin database (10,000 tuples in each of the first two relations, and 1,000 tuples in the third relation). The total size of the database including the indices is 10MB. This workload was selected to see how CGP performance differs when running the same queries on a

differing database size.

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

4. *Wisc+tpch* consists of all 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.

The queries in each workload were executed concurrently, each query running as a separate thread in 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* was about 3 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 sequences of functions calls that are made. Consequently, CGP performance is fairly independent of the database size that is used. To verify this claim, we simulated CGP on the *Wisc-large-2* queries with a 100MB data set and saw improvements quite similar to those for the 10MB data set.

5   Results

5.1   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 profile-directed code layout algorithm for reducing instruction cache misses [PH90]. OM performs two levels of optimizations. In the first level, OM uses profile information to determine the likely outcome of the conditional branches and rearranges the basic blocks within a function such that conditional branches are most likely not taken. This optimization increases the average number of instructions executed between

two taken branches. Consequently, the number of instructions used in each cache line increases, which in turn reduces 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* to provide better feedback information than that provided by running just one workload. Each of these workloads was run separately and the profile information of both 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 at link time opens up new opportunities for redoing some 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 the OM code by 12%, relative to O5 optimized code.

5.2 Performance Comparison

In this section we present the performance improvements due to OM optimizations. We also present the performance improvements due to CGP without OM optimizations and the 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, the O5+OM optimized binary and the binary generated by running the CGP algorithm on O5 binary and O5+OM binary. We selected two different values for N, the number of cache lines prefetched each time, namely 2 and 4 (corresponding, respectively, to bars labeled *O5+CGP_2/O5+OM+CGP_2*, *O5+CGP_4/O5+OM+CGP_4* in the graphs). For these experiments we used a two level CGHC with 2KB in the first level and 32KB in the second level. Figure 4 shows that on average OM optimizations result

in an 11% speedup over O5 optimized code. CGP_4 alone, without OM, achieves 40% speedup over O5. CGP_4 with OM achieves 45% speedup over O5. This shows that CGP alone can significantly improve the performance, and using CGP with OM gives additional benefits. But in both cases, CGP outperforms OM optimizations.

One observation might best explain why CGP improves the performance significantly over OM optimizations. Namely, 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 I-cache performance. On the other hand, CGP can prefetch *lock_record()* from those functions that invoke *lock_record()*, without having to replicate the function.
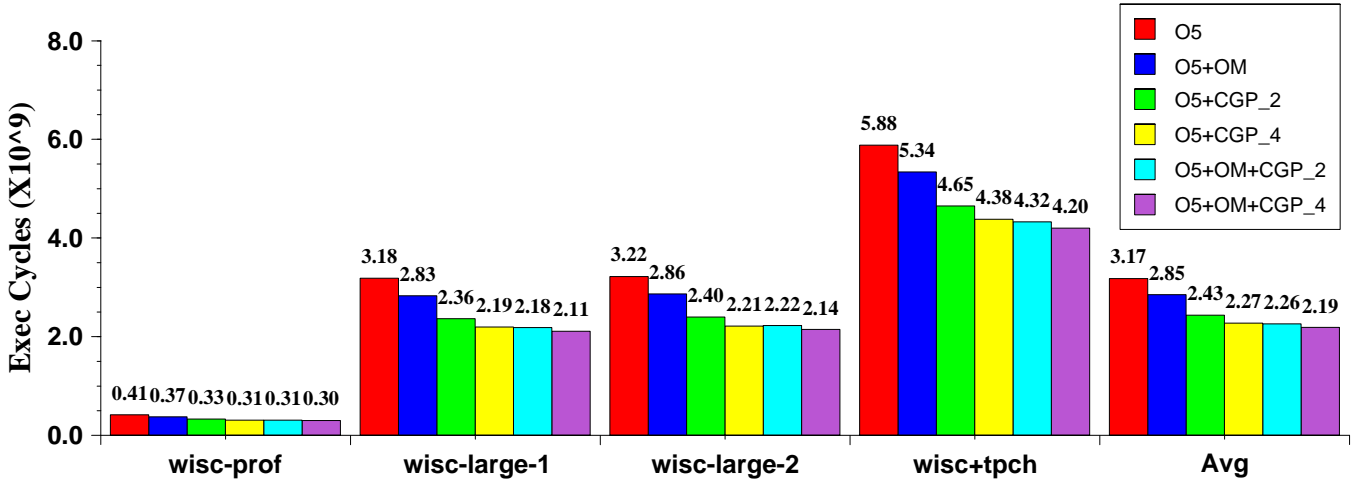

Figure 4: Execution cycle comparison of O5, OM and CGP

5.3 Exploring the design space of CGHC

The performance of CGP depends on the ability of the hardware to store enough call graph history so as to effectively issue prefetches for repeated call sequences. Since CGHC stores this history information,

we explored the effect of varying the size of CGHC on the overall performance of CGP. Figure 5 shows the performance of CGP_4 for five different CGHC configurations, namely 1KB CGHC (*CGHC-1K*), 32KB CGHC (*CGHC-32K*), 1KB+16KB two level CGHC (*CGHC-1K+16K*), 2KB+32KB two level CGHC (*CGHC-2K+32K*), and infinite CGHC (*CGHC-Inf*) where every function in the program has an entry in the CGHC that stores the entire function call sequence of its most recent invocation.

As seen from the figure, a 1KB CGHC is about 12% slower than an infinite CGHC. But the performance gap between the other three finite CGHC configurations and the infinite CGHC is very small. Surprisingly for the *wisc+tpch* benchmark the performance of the infinite CGHC is slightly worse than all configurations except *CGHC-1K*. Since the infinite CGHC caches all the history information, some of the prefetches issued on a function call were useless prefetches that resulted in increased bus traffic. These prefetches may not be issued in other configurations because the finite call history eliminates less recent function call sequences from the CGHC (via LRU replacement) thereby reducing the number of prefetches issued (and hence useless and polluting prefetches as well), and correspondingly reducing the bus contention. In *wisc+tpch* the gains of an infinite CGHC are outweighed by the losses.

Among the four finite CGHC configurations that were simulated, the performances of *CGHC-2K+32K* and *CGHC-32K* are better than the remaining configurations. But instead of using a 32KB one level CGHC with a one cycle access time, we chose to use a two level CGHC with 2KB in the first level CGHC and 32KB in the second. The access times to the two level CGHC are same as the access times of the two level cache hierarchy. On a miss in the first level CGHC, the second level CGHC is accessed. On a hit in the second level CGHC an entry from the first level CGHC is written back to the second level and the hit entry in the second level CGHC is moved to the first level. On a miss in the second level CGHC a new entry is allocated in the first level CGHC and the replaced entry from the first level is written back to the second level. All our further evaluations are presented using a *CGHC-2K+32K* configuration.
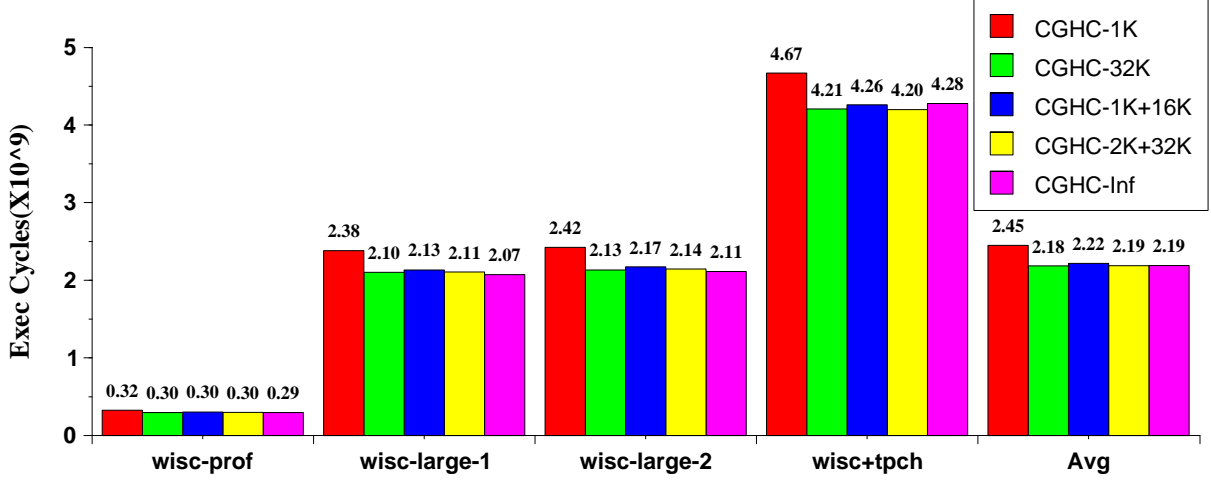
Figure 5: Performance of five different CGHC configurations

5.4 Comparison with Next-N-Line Prefetching

Since OM optimizations increase the sequential accesses to a code segment, a simple prefetching scheme such as NL, where the next N lines from the currently accessed cache line are prefetched, might be successful in bringing in more required cache lines and eliminating more cache misses when applied after OM. Figure 6 compares the performance of NL prefetching with CGP, where each is applied to the OM optimized binary. NL_2 and NL_4 are NL schemes that prefetch the next 2 and next 4 cache lines, respectively, from the currently accessed cache line.

The results show that the NL scheme is indeed effective in improving the performance of the OM optimized binary, but CGP still outperforms NL alone by about 7% and is within 19% of the perfect I-cache performance (labeled as *perf-Icache* in the graph), where all accesses to the I-cache are completed in 1 cycle. The NL scheme is effective for prefetching long straight line sequences of code within a function. In our workloads on average only 43 instructions were executed between two successive function calls. This frequent change in the control flow reduces the effectiveness of the NL scheme even with OM optimizations.
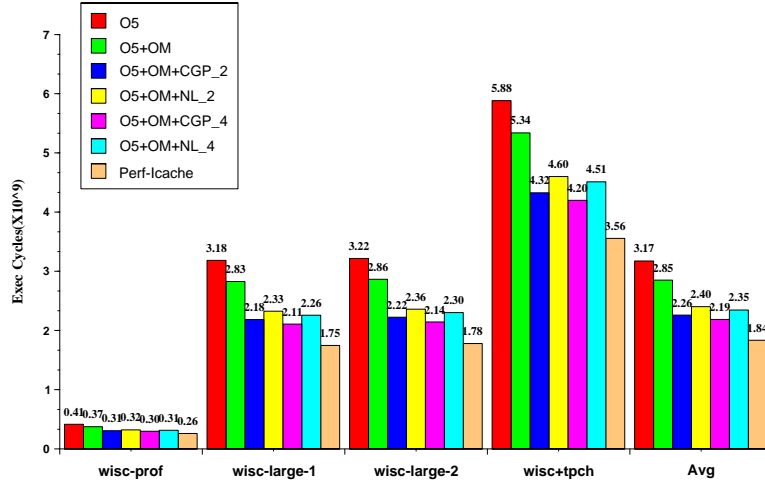
Figure 6: Execution cycle comparison of O5, OM, NL and CGP

5.5 I-cache Performance

Further analysis shows the reasons for the improved performance of CGP over NL. Figure 7 shows the number of I-cache misses. The OM reorganization reduces the number of cache misses by 21% relative to the O5 optimized binary, but OM+NL reduces cache misses by 77% and OM+CGP by 87%.
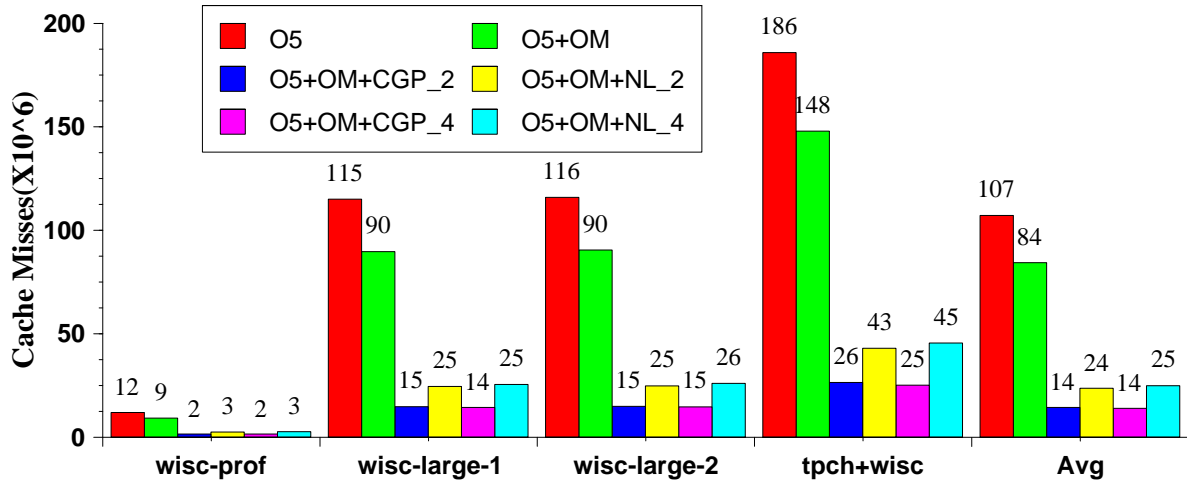

Figure 7: I-Cache miss comparison of O5, OM, NL and CGP

5.6   Prefetch Effectiveness and Bus Traffic overhead

Figure 8 shows the prefetch effectiveness of CGP and NL by categorizing all the issued prefetches into three categories. The bottom component shows *Pref Hits*, the number of times the first reference to a L1 cache line that was prefetched found the referenced instruction already in the L1 cache. The center component shows *Delayed Hits*, the number of times the first reference to a prefetched cache line finds the reference instruction not present in the L1 cache although it is already being fetched from the lower levels of memory due to the previous prefetch request. Finally the upper component shows *Useless Prefetches*, the number of times that cache lines were prefetched into the L1 cache and replaced before being referenced. Since CGP uses CGHC only to prefetch across function boundaries, and uses NL to prefetch within a function, the total number of prefetches issued by CGP_4 are only 8% more than those issued by the NL_4 scheme alone. But this ability to prefetch across function boundaries also allows CGP to do more effective prefetching than NL alone, particularly in the presence of frequent function calls. On average CGP_4 generates 22% more hits to prefetched cache lines than NL_4. Of the prefetches issued by CGP_4, 54% were useful prefetches, and 51% were useful for NL.

Since CGP uses NL to prefetch within a function, we expected that CGP would incur at least as many delayed hits as the NL scheme. But surprisingly there are fewer delayed hits with CGP than with NL alone. Such a reduction could, for example, occur if a callee function is laid out close to its caller and NL running without CGP prefetches the beginning lines of the callee function due to the sequentiality of the code layout, but they do not arrive before they are referenced. However, when CGP runs it issues prefetches to this callee function at the beginning of the caller function resulting in improved timeliness, and thereby converts such delayed hits to hits.

To improve the timeliness of NL, we implemented *run-ahead NL* prefetching, which is a modified NL prefetching scheme (results not shown here). This prefetching scheme, instead of prefetching the next N sequential lines from the currently accessed cache line, prefetches N lines that begin M cache lines after

the currently accessed cache line. Although this scheme did improve the timeliness of some delayed hits, the overall performance of this modified NL scheme is much worse than NL. With frequent control flow changes, and with an average of only 43 instructions between two consecutive functions calls, the *run-ahead NL* scheme prefetches too many useless instructions from too far ahead in the instruction stream, and fails to prefetch some closer lines that are needed, thereby significantly decreasing the number of useful prefetches.
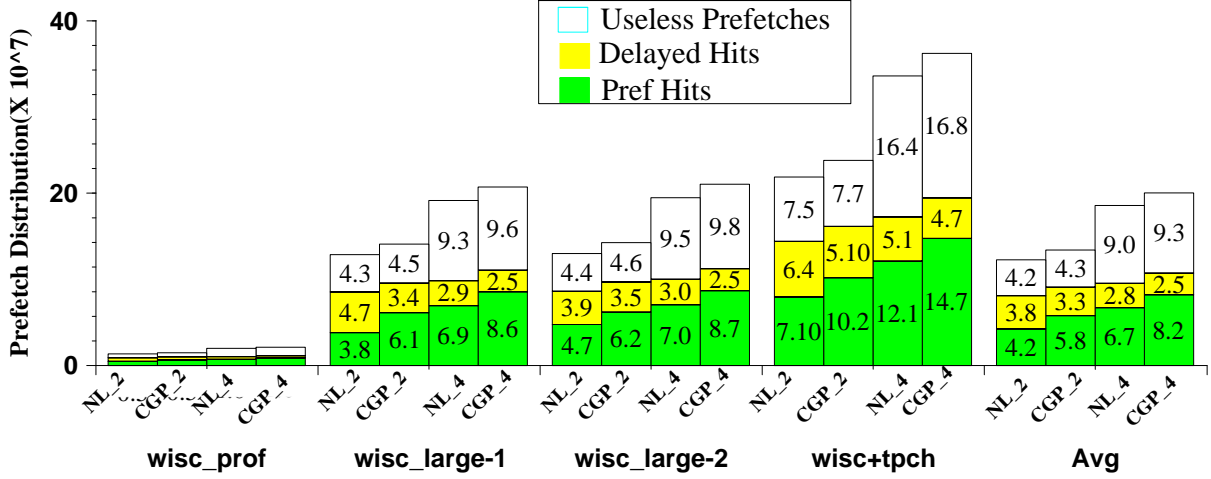

Figure 8: Prefetch Timeliness

5.7 Applying CGP to CPU2000 benchmarks

In this section we show that although CGP is a general technique which can be applied to applications in other domains, the layered software architecture of database applications make CGP particularly attractive for database applications. To quantify the impact of CGP when applied to other application domains, we used CGP on the CPU-intensive SPEC-CPU2000 benchmarks. We selected seven benchmarks from the CPU2000 integer benchmark suite, namely *gzip, gcc, crafty, parser, gap, bzip2* and *twolf*. These benchmarks were compiled, as above, with the Compaq C++ compiler with O5 and then OM. The test input set, provided by SPEC, was used to generate the required profile information for OM. The train input set was then run for two billion instructions to generate the results presented in this section.

In Figure 9, the last bar in each benchmark shows the execution cycles required with a perfect

20

I-cache, where all accesses to the I-cache are completed in 1 cycle. The performance gap due to using a 32

KB I-cache and the perfect I-cache is 17% in *gcc*, 9% in *crafty*, 2% in *gap*, and less than 1% for each of the

other benchmark. CPU2000 applications suffer very few I-cache misses and hence a 32KB I-cache and a

perfect I-cache achieve nearly the same performance. In fact the I-cache miss ratios are nearly 0%, except

for *gcc* and *crafty* which have 0.5% and 0.3% I-cache miss ratios, respectively. The I-cache is thus not a

performance bottleneck in most CPU2000 applications, in which case it is unnecessary to use prefetching

techniques such as CGP and NL. For those applications that do suffer from I-cache misses, namely *gcc* and

crafty, NL prefetching alone achieves performance gains similar to that of CGP. NL_4 and CGP_4 each speed

up the execution of *gcc* by 7 to 8% and *crafty* by 4% relative to O5+OM alone. This shows that CGP is not

especially attractive for workloads with small I-cache footprints and/or infrequent function calls.
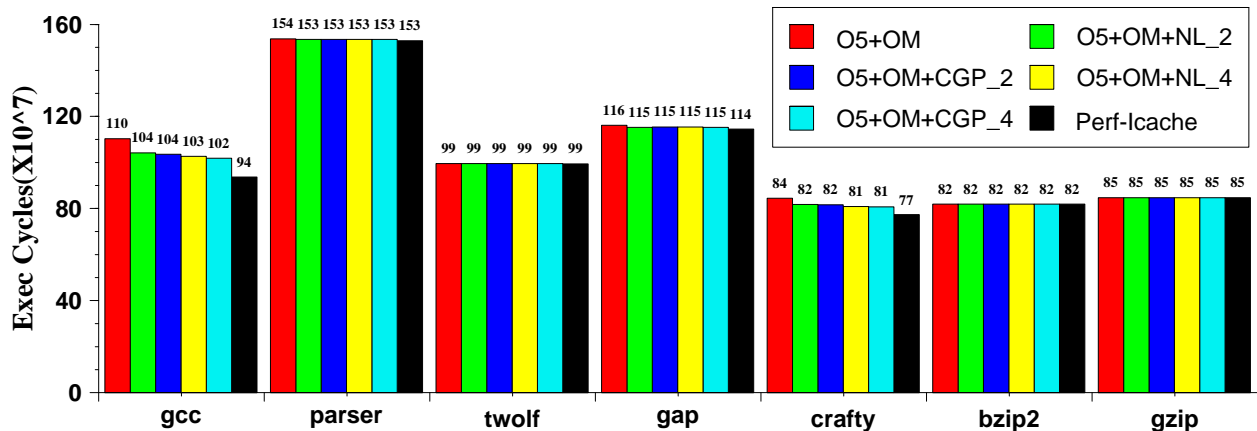

Figure 9: *Effectiveness of CGP on CPU-2000 applications*

6   Conclusions and Future work

This paper proposes *Call Graph Prefetching (CGP)* to increase the performance of database systems by

improving their I-cache utilization. With data sets that are mostly main memory resident, CGP can outper-

form the best existing feedback directed compiler optimizations by 30% and provides an additional speedup

of 7% over NL prefetching. The hardware requirements of CGP are quite modest. By adding a 2KB first

level CGHC with a 32KB second level CGHC, CGP achieves significant performance benefits.

In this paper CGP is implemented in hardware, which permits running legacy code without modification or recompilation. However, it is possible for the compiler to insert prefetch instructions into the code based on call graph information generated from profile executions.

We have demonstrated the effectiveness of CGP for database applications. We expect CGP to be useful as well for other benchmarks where I-cache performance is a bottleneck and function calls are frequent. Some of the more commonly studied benchmarks, such as CPU-2000, exhibit very little I-cache stall and there is no need to use CGP for such benchmarks. It would be interesting, however, to study the performance of CGP on media processing and other emerging applications.

7 Acknowledgements

References

[ADHW99] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. Dbmss on a modern processor: Where does time go? In *Proceedings of 25th International Conference on Very Large Data Bases*, pages 266–277, September 1999.

[BA97] Doug Burger and Todd M. Austin. The simplescalar tool set. Technical report, University of Wisconsin-Madison, Computer ScienceDepartment Technical Report #1342, June 1997.

[BBC+98] Philip A. Bernstein, Michael L. Brodie, Stefano Ceri, David J. DeWitt, Michael J. Franklin, Hector Garcia-Molina, Jim Gray, Gerald Held, Joseph M.Hellerstein, H. V. Jagadish, Michael Lesk, David Maier, Jeffrey F. Naughton, Hamid Pirahesh, Michael Stonebraker, and Jeffrey D. Ullman. The asilomar report on database research. *SIGMOD Record*, 27(4):74–80, 1998.

[BDT83]   Dina Bitton, David J. DeWitt, and Carolyn Turbyfill. Benchmarking database systems a system-
            atic approach. In *9th International Conference on Very Large Data Bases*, pages 8–19, October
            1983.

[BRK98]   Peter A. Boncz, Tim Rühl, and Fred Kwakkel. The drill down benchmark. In *Proceedings of
            24th International Conference on Very Large Data Bases*, pages 628–632, August 1998.

[CB94]   Z. Cvetanovic and D. Bhandarkar. Characterization of alpha axp performance using tp and spec
            workloads. In *Proceedings of the 21st Annual International Symposium on Computer Architec-
            ture*, pages 60–70, April 1994.

[CDF⁺94]   Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe,
            Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos,
            Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *Proceedings of the
            1994 ACM SIGMOD International Conference on Management of Data*, pages 383–394, May
            1994.

[CLM97]   I-C. K. Chen, C-C Lee, and T. N. Mudge. Instruction prefetching using branch prediction
            information. In *Proceedings of International Conference on Computer Design*, pages 593–601,
            October 1997.

[Cou99]   Transaction Processing Performance Council. Tpc benchmark h standard specification (decision
            support). In *Revision 1.1.0*, June 1999.

[EJK⁺96]   R.J. Eickemeyer, R.E. Johnson, S.R. Kunkel, M.S. Squillante, and S. Liu. Evaluation of mul-
            tithreaded uniprocessors for commercial application environments. In *Proceedings of the 23rd
            Annual International Symposium on Computer Architecture*, pages 203–212, May 1996.

[Fra94]   M. Franklin. Commercial workload performance in the ibm power2 risc system/6000 processor.
            IBM J. of Research and Development, 1994.

[LBE⁺98]   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. In *Proceedings of
            the 25th Annual International Symposium on Computer Architecture*, pages 39–50, June 1998.

[LM98]   Chi-Keung Luk and Todd C. Mowry. Cooperative prefetching: Compiler and hardware support
            for effective instruction prefetching in modern processors. In *Proceedings of 31st International
            Symposium on MicroArchitecture*, pages 182–193, December 1998.

[MDB94]   A.M. Maynard, C.M. Donnelly, and B.R.Olszewski. Contrasting characteristics and cache per-
            formance of technical and multi-user commercial workloads. In *Proceedings 6th international
            conference on Architectural support for programming languages and operating systems*, pages
            145–156, October 1994.

[NBC⁺94]   C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. Alphasort: a risc machine sort.
            In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*,
            pages 233–242, May 1994.

[PH90]   Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Programming Language
            Design and Implementation*, pages 16–27, June 1990.

[RBH⁺95]   M. Rosenblum, E. Bugnion, S.A. Herrod, E. Witchel, and A. Gupta. The impact of architec-
            tural trends on operating system performance. In *Proceedings of the 15th ACM symposium on
            Operating systems principles*, pages 285–298, December 1995.

[RCA99]   G. Reinman, B. Calder, and T. Austin. Fetch directed instruction prefetching. In *Proceedings of 32nd International Symposium on MicroArchitecture*, pages 16–27, November 1999.

[RVL$^+$97]   Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of win32/intel executables using etch. In *USENIX Windows NT Workshop*, pages 1–7, August 1997.

[SH92]   J. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Proceedings of Supercomputing'92*, pages 582–597, 1992.

[SKN94]   Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th International Conference on Very Large Databases*, pages 510–521, April 1994.

[Smi78]   A. Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11(2):7–21, 1978.

[SW92]   Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimization at link-time. Technical Report Technical Report 92/6, Digital Western Research Laboratory, June 1992.

[TLPZT97] Trancoso, J.L Larriba-Pey, Z. Zhang, and J. Torellas. The memory performance of dss commercial workloads in shared-memory multiprocessors. In *Procedings of the HPCA*, 1997.