COMPILER DRIVEN MEMORY SYSTEM OPTIMIZATION USING SPECULATIVE EXECUTION

HARIHARAN SANDANAGOBALANE

(B. Tech, Pondicherry Engineering College)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

APRIL 2004

Acknowledgements

Profound and sincere thanks are due to my supervisor A/P Wong Weng Fai for his excellent guidance, constant support and encouragement. Working with him has been a very pleasing experience, both personally and intellectually. I appreciate his help and support which were on many occasions, unexpected, but certainly very welcome. He has served as a good role model for a supervisor.

I would like to thank Dr. Rodric Rabbah from Massachusetts Institute of Technology for his valuable comments and guidance throughout the project.

I thank the members of the embedded systems research laboratory and my friends for giving me good company in Singapore. I would also like to thank David Chew, whose latex style files for NUS thesis made lexing a breeze.

Last, but not the least, I thank my family for their understanding and support.

Hariharan

April 2004

Contents

Summary

Wide-issue microprocessors are capable of remarkable execution rates, but they generally achieve only a fraction of their peak instruction throughput on real programs. This discrepancy is due to performance degrading events, largely branch mispredictions and cache misses. In this work we have addressed the performance degradation due to the latter through the use of Program Embedded Precomputation using Speculative Execution (PEPSE).

Our work on program embedded precomputation using speculative execution (PEPSE) aims at providing a unified framework to mitigate the ever-widening gap between the data processing rate of the processor and the data delivery rate of the memory subsystem. Towards this, we introduce the Load Dependence Graph (LDG), which is a sub-graph of the traditional Program Dependence Graph (PDG) that computes the address of a load instruction. The LDG affords a unique characterization of the program structure and its memory reference patterns and facilitates the discovery of appropriate memory management techniques.

In the context of data prefetching, we illustrate how PEPSE can accurately predict and effectively prefetch future memory references with negligible overhead for

both regular array-based applications as well as irregular pointer-based applications. We narrow down the scope of the optimizations by limiting our processing only to delinquent loads in a program, identified with the help of a profiler. LDGs are created only for those delinquent loads. Subsequently, speculative versions of the LDG operations are statically scheduled along with a prefetch instruction for the computed address, such that these instructions execute and prefetch the value before the actual load is encountered resulting in either an elimination or reduction of the processor stall cycles due to the load instruction. Our prototype implementation of the optimizations using LDGs within the Open Research Compiler (ORC), an open source compiler for the Itanium Processor Family (IPF), delivered encouraging results. For a 900 MHz Itanium 2 server, we could achieve speedups ranging from 1.05 to 2.14 for several benchmarks from SPEC and OLDEN suites.

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Out-of-order execution is the norm in current day processors. It is intended to allow processors to tolerate pipeline stalls due to data dependencies, resource conflicts, cache misses, etc., by buffering stalled instructions in reservation stations and executing other ready instructions out of program order. However, today's dominant application domains, including databases, multimedia and games, have large memory footprints and do not use processor caches effectively resulting in many cache misses. The resulting processor stalls degrade the performance of applications considerably.

Furthermore, exponential increases in processor speeds continue to widen the gap between the data consumption rate of the processor and the data delivery rate of the memory. High computation power becomes useless if it is not backed by a powerful memory system. Historically, the processor performances have been increasing at a rate of 35% per year till 1986, and 55% per year since then. On the other hand, the access time of DRAM has been improving at a rate of mere 7% per year [11]. Figure 1.1 illustrates the performance disparity between processor

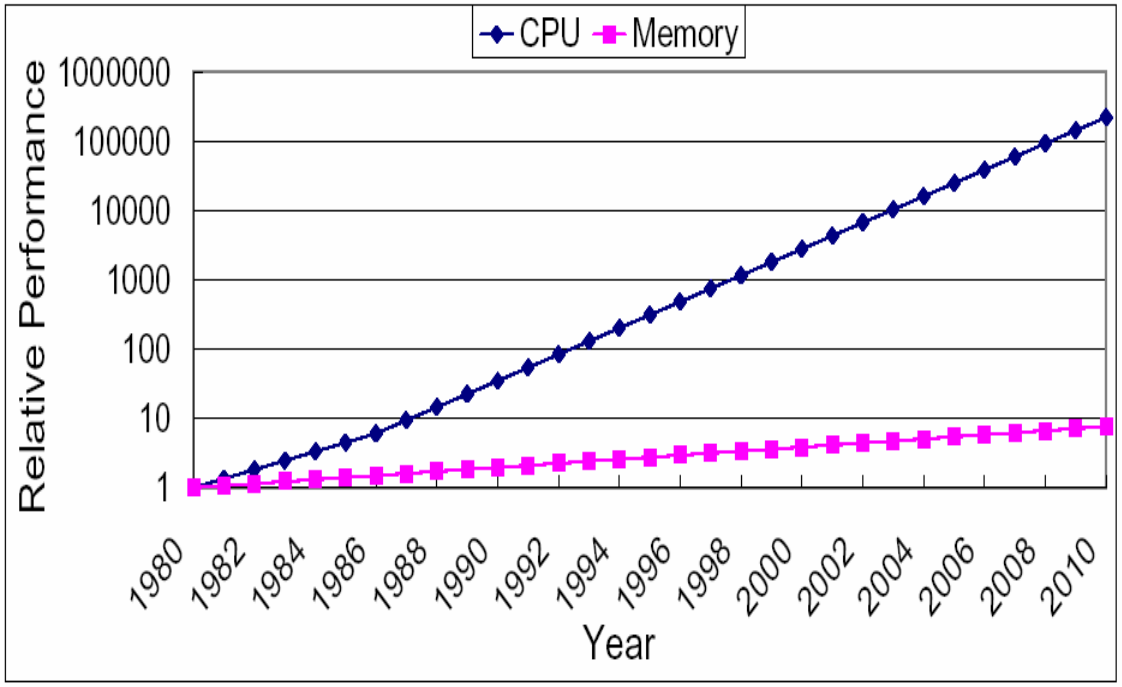

Figure 1.1: The processor and memory performance trends plotted over time.

and memory with 1980 performance as the baseline.

In order to solve this problem, cache memories are widely used. They take advantage of the locality of data accesses present in the programs. While deeper and wider caches help mitigate this imbalance, there still remains a significant gap in the ability of the memory systems to service data requests of the processor. The current trends, viz., clock speed acceleration and Instruction Level Parallelism(ILP) exploitation increase the delays between the processor and the memory. This is

especially true of the explicitly parallel instruction computing (EPIC) platforms which provide massive ILP. For example, the Intel Itanium processor consists of a three-level cache hierarchy: 32KB primary cache, 256KB secondary cache and tertiary cache as large as 6MB [24], with latencies ranging from 1 to 30 cycles [1]. It has a tertiary cache miss latency[1] in excess of 200 cycles. Such long access latencies degrade the processor performance and hence necessitate latency masking techniques.

Explicitly parallel processors have features derived from both VLIW and superscalar architectures. They use large instruction words and issue multiple instructions per cycle. They continue to gain wider acceptance and play a significant role in various aspects of the computer industry, ranging from the high end server platforms such as the Itanium Processor Family(IPF) [24], to digital signal processing engines such as the T1-C6x processors [12], to custom computing systems such as the Trimedia VLIW products [27] and the HP-STMicroelectronics Lx processors [23]. These EPIC processors expose the architecture to the compiler by extensions to the Instruction Set Architecture(ISA). The extensions enable the compiler to communicate with the hardware through hints attached to the instructions or through special instructions and hence allow them to manage the data movement across the memory hierarchy better.

During compilation, it is important to have the ability to predict the future memory accesses and the access patterns so as to utilize the EPIC's features to ameliorate the difference in performance between the processor and the memory system. This foresight would enable the compiler to make more informed decisions about the placement and evacuation of data in caches, which could be communicated to the hardware through the ISA. Towards this, a lot of hardware and software techniques have been proposed that prefetch the data ahead of its actual consumption,

[1]Tertiary cache miss latency is the latency due to a memory access.

resulting in a significant performance improvement.

Another orthogonal line of research towards reducing the memory bottleneck problem is to improve the data locality by reordering the execution of iterations. An important example of such a transformation is *blocking* [32, 31, 9]. Instead of operating on entire rows or columns of an array, blocked algorithms operate on submatrices or blocks, so that data loaded into faster levels of the memory hierarchy are reused. Other useful transformations include unimodular loop transforms like *interchange*, *skewing* and *reversal* [31]. These transformations complement *blocking* and hence can be used together with it to enhance the application's performance. Since these transformations improve code's data locality, they not only reduce the effective memory access time but also reduce the memory bandwidth requirement. Since these transformations aim at reducing the capacity misses, they complement prefetching methods which help reduce the cold misses that occur due to the first access to a data item. Hence, they can be used together to achieve even better performances.

1.2 Research Goals

The objective of our research is to provide a unified framework for alleviating the memory bandwidth bottleneck using static compilation techniques. The research goals that we set out for our work are

1. To devise an algorithm that would be effective for both array and pointer based programs.

2. The algorithm should only utilize the architectural features that are commonly available and should not require drastic changes to the underlying architecture.

3. The benefits of prefetching correctly should not be lost in the overhead or prefetching incorrectly.

4. The prefetching should be effective in improving the overall performance of the application.

1.3 Technique Overview

In this work, we explore the usage of Program Dependence graph(PDG) to predict the future memory accesses. We introduce the concept of Load Dependence Graph(LDG), which is a subgraph of the PDG that contains instructions that contribute towards the calculation of the load address. Typically, a small set of load instructions contribute to over 90% of the misses in most applications. We modify the code generation stage of the Open Research Compiler(ORC) to instrument the assembly code so as to couple the original program with Dinero IV cache simulator [10]. The output of the profiler is a detailed record of cache hits and misses for each static load, along with its contribution to the total program stall cycles.

We focus our attention to only loads identified as delinquent by the profiler. LDGs are created for these instructions by starting from them and moving up and including any instruction that contributes to their address calculation. Ideally, this LDG creation is stopped when it has moved a distance $\delta + \alpha$ from the delinquent load, where δ corresponds to the average latency of the load operation and α to the schedule length of LDG itself. But other constrains, such as explosion of LDG length and absence of enough free slots might stop it earlier. Program Embedded Precomputation via Speculative Execution(PEPSE) inserts a speculative version of the LDG instructions statically in the program along with a prefetch for the load in the empty[2] slots, as much as possible. These instructions would execute in

[2]NOPs are considered to be empty slots.

advance and bring the data closer to the processor, resulting in a reduced latency for the load.

We introduce a technique called **Induction Unrolling** to effectively prefetch for loads in loops. We also modify the induction unrolling technique to enhance the performance of pointer intensive programs dominated by pointer-chasing loops. A pointer chasing loop is characterized by a cyclic dependence between two loads. We implemented a prototype of our optimizations on Open Research Compiler and obtained promising results. Our proposed methodology relies heavily on speculation, a concept that is widely used to improve ILP and overcome long branch delays.

1.4 Thesis Overview

Chapter 2 gives a survey of the different techniques that have been proposed to address the memory bandwidth problem and show how our technique differs from them. Chapter 3 describes the Load Dependence Graph and details on how they are created and embedded in the application using PEPSE. Chapter 4 explains the implementation of PEPSE scheme in the Open Research Compiler. Chapter 5 discusses the experimental setup and the performance results obtained using PEPSE on an Itanium 2 machine. Chapter 6 concludes the thesis and gives pointers for future directions of research.

Chapter 2

Related Work

The speed of computer systems have been increasing steadily through the years. This is partly through the advancement of technology and partly because of the certain properties exhibited by the programs. The most important program property that is exploited is the *principle of locality*. Programs tend to reuse data and instructions they have used recently. A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code. An implication of locality is that we can predict with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past.

Principles of locality also applies to data accesses, though not as strongly as to code accesses. Two different types of locality have been observed [11]. *Temporal Locality* states that recently accessed items are likely to be accessed in the near future. This happens, say, when every iteration of an outer loop accesses the same set of items in the inner loop. *Spatial Locality* says that items whose addresses are near one another tend to be referenced close together in time. This happens when the loop has a sequential access along the data items placed contiguous to each other.

To exploit the locality in the programs, a small cache memory was added to the

processor. An access to the cache memory is an order of magnitude faster than a memory access, which is generally off the processor chip. But still, the addition of cache memory doesn't serve as a panacea to the *memory wall*[1] problem. This is because not all data accesses hit the cache and the misses would have to be served by the slower main memory and the processor might have to be stalled till the data item becomes available.

There are three kinds of cache misses : Conflict misses, Compulsory misses and Capacity misses [13]. Conflict misses are those that would be avoided by having a fully associative cache with LRU replacement. They occur because two data items conflict for the same cache line and hence the earlier one needs to be evacuated to give way for the latter, even though it may be accessed again soon. Capacity misses occur when cache is too small to hold data between references. Compulsory misses occur in every cache organization because they represent the first access to the data item. Past research on conflict misses have reduced them largely without resorting to fully associative caches, by the use of set-associative caches. The set-associative caches provide a trade-off between cache misses on the one side and the access time and energy on the other side.

To effectively reduce capacity misses, one has to either enlarge the cache or rearrange the program so that the working set would fit in the cache, both of which has been done to a large extent. Nowadays, the amount of on-chip cache is quite large and we have a hierarchy of caches so that the large caches do not increase the average memory access time. *Tiling* or *Blocking* [9] and *loop interchange* are commonly used compiler techniques to rearrange the memory accesses in the program to match the cache structure. But, some form of prefetching is required to minimize compulsory misses, also called cold misses. There are various hardware

[1]The problem of the memory system not being fast enough to serve the processor is commonly called the memory wall problem.

and software methodologies proposed to reduce the compulsory misses. We will review some of those methods in the following sections.

2.1 Hardware Techniques

The hardware prefetching methods were the first to be introduced and implemented. Long cache lines and hardware prefetching [16] are two of those hardware methods. With long cache lines, a cache miss results in the retrieval of data of one cache line size. Future loads might hit the cache now, even though they are the first accesses to that data item, if the data item happens to be in the same cache line. In hardware prefetching, an access to a cache entry invokes a prefetch to the address of the next datum in the address space, assuming it will be accessed in the near future. This method has the advantage of allowing sequential array accesses to be fetched with only one miss for the first item. Though both of these methods reduce the miss rate in a few circumstances, they cannot be disabled in other circumstances since they are implemented in hardware. For example, in case of array access in a loop with a high step size or a pointer chasing code with arbitrary memory access, both long cache lines and hardware prefetching would prefetch values that would not be used in the future. In such cases, it increases the data traffic between the cache and the main memory and also pollutes the cache with unwanted data.

In 1991, Baer and Chen [4] proposed a scheme that uses a history buffer to detect strides. In their scheme, a "look ahead PC" speculatively walks through the program, ahead of the normal PC, using branch prediction. The processor is extended with a Reference Prediction Table(RPT) which is used to keep track of previous reference addresses and associated strides. When the look ahead PC hits a load and finds a matching entry in this table, it issues a prefetch. They evaluated the

scheme in a memory system with 30 cycles miss latency and found good results. In the context of multiprocessors, Multiple-Context Processors [30] were introduced, where each processor maintains multiple processes as multiple contexts and switches between them when there is a long latency load in one context. In this manner the memory latency of one context can be overlapped with computation of another context. The interval between long latency operations is becoming fairly large, allowing just a handful of hardware contexts to hide most of the latency. But this method has the disadvantage of context switch overhead and the high processor complexity resulting from the inclusion of contexts in it. Also, since the different contexts share a single processor cache, they can interfere with each other, both constructively and more often, destructively.

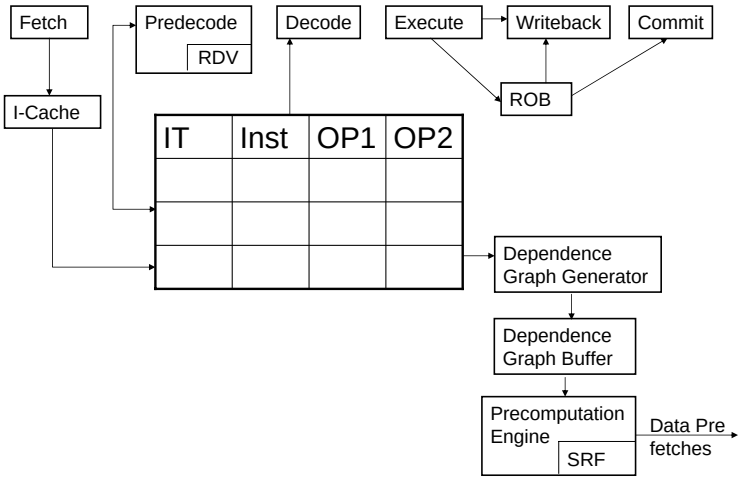

Figure 2.1: DGP hardware

More recently, Annavaram et.al. [22] have introduced an extension to the processor to pre-compute the load address and issue a prefetch. Figure 2.1 shows the

additional hardware required for this implementation. The fundamental idea of this method is to pre-compute the address of a load available in the Instruction Fetch Queue(IFQ), instead of predicting it, and then issuing a prefetch. The IFQ is extended(with extra columns) to help dependence graph creation and the pre-decode stage is also modified to fill in those extra columns. The dependence graph of a load/store instruction, I , in the IFQ is the set of all unexecuted instructions waiting in the IFQ, that contribute to the address calculation of I. The Dependence Graph Generator generates the graph based on the dependence information available in the OP1 and OP2 columns of IFQ, which contains pointers to the instructions that produce the values for operand one and two respectively.

The processor is augmented with a Precomputation Engine(PE) which is used to execute the dependence graphs stored in the dependence graph buffer. The PE executes instructions speculatively. The results generated by the PE are used only for prefetching data, and in particular, they never update the architected state of the main processor. Note that the dependence graph generation does not remove any instruction from the IFQ[2]: Consequently, all precomputed instructions will be executed in the normal manner by the main processor pipeline. The precomputation engine has a scratch register file(SRF) to store the live results of precomputed instructions. PE executes at most one instruction every cycle, and hence SRF needs only two read ports and one write port. If the OP field of an operand is not null, it would have been generated by an already executed instruction and hence available in the SRF. If it is null, the PE obtains the corresponding operand value by accessing the processor's register file and the Re-ordering buffer[3] for forwarding uncommitted register values.

In their work, Roth et.al. [3] also use an extra computation engine[4] to run ahead of

[2]It just makes speculative copies.

[3]The processor's register file and ROB each need two additional read ports for PE accesses.

[4]They call it prefetch engine.

the processor, executing only load instructions that are required to iterate through the Linked Data Structure. Dependence relationships between loads that produce addresses and loads that consume these addresses is exploited by constructing a compact representation for them and their traversal. To achieve prefetching, the prefetch engine speculatively traverses this representation ahead of the executing program. Since the prefetch engine executes only the loads that are required to traverse through the data structure, this engine initiates accesses faster, producing the desired prefetching effect.

Though some of the hardware techniques are effective in certain circumstances, they are not flexible. It would be hard to adapt the hardware technique to suit a given program. In the next section we review some of the software techniques for prefetching.

2.2 Software Techniques

2.2.1 Preliminary Work

Software prefetching was introduced by Callahan et.al. [8] and since then several prefetching algorithms [28, 33, 20] have been proposed and implemented. Software prefetching needs hardware support in the form of a special prefetch instruction, which would issue a non-blocking prefetch. The cache needs to be lockup-free [18], that is, the cache must allow multiple outstanding misses. Otherwise, an outstanding prefetch instruction might block a load instruction from the original program, degrading its performance. Also, this instruction should not affect the correctness of the program, viz., the insertion of prefetch should not raise exceptions or produce incorrect results, if the speculative address is wrong. These hardware supports are available in almost all processors nowadays, since, even with simple algorithms [5]

, prefetching is effective in overlapping the memory latency with other useful computation. Software techniques introduced in this section are compiler algorithms which insert prefetch instructions along with the original program to avoid the processor stalls due to memory accesses.

The first successful prefetching algorithm, which is implemented most commonly in compilers today, was devised by Mowry [28]. The domain of this algorithm is the set of array accesses whose indices are affine functions of loop indices. A substantial amount of data references in scientific code belong to this domain. There are three major steps in this prefetching algorithm.

1. For each reference, determine the accesses that are likely to be cache misses and therefore need to be prefetched.

2. Isolate the predicted cache miss instances through loop splitting. This avoids the overhead of adding conditional statements to the loop bodies or adding unnecessary prefetches.

3. Software pipeline prefetches for all cache misses.

The first step determines those references that are likely to cause a cache miss. This locality analysis consists of discovering data reuses within a loop nest and determining whether the set of reuses would be exploited by a particular cache configuration. The reuse could be one of spatial, temporal or group reuses. In the example program of figure 2.2a, there is a spatial reuse in the access of A[i][j] if the cache line size is larger than an array element size. There is also a temporal reuse of B[j][0] in the outer loop, viz., every time around the outer loop same elements of B array are accessed. But, whether this reuse would turn into a cache hits depends on the size of the cache and the iteration count of the inner loop. In this case, since the iteration count of the inner loop is small(100), this reuse would be converted

```
for ( i = 0 ; i < 3 ; i ++ )
   for( j = 0 ; j < 100 ; j ++ )
      A [ i ] [ j ] = B [ j ] [ 0 ]  +  B [ j + 1 ] [ 0 ] ;
```

a) Source Program

```
prefetch ( & A [ 0 ] [ 0 ] ) ;
for ( j = 0 ; j < 6 ; j + = 2 ) {
   prefetch ( & B [ j + 1 ] [ 0 ] ) ;
   prefetch ( & B [ j + 2 ] [ 0 ] ) ;
   prefetch ( & A [ 0 ] [ j + 1 ] ) ;
}
for ( j = 0 ; j < 94 ; j + = 2 ) {
   prefetch ( & B [ j + 7 ] [ 0 ] ) ;
   prefetch ( & B [ j + 8 ] [ 0 ] ) ;
   prefetch ( & A [ 0 ] [ j + 7 ] ) ;
   A [ 0 ] [ j ] = B [ j ] [ 0 ] + B [ j + 1 ] [ 0 ] ;
   A [ 0 ] [ j + 1 ] = B [ j + 1 ] + B [ j + 2 ] [ 0 ] ;
}
for ( j = 94 ; j < 100 ; j + = 2 ) {
   A [ 0 ] [ j ] = B [ j ] [ 0 ] + B [ j + 1 ] [ 0 ] ;
   A [ 0 ] [ j + 1 ] = B [ j + 1 ] + B [ j + 2 ] [ 0 ] ;
}
for ( i = 1 ; i < 3 ; i ++ ) {
   prefetch ( & A [ i ] [ 0 ] ;
   for ( j = 0 ;  j < 6 ; j + = 2 )
      prefetch ( & A [ i ] [ j + 2 ] ) ;
   for ( j = 0 ;  j < 94 ; j + = 2 )
      prefetch ( & A [ i ] [ j + 7 ] ) ;
      A [ i ] [ j ] = B [ j ] [ 0 ] + B [ j + 1 ] [ 0 ] ;
      A [ i ] [ j + 1 ] = B [ j + 1 ] + B [ j + 2 ] [ 0 ] ;
   for ( j = 94 ; j < 100 ; j + = 2 ) {
      A [ i ] [ j ] = B [ j ] [ 0 ] + B [ j + 1 ] [ 0 ] ;
      A [ i ] [ j + 1 ] = B [ j + 1 ] + B [ j + 2 ] [ 0 ] ;
   }
}
```

b) Resulting loop with prefetches inserted

Figure 2.2: A prefetching example

to a cache hit. There is also a group reuse between B[j][0] and B[j+1][0]. Elements accessed by the second would be accessed by the first in the next iteration.

The second step uses the locality analyzes of the first step to reorder the loop and split it between cache hit and cache miss iterations. The presence of temporal locality in a loop with index i means that prefetching is necessary only when i=0. The presence of spatial locality in a loop with index i implies that prefetching is necessary only when (i mod n)=0, where n is the number of array elements that would fit in a cache line. Prefetch predicates are defined for references and they determine if, in a particular iteration, that reference needs to be prefetched. Ideally, only iterations satisfying the prefetch predicate should issue prefetch instructions. To accommodate this,we can decompose loops into different sections so that the predicates for all instances for the same section evaluate to the same value. This process is known as loop splitting. In general a predicate i=0 requires the first iteration of the loop to be peeled. The predicate (i mod n)=0 requires the loop to be unrolled by a factor of n with only one prefetch. Peeling and unrolling can be applied recursively to handle predicates in nested loops. Figure 2.2b shows the result of applying these transformations to the loop-nest of figure 2.2a.

2.2.2 Prefetching methods for pointer intensive applications

One prefetching heuristic that works well for pointer based applications was introduced by Lipasti et.al [20]. In this, a prefetch instruction is inserted at the call site for every function call with at least one pointer parameter. The basic premise of this heuristic is that the pointer arguments passed on procedure calls are highly likely to be dereferenced within the scope of the called procedure. In this work, they had showed that with the insertion of just one or two prefetch instructions at each call site, performance can be improved by 5-7% for benchmarks with high

call sites and lower procedure lengths, without significantly increasing the memory traffic. This particularly works well for C++ programs since, in the xlC implementation of C++, the first argument is always the *this* pointer, which, intuitively has a very high probability of being dereferenced in the ensuing method call. But this work has a limited scope of prefetching only the pointers passed as parameters.

Youfeng [33] introduced another heuristic for prefetching in pointer-based applications. This is based on the fact that some important load instructions in irregular programs contain stride access patterns. Namely, the difference between addresses of two successive data accesses changes only infrequently at runtime. But these strides are impossible to identify with compiler techniques since the memory allocation is decided at runtime. In this work, they designed a new profiling method that integrates profiling for stride information and the traditional profiling for edge frequency into a single profiling pass. The collected stride information helps the compiler to identify load instructions with stride patterns that can be prefetched efficiently.

The work by Chi Keung Luk and Todd Mowry [19] analyzes the major issues and challenges involved in software-controlled prefetching for Recursive Data Structures(RDS) like lists, trees and graphs. In general, analyzing the address of heap-allocated objects is a very difficult problem for the compiler. They propose three possible solutions to overcome this problem.

1. In a k-ary RDS[5], all k pointers can be used in prefetching in the hope that the objects pointed to by the other pointers would also be used in the future.

2. The first traversal through the RDS can be used to create a history. The history would add an extra pointer to each node to indicate which node is to be prefetched from the current node. Subsequent traverses through the RDS

[5]Each node contains k pointers to other nodes.

would use this history information for prefetching and prefetch the address pointed by the added pointer.

3. The heap-allocated nodes that are likely to be accessed close together in time can be mapped into contiguous locations. This would also improve the spatial locality.

In recent times, multithreaded processors are becoming popular. There is an enormous amount of research interest to investigate if these extra threads could be used in improving the performance of single threaded applications. In the next section, we review some of those techniques which use a helper thread for prefetching.

2.2.3 Thread Based techniques

Despite the importance of mispredicted branches and loads that miss in the cache, a sequential processor is not able to prioritize these computations because it must fetch all computations sequentially, regardless of their contribution to performance. Alleviating this by spawning separate threads to execute only the delinquent operations and other instructions that contribute to them is the fundamental idea behind all thread based techniques.

Speculative Data Driven Multithreading(DDMT) was introduced by Amir Roth et.al. [25]. In DDMT, critical computations are identified with the help of a profiler and annotated, so that they can execute stand alone. When the processor predicts an upcoming instance of critical instruction, it microarchitecturally forks a copy of its computation as a new kind of speculative thread. This thread executes in parallel with the main thread, but typically generates results faster. These threads execute speculatively, they do not change the architected state of the machine though they may impact the performance of the application.

Collins et.al. [15] extend the thread based latency tolerance ideas of Amir Roth

[25]. In this work, they first identify delinquent loads[6] with the help of a profiler. Then the program is simulated on a functional Itanium simulator to create p-slices[7] for each delinquent load. Whenever a delinquent load is executed, the instruction that had been executed 128 instructions prior to it in the dynamic execution stream is marked as a potential basic trigger. This is achieved by keeping the most recent 256 retired instructions in a buffer and looking it up for the 128th instruction. The next few times that this potential trigger is executed, the instruction stream is observed to verify that the same delinquent load is executed somewhere within the next 256 instructions. If the potential trigger consistently fails to lead to the delinquent load, it is discarded. Otherwise, if the trigger consistently leads to the delinquent load, the trigger is confirmed and the backward slice of instructions between the delinquent load and the trigger is captured. Instructions between the trigger and the delinquent load constitute potential instructions for constructing the p-slice. Those unnecessary to compute the address are eliminated.

In addition to these basic triggers, they use chaining triggers, which allows one speculative thread to explicitly spawn another speculative thread. A key feature for applying chaining triggers is the presence of stride in addresses consumed by a load that is a dynamic invariant whose value is fixed for the duration of the loop. Thus p-slices containing chaining triggers typically have three parts - a prologue, a spawn instruction for spawning another copy of this p-slice and an epilogue.

Most of the thread based techniques differ only in the way threads are created and how they are triggered. On the one end, researches [17] propose a source-to-source C compiler that extracts p-slices, reducing the dynamic hardware required. On the other end, in long range prefetching technique [15], p-threads are constructed spawned, improved upon, evaluated and possibly even removed, entirely by hardware. In either case, some amount of hardware support is required, in the form of

[6]Loads that have the largest impact on performance.

[7]Precomputation slices

threads and their spawning mechanisms. Though the thread-based techniques are generally effective in accurate address generation[8] and timely prefetching, it comes at a high hardware overhead. Also, since threads are delinked from the original program, their scheduling becomes a problem. Giving low priority to them might not let them fetch the required values in time. Giving them a high priority might slow down the original program. One more problem with thread-based prefetching techniques is the non-determinism introduced in the instruction cache behavior because of addition of a new thread(s), which may interact with the original thread both constructively and destructively.

A combination of hardware and software techniques was used by Abraham et.al. [26] to predict the latencies of load/store instructions and subsequently use them to improve performance of the application. This method requires that the ISA have instructions that permit the software to manage the cache, e.g., DEC Alpha. In addition to the standard load/store operations, the architecture needs to provide explicit control over the memory hierarchy. For example, there could be two modifiers associated with each load operation specifying which level in the memory hierarchy is this load is likely to be found and another to specify which level the loaded value should be placed. These hardware support are becoming increasingly common in commercial microprocessors. In this work, they use profiling to get the memory referencing behavior of individual machine-level instructions. The information gained by the compiler through profiling can be passed on to the hardware by annotating the instructions, viz. adding values to these modifiers. If the compiler is unable to gain this information, these modifiers are set to a special *nta*[9] value, which specifies that no information is available. This allows for a mixed compiler/hardware control over the cache hierarchy where the compiler interferes only if it has some insight into the program behavior.

[8]They are precomputation based not prediction based

[9]Not available

2.3 Application Restructuring

Instead of using either hardware or software methods to effect a prefetch, there are techniques that have been proposed for restructuring the program to modify its cache behavior. One such methodology is detailed below.

A method of creating and utilizing the cache hit/miss heuristics and utilizing that in the amelioration of memory latency bottleneck was introduced by Toshihiro et.al. [29]. In this work, they have developed simple compiler heuristics to identify load instructions that are likely to cause a cache miss. Firstly, the loads are classified into either list accesses, stride accesses or others. List access refers to a load instruction whose load address comes from another load instruction, which is typical of pointer-chasing. Stride access refers to loads in a loop with constant or variable address increment. For every load that falls into either one of these two classes, there is a high probability of a cache miss. Hence the compiler tries to insert sufficient instructions between the selected load instruction and instructions that use the loaded data by one of the following three ways: selected load instruction and its address calculation are moved up or the instruction that uses the loaded data and its dependents are moved down or instructions not related to this load are moved between the load and its use. These moves are allowed to cross basic block boundaries. This, in effect, would reduce the stalls due to the load since there are computations inserted in between, which are independent of the load.

2.4 Limitations

All the above said methods fall short of the proposed PEPSE, which

- Provides a unified framework for prefetching in both scientific and pointer-intensive applications using well known concepts of speculative execution and Program Dependence Graph(PDG).
- Ensures accurate and timely precomputation of the load addresses and hence does not issue unnecessary prefetches.
- Does not require any special hardware to implement.
- Has little resource overhead, since it utilizes the available unutilized resources in the architecture.

Chapter 3

LDG and PEPSE

In this chapter we elaborate on our proposed methodology. First, we explain the concept of Load Dependence Graph(LDG). Then we explain the Program Embedded Precomputation using Speculative Execution(PEPSE), our technique to embed the speculative program slices along with the original program. Throughout this chapter, we assume that the reader is familiar with the standard control and data flow analysis techniques.

3.1 Load Dependence Graph

The concept of Program Dependence Graph is well established in the compiler arena. At compile time, validity of operations are governed by the dependencies that need to be followed. If a transformation would disrupt a dependence, then it would not be allowed. A typical compiler would construct the data and control dependence graphs before it begins optimizing code, as these graphs are essential for verifying if certain transformations are possible on the code. In the following subsections, we show how the concept of PDG can be used to extract the subset of a program which computes the address of a load, the Load Dependence Graph.

3.1.1 Delinquent Load Selection

Callahan et.al [5] show that, on an average, an application spends about one-third of its execution time waiting for cache miss(for a memory latency of about 50 cycles). The current trends in processor design increases this even further. Also, they [5] observe that a small percentage of the references cause majority of the misses in the programs. To validate these claims, so that we could focus our optimizations to only a few delinquent loads in a program, we profiled various programs to find the number of loads that account for more than 90% of the misses. Empirically, we modelled different memory system architectures including the Pentium4, Itanium and Itanium 2, and we overwhelmingly found that a very small number of load instructions cause more than 90% of the data stalls incurred by the processor. The results are shown in Table 3.1. This characteristic allows us to focus the memory system optimizations to a small subset of the total load instructions in the program.

Our framework identifies the delinquent loads in a program using profiling, a technique that is becoming popular in feedback driven optimizations. We generate the profile information by instrumenting the code generated by ORC to couple it with the Dinero *IV* cache simulator [10]. The simulator allows various parameters of each cache to be set separately (architecture, policy, statistics). During initialization, the configuration to be simulated is built up, one cache at a time, starting with each memory as a special case. After initialization, each reference is fed to the appropriate top-level cache by a single simple function call. Lower levels of the hierarchy are handled automatically. The simulator is trace driven, viz., it works on the traces of memory accesses generated by the program. The loads in the program are identified with the help of a centralized identifier generator which initializes a new identifier for all the memory operations in the program. This identifier along with the reference address are passed as parameters to the cache

Benchmark	Total Number of Static Loads	Number of Delinquent Loads
132.ijpeg	5079	43
164.gzip	1226	9
175.vpr	5289	30
181.mcf	515	14
183.equake	945	30
188.ammp	776	3
197.parser	4368	6
255.vortex	21298	361
256.bzip2	1064	28
300.twolf	10695	99

Table 3.1: Number of static load instructions accounting for more than 90% of the memory stalls (assuming an Itanium 2 processor and memory hierarchy configuration.)

simulator.

When the instrumented code is run with the simulator, it produces the statistics of the hits and misses of the program in the memory hierarchy. For each load, we compute the total stall cycles caused by that load,

$$Total\ Stall\ Cycles = \sum_{n} number\ of\ accesses\ *\ latency_n \tag{3.1}$$

where $latency_n$ is the latency of a particular cache level/main memory. This gives the total performance degradation of the application due to this load. After sorting the loads according to their total stall cycles, we pick up the top 5% of them for our analysis.

Since our methodology is profile driven, we recognize the importance of addressing the issue of profile sensitivity to different input workloads. This is to check if the set of delinquent loads for an application remain relatively constant across different inputs. For our work, we used the distributed training input(*train*) to profile applications. All our reported results in the later sections are collected using the ref input set(*ref*). Though we would expect the set of delinquent loads to be dependent on the workloads distributed with the program and also on the program's characteristics, we have observed that the set of delinquent loads does not vary much among the different input workloads.

3.1.2 LDG Creation

We use the concept of PDG to create Load Dependence Graph(LDG), which is a program slice of the set of instructions that contribute to the address calculation for the load instruction. The LDG creation starts with the delinquent load and moves up, including any instruction that produces results that any of the existing LDG instructions is dependent on.

Ideally, the last instruction of the LDG(the prefetch instruction) should be initiated δ cycles before the actual load is encountered, where δ is the average latency of the load instruction. This would prefetch the address just in time for the load instruction. But to achieve that, the LDG has to be started $\delta + \alpha$ ahead of the load, where α is the schedule length of the LDG. This may not always be possible because the LDG creation would have to be stopped if one the following happens.

- The LDG creation encounters a function call. Inter procedural analysis is beyond the scope of this work, though it remains an interesting topic to explore. Since we cannot determine the effect of the procedure call on the LDG instructions, we stop the LDG creation.
- The length of LDG increases beyond a predefined limit. This would ensure that the program embedding of speculative LDG instructions does not drastically increase the static length of the program.
- When the current block is the first region or if all the predecessor blocks are visited, then the LDG creation is stopped.

If the LDG creation has to be stopped prematurely because of one of the above reasons, then the insertion of LDG would not be able to fully absolve the load latency. But, it is still effective in reducing the latency of the load instruction.

While building the LDG, the LDG creation algorithm is allowed to cross basic block boundaries. In this case, a path specific LDG would have to be created for each of the incoming paths. Without some kind of path profiling and pruning, the number of path-specific LDGs would be excessively large. For this, we use the branch profile and create path-specific LDGs only for incoming edges with atleast 20% edge frequency, meaning that a branch edge must have been taken atleast 20% of the time to be considered for a path-specific LDG.

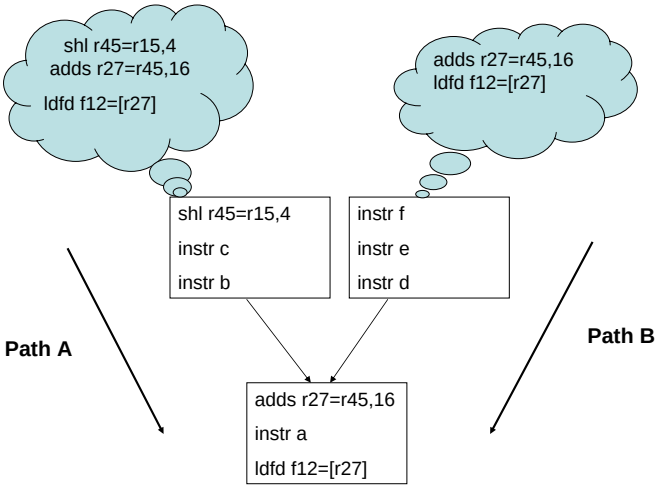

Figure 3.1: An LDG example

Figure 5.1 depicts the construction of LDG for a simple program. In this figure every instruction that is unrelated to the load is referred to simply to as instr(a, b, c, d and e). When the LDG creation algorithm hits at the end of the basic block, it has to start creating path-specific LDGs for the two incoming paths. For this example, we assume that both the incoming edges are frequently taken. For the two paths A and B, path specific LDGs are created as shown in the two cloud structures attached to them.

To effectively mask the load latency, the first LDG instruction must be scheduled as far before the load as possible. But when the LDG creation moves up, it would include more instructions into it. This would mean that we would have to move further up to fully overlap the latency of the inserted instructions. Though this might look like a vicious loop, in practice, after we move a few instructions above the load, we hit upon instructions unrelated to the load. This would generally provide a “Sweet Spot” to place the instructions.

The LDG described in this section is the program slice for the computation of a load address. Program Embedded Precomputation using Speculative Execution (PEPSE) embeds a speculative version of this and schedules it alongside the original program and ensures the timely availability of the loaded value.

3.2 PEPSE

We perform the PEPSE after pre-pass scheduling. As we assume that some scheduling has already taken place, we note the following

- Each function consists of a set of blocks or regions.
- Each operation i in a block is a member of a unique instruction word w_i. The bundled operations will be issued in parallel.
- The schedule time of an operation i is the schedule time of the bundle w which contains this operation.

The effect of compiler phase ordering problem on LDG is beyond the scope of this work.

The effectiveness of the prefetch algorithm depends on its ability to issue the prefetch enough cycles ahead of the actual load so that it can mask the load latency completely. Towards this, PEPSE tries to schedule the instructions of LDG as tightly as possible. Figure 3.2 shows the steps involved in scheduling the LDG instructions. This algorithm assumes that the delinquent loads have been identified and the LDGs are constructed for them in previous stages. Note that the destination registers have to be changed for the LDG operations to make them run speculatively. Otherwise, they would interfere with the correctness of the original program. This mapping information is maintained in a *map* data structure, which

is used to change the source registers of subsequent instructions that may use the changed register value.

Input: function f, LDG and the operation c where scheduling is to begin
Ouput: function f with LDGs

1. Perform register live range analysis
2. Create a map and initialize it to be empty.
3. Process each operation j in the LDG from head to tail
4. Find the earliest available scheduling slot occurring at time t, $t>t_c$ along the visited blocks, where tc is the schedule time of the last scheduled LDG instruction.
5. d ← destination operand of j.
6. find an available register r.
7. use r as the new destination register for j.
8. for each source operand s of j do
9. if s is in map then replace s with map(s)
10. end for.
11. map(d) ← r

Figure 3.2: The scheduling algorithm

We perform the LDG insertion just after pre-pass scheduling and before the register allocation. Hence we use the compiler's register allocator to allocate registers for LDG operations. If the register allocator runs out of registers, it will insert register spill and restore operations as it would for the registers used by the original instructions in the program. But,we observe that in almost all cases, we successfully scheduled and register allocated the LDG instructions without (i) increasing the

static schedule length of a block or (ii) significantly increasing register pressure[1]. Finally, the load instruction is changed into a prefetch. A prefetch instruction would not have a destination register since it would only try to get the data closer to the processor by placing it in the primary cache.

3.2.1 Optimizations

Pruning the list of LDGs

The initial delinquent load selection is based on the cache hit/miss statistics derived from profiling. For all loads identified as delinquent, LDGs are created. But, we evaluate the effectiveness of LDG in masking the latency and the resources available to eliminate non-profitable LDGs or LDGs with substantial resource requirements. For a load i, the following heuristic is used to compute the LDG's benefit factor α_i.

$$\alpha_i = \frac{d_i \ * \ available\ resources}{\mid LDG \mid} \tag{3.2}$$

where d_i is the dependence distance between the starting point of LDG[2] and the load instruction, available resources refers to the amount of free slots available in this part of the code and $\mid LDG \mid$ refers to the size(latency) of the LDG itself. As it is clear from the above equation, we would give higher priority to LDGs that (i) have higher distance d_i which would enable better masking of the load latency, (ii) has more free resources available in which case the LDG insertion would not need much additional resources and (iii) have less instructions, otherwise, it would lead to static code explosion. We use the above equation as a guide to maximize the performance gains due to LDG insertion without increasing the overhead.

[1]The processors in the Itanium family contain 128 registers and hence a slight increase in the register presssure does not adversely affect the performance.

[2]The location in the original source code from where the LDG scheduling is to start.

Loop Optimizations

Since LDG, in principle, is similar to a PDG, all transformations that are available to the PDG are applicable to the LDGs also. Cyclic dependence between two loads in a loop, for example, would indicate that the load is a part of a pointer-chasing loop. We observe that most of the delinquent loads in a program are located in tight loop nests. If the delinquent load is present in a straight line code, it is generally a small procedure[3] that is called from a loop. But since our current implementation does not include interprocedural analyzes, identifying these LDGs is out of our scope.

The PEPSE methodology is to identify the load dependence graph for a delinquent load l, and statically schedule speculative equivalents of the LDG operations in the original program. Ideally the distance between the last LDG operation and the load instruction should be equal to the average miss latency for l. In a cyclic program region, it is often the case that the prefetch is necessary in some iteration k in order for the data to arrive in time for processing in a future iteration m. The LDG lends itself well for such purposes. Unrolling of a LDG contained in a loop, for example, is very simple and straightforward.

In case of loops, we perform a LDG transformation called Induction Unrolling. Initially, the LDG is created for the load by the normal procedure, but it is kept within the loop's limits. This LDG alone(and not the whole loop) is then unrolled n times, where n is the loop distance by which we want to prefetch. The unrolling factor is ideally equal to $\lceil L_l/C \rceil$, where L_l is the average miss latency of the load and C represents the critical path length of the loop(i.e. the longest path from the start of the loop to its exit operation).

Figure 3.3 shows an example of the transformations performed for induction unrolling. Figure 3.3a shows the original loop. The loop might have other instructions

[3]For example, a sin function called to calculate the value in the innermost loop.

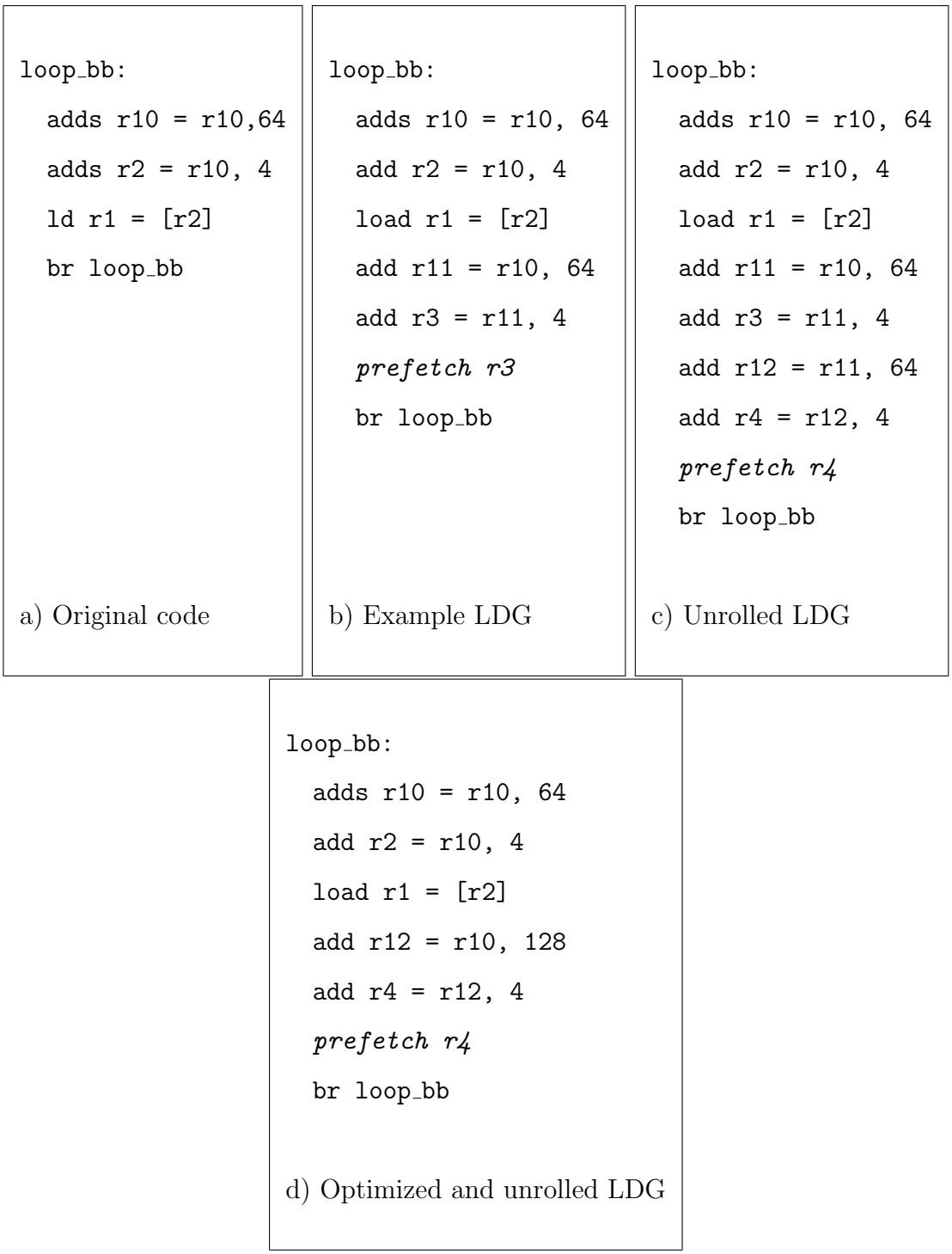

Figure 3.3: Example load dependence graph for a simple loop construct.

unrelated to the load, which are not shown in the figure. The access pattern shown in the loop is similar to accesses in array of structures or multi-dimensional arrays. Figure 3.3b shows the program with LDG operations (the last 3 operations before the *br*) inserted. If the original loop has some available resources to accommodate the precomputation and the prefetch operation, then the critical path of the loop would not be lengthened. When the loop body executes, the embedded speculative operations will initiate prefetch requests one iteration ahead of the actual loop. For a longer prefetch distance, the LDG, consisting of a two add instructions in this example, is unrolled as necessary. Figure 3.3c shows the result of unrolling the LDG two times to precompute the memory addresses two iterations ahead of the host loop region. In addition to unrolling, we can also apply other optimizations like constant folding and dead code elimination to achieve a more compact LDG. Figure 3.3d shows the result of applying these optimizations to figure 3.3a.

3.2.2 Pointer Applications

In the previous section, we described a method of prefetching for loop structures, namely the induction unrolling. Though the induction unrolling technique is effective in prefetching for array structures, it is not effective for pointer-chasing code in the given form. An example of that is shown in figure 3.4. Here, figure 3.4a shows a pointer chasing loop. A pointer chasing loop code generally contains two loads that are cyclically dependent on each other. The computations that are not related to the load are removed for simplicity. For this example, let us consider the second load to be delinquent. Figure 3.4b shows the result of attaching the LDG instructions to the original loop body. The added instructions would try to look ahead and prefetch the load one iteration ahead of time. Note that the LDG would also contain a copy of the original delinquent load. But that load is converted to a prefetch in this figure.

In case of unrolling the LDG for an array based code, the delinquent load instruction(which is the first instruction to be added to the LDG) is not required to calculate the load address for a future iteration. Generally, the load in an array based code loads the value into a local variable which is either used for some calculations or used to update another array. But in pointer based code, the value that the delinquent load loads is necessary for the address computation for the next iteration. This is because of the loop carried cyclic dependency present in pointer-chasing loops. Hence the unrolled loop shown in figure 3.4c is full unrolling of LDG(including the delinquent load instruction) and then the final load instruction is converted to a prefetch instruction.

As shown in figure 3.4c, to look-ahead in a pointer code by a few iterations, the unrolled LDG would have a few loads in it, which cannot be compacted any further. And these loads themselves might miss the cache, in which case, the LDG might degrade the performance of the application.

We change our Induction Unrolling technique to accommodate pointer-chasing code. Generally, for pointer chasing, the lead is available in the basic block that is executed just before the loop. We first identify the predecessor basic block for the loop containing the delinquent load. This is the last basic block visited by the program before the start of the loop. We have not encountered situations when the loop(which iterates through the list) is preceded by more than one basic block. In that case, we would have to place the prologue(explained below) in each of the predecessor basic blocks.

Figure 3.5 illustrates the mechanism. We place $\lceil L_l/C \rceil$ unrolled iterations of the LDG in the predecessor basic block(s) of the loop, where $\lceil L_l/C \rceil$ is the desired prefetching distance (as defined in the previous section). This serves as a prologue to the LDG in the loop. The loop itself contains only one iteration of the LDG. Hence, during every iteration of the loop, one iteration of the LDG and prefetch

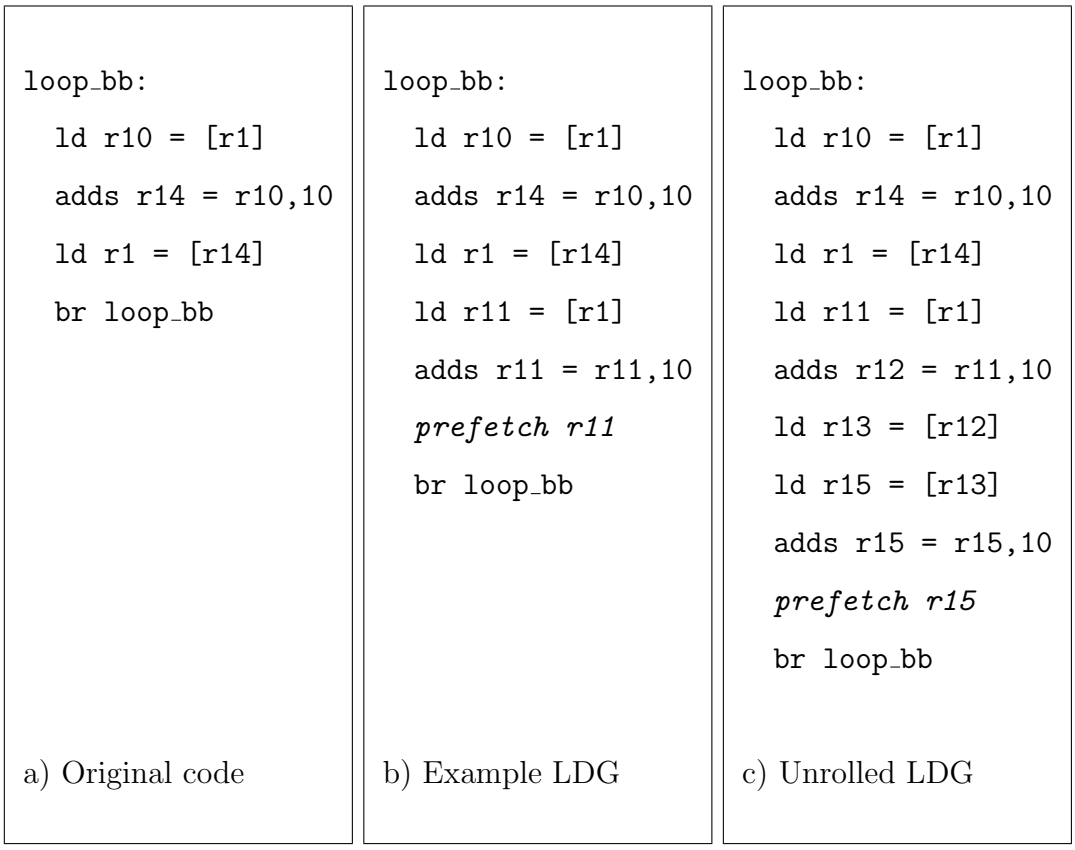

Figure 3.4: Example load dependence graph for a simple pointer-chasing loop construct.

are executed, which would prefetch data that would be accessed in some future iteration. Note that the data prefetched during the last few iterations would not be useful. But in case of pointer chasing loop which ends with a null pointer, the prefetch would be quashed since it would try to prefetch a null value. In other times the amount of unwanted prefetch is still very less and hence ignored.

This acts as an effective technique for prefetching in pointer-chasing loops. But, as explained above, the inserted LDG instructions contain load instructions, which might incur a miss themselves. But these are misses that would anyway have occured during the original execution of the loop. If the loop is big, some of this

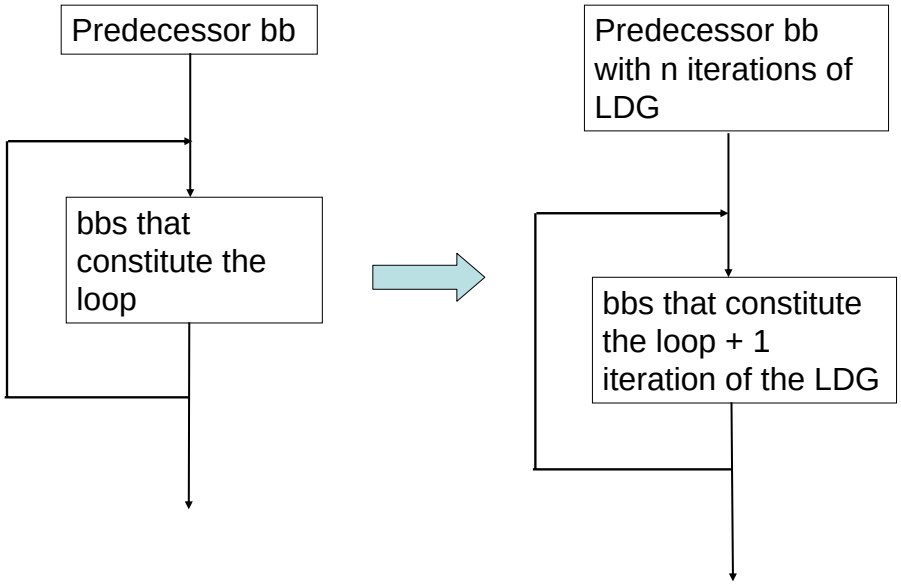

Figure 3.5: Induction Unrolling for Pointer-chasing code

miss latency can be reduced by placing the load and its use as far away as possible, exploiting the fact that the processor implements a stall-on-use[4] technique. Also, since the prefetch brings in a whole cache line, accesses to other members of the structure would also hit the cache. Squashing the LDG instructions when a load in the LDG misses, remains an important direction of future research. This could help eliminate the overhead that may result from the LDG without affecting its efficacy in favorable situations.

[4]The processor is stalled not when a load misses the cache, but only when the load value is needed by another instruction

Chapter 4

PEPSE Implementation

This chapter explains the implementation of the PEPSE scheme in the Open Research Compiler. For a reader uninterested in the implementation details, this chapter can be skipped without any loss of continuity. The next chapter gives a detailed description of the results achieved by our PEPSE implementation on the Open Research Compiler. This chapter is also intended to serve as a reference for researchers working at the code generation stage of Open Research Compiler.

4.1 Open Research Compiler

Open Research Compiler(ORC) is an open source compiler for the Itanium Processor Family(IPF) developed by researchers in Intel and Chinese Academy of Sciences. It is a sequel to the Pro-64 compiler from Silicon Graphics(SGI). The Pro-64 compiler was originally targeted for the MIPS processor. It was changed to retarget it for the IPF. ORC includes a comprehensive set of optimizations which include *blocking*(tiling), *loop unrolling*, *software pipelining*, *if-conversion*, *data prefetching*(based on Mowry et.al.[28]) and a *global instruction scheduler* integrated with a finite-state-automaton-based resource management.

ORC provides a common robust infrastructure and is modular, which enables quick prototyping of new ideas. For our implementation, we were concerned only with the Code Generation(CG) module of ORC. ORC provides separate compilation for different modules which makes it easier to locate errors in a particular module. It uses region based compilation, where the regions act as boundaries for the optimizations. This enables better management of compilation time and space, since only regions considered important would have to be fully optimized. ORC has the leading performance amongst the open source compilers for the IPF. It provides front-ends for C/C++, fortran77 and fortran 90.

The abstract syntax tree based intermediate representation used by ORC for its optimizations is called *Whirl*. Most of the interprocedural optimizations like aliasing analysis, call tree, function inlining, dead function elimination and loop nest optimizations like loop distribution, unimodular transformations and blocking are performed on the *whirl* representation of the original program. This intermediate representation was a legacy from the Pro-64 compiler. But the code generation stage of ORC uses a register based Intermediate Representation(CGIR). Most of our work was confined to the code generation stage and hence we use this representation.

The structure of an operation in this representation is shown in figure 4.1. Most of the fields in the structure are self explanatory. The *scycle* of an operation is set by the scheduler to indicate the start cycle of the operation and *order* shows the order of the operation in the basic block. A basic block contains a set of ops connected using the prev and next pointers. Since the code generation stage is the compiler stage that creates the assembly code, we can note that the structure of an operation in this representation encompasses all the information required to produce an assembly instruction.

ORC contains functions to create and manipulate OPs, BBs and dependence edges

```
SRCPOS      srcpos;          /* source position of the OP */
OP          *next;           /* Next OP in BB list */
OP          *prev;           /* Preceding OP in BB list */
struct bb   *bb;             /* BB in which this OP lives */
struct bb   *unroll_bb;      /* BB just after unrolling */
mUINT16     order;           /* relative order in BB */
mUINT16     map_idx;         /* index used by OP_MAPs*/
mUINT16     orig_idx;        /* index of orig op before
unrolling*/
mINT16      scycle;          /* Start cycle */
mUINT32     flags;           /* attributes for OP */
mTOP        opr;             /* Opcode. topcode.h */
mUINT8      unrolling;       /* which unrolled replication */
mUINT8      results;         /* Number of results */
mUINT8      opnds;           /* Number of operands */
mUINT8      flag_value_profile; /* flag for value_profile */
mUINT32     value_profile_id;   /*ID for value profile No. */
mUINT64     exec_count;         /* Execution count */
struct tn   *res_opnd[10];      /* result/operand array */
```

Figure 4.1: Structure of an Operation

between the OPs. It also contains iterator classess and functions to walk through the regions within a procedure, OPs within a BB, etc. In both our profiler and PEPSE implementations, we have heavily used these functions and iterators.

Itanium architecture incorporates an advanced mechanism of register stacks to avoid the unnecessary spilling and filling of all general purpose registers at procedure call and return interfaces through compiler-controlled renaming. This technique is important in our implementation and hence we describe it in the next few paragraphs.

At a call site, a new frame of registers is made available to the called procedure without the need for register fill and spill(either by the caller or by the callee). Register access occurs by renaming the virtual register identifiers in the instructions through a base register into the physical registers. The callee can freely use

available registers without having to spill and eventually restore the caller's registers. The callee executes an *alloc* instruction specifying the number of registers it expects to use in order to ensure that enough registers are available. This frame of registers is allocated by the hardware from the register stack. If sufficient registers are not available(stack overflow), the *alloc* stalls the processor and spills the caller's registers until the requested number of registers are available.

At the return site, the base register is restored to the value that the caller was using to access registers prior to the call. Some of the caller's registers may have been spilled by the hardware and not yet restored. In this case(stack underflow), the return stalls the processor until the processor has restored an appropriate number of the caller's registers. The structure of an *alloc* statement is shown below.

$$\text{(qp) alloc } r_1 = \text{ar.pfs, i, l, o, r}$$

At the execution of the alloc instruction, a new stack frame is allocated on the general register stack, and the Previous Function State register is copied on to GPR[1] r_1. The change of register frame is immediate at the execution of this instruction. The write of GPR r_1 and the subsequent instructions use the new frame. The four parameters i, l, o, and r specify the number of input, local, output and rotating registers being used in this procedure respectively. Note that most of the instructions in Itanium are predicated. The qp in the above instruction is the qualifying predicate register.

4.2 Profiler Implementation

To get the profile information of the loads, we needed to couple the original program with Dinero IV cache simulator. This is achieved by inserting calls to the simulator

[1]General Purpose Register

modules from the original program. The simulator program, compiled separately, is then linked to this program. The simulator is trace driven, viz., it examines the trace of memory addresses accessed by the program and simulates them for the given cache configuration. The simulator needs the following information from the original program. (i) The address accessed by the instruction, (ii) The identifier for the instruction (This is used to generate the hit/miss statistics for each static load in the program), (iii) The size(in bytes) of the data access.

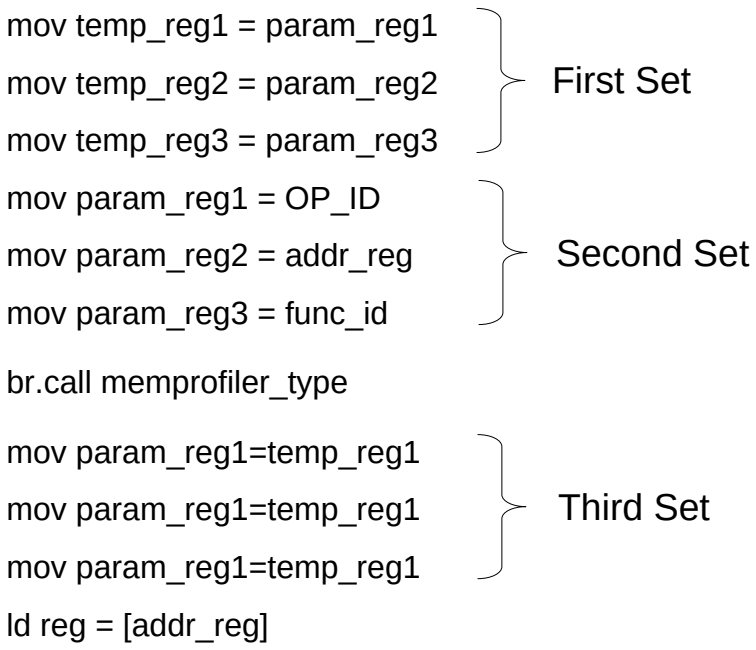

Figure 4.2: Profiler Implementation

We achieve the above said goal by inserting 10 extra instructions for every memory access operation. This mechanism is depicted in figure 4.2. The extra 10 instructions are the 3 mov instructions moving the parameters to the appropriate

parameter registers, the procedure call instruction, 3 instructions each for saving up and restoring the parameter registers' values before and after the procedure call. We would call the sets of move instructions first, second and third set respectively in the subsequent discussions. Note that the calls contain the operation identifier, function identifier and the address of the access as parameters. Since the profiler provides different calls for different types of memory accesses, size of the data access can be derived from the procedure called. The saving and restoring of the values held by parameter registers is necessary to avoid the parameters for the memprofiler modules (the procedures in the cache simulator have the name memprofiler_type, where type represents the type of the memory instruction) being sent in as parameters to any other normal procedure call that follows, but whose parameters were assigned before our simulator call.

The challenge of allocating unique identifiers for every memory access operation was achieved by attaching an extra *map* structure to map each memory access operation to an identifier. Every time a new memory access operation is created, a new member of this structure is created with the next higher identifier. The uniqueness of the identifier is maintained across various procedure calls in a source file and also across various source files within the same application.

Since the instructions are inserted before the register allocation stage, the temporary registers can be freely used. The register allocator then tries to map the virtual register identifiers to physical registers which is limited in number. In the Itanium architecture, the number of registers that a procedure can use is limited to 128 registers. So, sometimes, when the register pressure in the original program is very high, the insertion of these extra ops which use temporary registers might compel the register allocator to spill and restore some of these or other registers. To avoid the extra spills and restores, we used the branch registers to hold the values contained in the parameter register temporarily. In Itanium processor, there

are 8 branch registers(b0 to b7) and only b0 is used to hold the return address. The other branch registers are unused and are intended for future extensions. We used those registers to hold the values of parameter registers temporarily and to restore them to their original values.

Note that for every memory access instruction in the original program, there is an overhead of 10 inserted instructions along with a procedure call. Hence the running time of the original program for profiling would increase considerably. The simulator modules basically check to see if the address provided would hit or miss the cache configuration that it simulates and adds the corresponding hit/miss statistics for the identifier provided.

There are a few optimizations that ORC performs for *leaf* procedures. *Leaf* procedures are those without any procedure calls in them. Some of the status registers need not be saved up and restored in them. But if these procedures contain memory operations, we insert procedure calls to memprofiler modules(which changes the *leaf* status of the procedure). Hence, we change ORC to consider all procedures to be non-leaf procedures. Also, the calls to memprofiler requires three parameter registers(output registers). If the original procedure has procedure calls exceeding three parameter registers, those registers can be used by these additional calls. But, if the procedure does not contain any procedure call, or if it contains procedure calls with less than 3 parameters, we change the register allocator to allocate a minimum of 3 output registers. This would be reflected in the *alloc* instruction for the procedure.

Though the inserted instructions are aligned properly at insertion, as shown in figure 4.2, the local instruction scheduler and register allocator can change the order of these instructions, or worse still, delete some of the operations. To avoid this, we need to insert dependence edges between these operations to preserve their order. The order between the three sets of three moves, viz., three moves

to temporary registers, three moves to parameter registers and the three moves back from temporary registers have to be preserved though we can allow arbitrary mixing within each set. We draw PREBR[2] arcs from the first and second set of moves to the procedure call and POSTBR[3] edges from the *call* instruction to the third set. We also draw register dependency edges between the three sets as required by their register usages. The addition of these dependence edges ensures that the order between them is maintained.

4.3 PEPSE Implementation

Itanium architecture provides a non-blocking prefetch instruction *lfetch*. The syntax of the instruction [14] is

(qp) lfetch.lftype.lfhint [r_1]

The function of this instruction is to move the line containing the address specified by the value in register r_1 to the highest level of the data memory hierarchy. The lftype component decides whether to raise faults normally associated with a regular load for this prefetch instruction. This instruction has an immediate and base-update variants.

The objective of our scheme is to construct dependence graphs for delinquent loads and statically schedule speculative versions of them earlier in the program. ORC contains functionality to create and analyze the dependence graph at basic block, region and procedure levels. The kinds of dependencies that are of interest to us are CG_DEP_REGIN, CG_DEP_REGOUT and CG_DEP_REGANTI which represent

[2]A PREBR edge between an operation and a call instruction indicate that the operation has to be strictly executed before the ensuing procedure call.

[3]An POSTBR edge between a call instruction and an operation indicate that the operation has to be executed strictly after the procedure call.

register flow(true dependency), output and anti dependencies respectively, between the registers of the predecessor and successor instructions. The structure of a dependence edge between two OPs is shown in figure 4.3.

```
OP          *pred;          /* the predecessor */
OP          *succ;          /* the successor */
mINT16      latency;        /* latency in cycles from pred to succ */
mUINT8      omega;          /* iteration distance for loop-carried deps */
mUINT16     kind_def_opnd; /* kind is LOW 8 bits, definite is next bit,
dotted edge is the next bit, which tells if the edge is not always strict
and opnd is the HIGH 4 bits */
struct arc  *next[2];       /* next ARC in pred/succ list, respectively */
```

Figure 4.3: The structure of a dependence edge

For LDG creation, we first create the full dependence graph for the procedure using the ORC's functionality and then carve out LDGs for the delinquent loads. The set of predecessors for the operations are maintained in the compiler. The LDG is constructed by iterating the set of predecessors for each operation in the LDG. In the first iteration, the predecessors of the delinquent load instruction would be examined for inclusion in the LDG and these instructions in turn would be iterated to include the instructions on which they are dependent on. The boundary conditions are implemented as explained in section 3.2.

In our implementation, LDG creation and PEPSE scheduling are done in the same phase. The delinquency information is available to this phase from the profile run of the program. So, LDGs are created and PEPSE-scheduled only for the

delinquent loads. Before this phase starts, the delinquent loads are identified by sorting the loads according to the total stalls caused by them and then selecting the top 5% of them.

To create speculative versions of the LDG instructions, the scheduler needs to interact with the register allocator. The speculative version of the instructions are copies of original instructions with the source and destination registes changed so that they wouldnt affect the execution of the original program. Since our PEPSE implementation occurs before the register allocation stage, these new set of registers are easy to obtain. We just use temporary identifiers which will then be assigned to physical registers by the graph-coloring based register allocation algorithm. The insertion of LDGs only slightly increases the register pressure of the program since we limit the size of LDG to be 7 instructions. In case of delinquent loads in loops, the LDG typically contains only 1-3 instructions and it is quite easy to create a speculative version of them with very few extra registers.

Chapter 5

Evaluation Framework and Results

In this chapter we describe the results obtained from the implementation of the algorithms described in chapter 3 on the Open Research Compiler(ORC). In Section 5.1, we describe the evaluation framework used to implement the optimizations and in Section 5.2, we present the details of the results obtained.

5.1 Evaluation Framework

Our experimental platform is a 900MHz Itanium 2 server with four processors. Each processor has 4-way 16KB Level 1 split instruction and data caches, 8-way 256KB unified secondary cache and 12-way 1.5MB unified tertiary cache. The latencies for primary secondary and tertiary caches are 1,5-6,12-13 cycles respectively. The processor core has a 8-stage pipeline, can issue upto 6 instructions[1] at a time and incurs 6 cycle penalty on a branch misprediction. The Itanium architecture provides mechanisms, such as instruction templates, branch hints and cache hints to enable the compiler to communicate compile-time information to

[1]Each Long instruction in Itanium consists of three instructions. It is also called as an instruction bundle. Six simple instructions translate into two bundles in Itanium.

the hardware.

Every memory load and store instruction in the Itanium architecture has a 2-bit cache hint field in which the compiler is allowed to fill its prediction on the spatial and/or temporal locality of the memory area being accessed. By using the program's structural information, the compiler can fill out which cache level the load value is likely to be found and to which cache level it should be fetched. A processor based on the Itanium architecture can use this information to determine the placement of cache lines in the cache hierarchy to improve the memory utilization.

One important property of the Itanium processor that is worth noting is that it implements a stall-on-use policy: suppose a load instruction is issued at cycle time t_i, and the first use of the delivered data occurs at cycle time t_j, then the processor is not stalled unless the data is unavailable in the required register at time t_j. When this happens, the processor will be stalled for $L - (t_j - t_i)$ cycles where L is the latency of the memory hierarchy in which the data is found. During the stall, no further instructions can be issued until the data is available for the processor to continue.

Intel C/C++ compiler and Intel Fortran compilers produce the best results on the Itanium machine across all the SPEC benchmarks. But both of them are proprietary softwares and we do not have access to their source codes. Hence, we implemented a prototype of our optimizations on the Open Research Compiler(ORC)[6], version 1.1. ORC is an open source compiler for the Itanium Processor Family(IPF). It has the leading performance amongst the open source compilers for the IPF. ORC includes a comprehensive set of optimizations which include blocking(tiling), loop unrolling, software pipelining, if-conversion, data prefetching(based on Mowry et.al.[28]) and a global instruction scheduler integrated with a finite-state-automaton-based resource management. It provides common robust infrastructure and is modular, which enables quick prototyping of novel ideas. It provides

front-ends for C/C++, Fortran77 and Fortran 90.

Benchmark	Profile	Evaluate	Delinquents
101.tomcatv	train	ref	70
168.wupwise	train	ref	10
171.swim	train	ref	120
172.mgrid	train	ref	70
179.art	train	ref	100
183.equake	train	ref	60
189.lucas	train	ref	70
em3d	2000 2 50	30000 2 200	8
tsp	8000 0	8000000 0	10

Table 5.1: Our IPF benchmark suite and input workloads used for profiling and evaluation.

The benchmarks we use are a collection of 7 programs selected from the SPEC CFP suite (179.art and 183.equake are C programs, the others are implemented in Fortran)[7] and two kernels from the Olden pointer-intensive benchmarks. The reason for this selection is to demonstrate that our methodology is simultaneously applicable to both array and pointer based programs. We chose ORC's best optimization options[2] as the baseline, which implements prefetching based on Mowry's work[28]. The prefetch algorithm implemented in ORC is optimized for fortran programs but their prefetching support for C is poor because of the aliasing problem. We would like to demonstrate that our method works good when implemented on top of the native prefetching implemented in the ORC. All the selected benchmarks are comprised of various loops and contain most of the delinquent loads within *do-while* or *for* loops. Table 5.1 shows the set of benchmarks used, the

[2]ORC executed with -O3 option

inputs used for profiling and evaluation runs(with PEPSE-enabled code) and the number of delinquent loads used for our optimizations. Note that, in the case of SPEC benchmarks, we have used the *train* input workload for profiling run and *ref* input workload for reporting purposes.

We would expect the SPEC CFP benchmarks to exhibit significant ILP and that they are highly optimized by the ORC. But, the results in the next section show that there are still enough resources available that can be utilized by PEPSE. The PEPSE scheme has a few compile-time parameters which we consistently set to be as follows: The number of delinquent loads selected are the top 5% of the total loads present in the program rated according to total stall cycles, calculated as described in equation 3.1 from the profile information. The budget size for the LDG is set to be 7 instructions. The minimum branch frequency for a path-specific LDG to be constructed was set to be 20%.

Most of the recent processors have special performance counters to measure application characteristics. These counters exist as a small set of registers that count events, occurrences of specific signals related to the processor's function. Monitoring these events facilitates correlation between the structure of source/object code and the efficiency of the mapping of that code to the underlying architecture. This correlation has a variety of uses in performance analysis including hand tuning, compiler optimization, debugging, benchmarking, monitoring and performance modelling.

Performance Application Programming Interface(PAPI)[2] provides a simple, high level interface for the acquisition of simple measurements from the underlying counter hardware. As part of PAPI, there are a predefined set of events that represents the lowest common denominator of every good counter implementation. In this work, we use the PAPI software to verify the speedups obtained using the *time* command and also to check the overhead due to PEPSE.

5.2 Results

The results reported in this section were obtained by running the benchmarks compiled using the ORC compiler with and without our optimizations. The experiments for PEPSE are conducted in two stages. In the first stage, the assembly code is instrumented to be coupled to Dinero IV cache simulator and the profile information is obtained. To achieve this, the code generation stage of ORC was changed to add calls to dinero IV simulator just before the load instruction, with the load address and an identifier as parameters. In the second phase, the profile information generated in the first phase is used to identify the delinquent loads and enhance the application performance by applying PEPSE optimizations to those delinquent loads.

In Figure 5.1 we graph the normalized execution time of each of the benchmarks from our evaluation suite. For each benchmark, there are two bars showing the performances of PEPSE-enhanced and Load Sensitive Scheduling(LSS) enhanced programs. We have considered the performance of ORC with its maximum optimization option(-O3) as the baseline. We note that the baseline implements software pipelining and a data prefetching mechanism largely based on Mowry's doctoral thesis [21].

Load sensitive scheduling is a method by which the load latency information is made available to the scheduler through dependence edges between the load instruction and its use. The fundamental premise for this method is that the processor implement a stall-on-use policy. The scheduler equipped with the latency information tries to schedule more instructions in the region between the load and its use so as to nullify the effect of the long-latency load. We had implemented a simple case of load sensitive scheduling. Each time a new dependence arc is drawn, if the predecessor operation is a delinquent load, the arc is annotated with the average miss penalty of the load. This informs the scheduler of the memory

sensitive latency, and as such, leads to improved scheduling decisions.

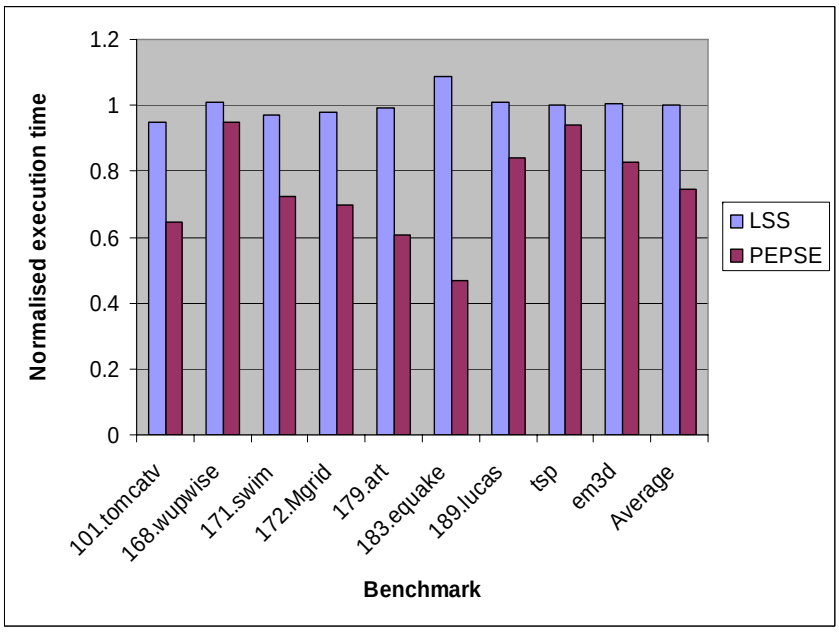

Figure 5.1: Normalized `user` CPU times for the Itanium 2 processor for PEPSE-enhanced and LSS-enhanced benchmarks relative to the baseline ORC optimizations.

In the graph of Figure 5.1 a value of 1 represents the execution time of the baseline. Any value less than 1 shows that the optimization has enhanced the performance and hence shortened the execution time. A value greater than 1 indicates that the optimization has degraded the performance of the application. According to

LDGs	101.tomcatv	LDGs	183.equake
	time (secs)		time (secs)
10	35.56	10	292
20	32.06	20	261
40	29.78	30	253
60	29.35	40	205
80	29.67	50	200
100	29.50	60	200

Table 5.2: CPU user time as a function of the number of embedded LDGs.

the results, the execution time of the PEPSE-enabled benchmarks reduced by 25% on an average. But our LSS implementation did not yield significant performance improvements and we attribute this observation to the lack of sufficient ILP in the benchmarks. The scheduler would then have to add nop instructions in those slots to make up for that latency, which ultimately degrades the performance.

As another experiment, we measured the running time of the applications by varying the number of delinquent loads selected for PEPSE. Table 5.2 shows the results for two benchmarks. The results of Table 5.2 show that the performance of benchmarks increase with increasing number of embedded LDGs. This is because of the higher coverage of delinquent loads. However, we cant increase the number of LDGs arbitrarily. Ultimately, we reach a stage at which, further increase in the number of LDGs increase the overhead without sufficiently reducing the processor stalls and hence the overall performance degrades. Our experiments show that we gain maximum performance when around the top 10% of the total loads in the program are considered for PEPSE processing.

In addition to the application running time, we use the PAPI toolkit [2] to monitor the performance counters in the Itanium and record the number of cycles elapsed

Benchmark	Time (in secs)			Cycles (in millions)		Speedup
	ORC	PEPSE	LSS	ORC	PEPSE	
101.tomcatv	45.6	29.3	43.3	39,069	26,329	1.56
168.wupwise	622	591	629	568,122	528,326	1.05
171.swim	317	230	307	305,825	219,447	1.38
172.mgrid	369	257	362	329,280	232,467	1.44
179.art	489	297	485	476,161	240,952	1.65
183.equake	471	220	512	422,920	199,857	2.14
189.lucas	366	307	369	332,537	283,951	1.19
tsp	80.7	75.9	80.7	69,347	67,866	1.06
em3d	6.39	5.29	6.41	5,643	4,726	1.21

Table 5.3: The `user` CPU time and total execution cycles for each benchmark.

between the start and end of the program. PAPI runs in the back ground and counts the cycles. The results are listed in Table 5.3. They are similar to the results obtained using the *time* command, shown in the second and third columns of the table.

In addition to the performance, one should also quantify the computation overhead due to the optimization. Towards this, we record the total number of dynamic instruction bundles issued before and after the prefetch orchestration using PEPSE. Each instruction bundle consists of a set of operations that are issued simultaneously and execute in parallel. The extent to which the prefetch orchestration lengthens the critical path is reflected in the number of instruction bundles that are processed. Counting the static size of the program would not reflect fully on the overhead. If the ILP is not very high in the program, adequate amount of resources may be available for PEPSE and hence the overhead would be very less. Table 5.4 quantifies the overhead due to orchestration of the PEPSE scheme to the

Benchmark	Time (in secs)			Instructions (in millions)		Overhead
	ORC	PEPSE	LSS	ORC	PEPSE	
101.tomcatv	45.6	29.3	43.3	63,599	69,510	1.09
168.wupwise	622	591	629	834,064	855,467	1.03
171.swim	317	230	307	510,672	557,737	1.09
172.mgrid	369	257	362	253,719	287,007	1.13
179.art	489	297	485	90,687	112,498	1.24
183.equake	471	220	512	253,719	287,007	1.13
189.lucas	366	307	369	467,338	471,439	1.01
tsp	80.7	75.9	80.7	75,699	80,673	1.07
em3d	6.39	5.29	6.41	967	1039	1.08

Table 5.4: The `user` CPU time and the dynamic number of operations for each benchmark.

benchmark suite. From the data on the table, the PEPSE implementation incurs a 3.66% increase in the number of dynamic instructions, on an average. Disregarding static schedule length of host regions and performing program embedded precomputation aggressively results in a 32% increase in the instruction count with detrimental effect on the performance. Hence it is prudent to narrow the scope of the optimizations to only severely delinquent loads in a program.

Chapter 6

Conclusions

This chapter concludes the thesis with a summary of the technique, its applicability and also gives directions for future research.

6.1 Summary of the thesis

In this thesis, we have addressed the ever-widening gap between the speed at which a processor processes data and the speed at which the memory sub-system supplies data to the processor with the introduction of PEPSE. We introduced the concept of Load Dependence Graph, which is a slice of the original program that calculates the address of a load instruction. First, we instrument the assembly code of the program to couple the program with Dinero IV uniprocessor cache simulator. This helps in identifying the delinquent loads in a program by creating the profile statistics of the program, consisting of a list of hits and misses due to the memory access instructions.

For all memory access instructions identified as delinquent by the profile run, we create the Load Dependence Graphs for those instructions. We then illustrate how

Program Embedded Precomputation using Speculative Execution(PEPSE) schedules a speculative version of these loads along with the program and ensures the timely availability of the loaded value so that the latency of the load is nullified or reduced. We also describe algorithms for generating the LDGs and embedding the corresponding address precomputation and data prefetch into the instruction stream of the application, compiled for the EPIC architecture. Then we introduced a technique by which the LDG can be applied to loops accessing an array structure(Induction Unrolling). We also propose a modification to the induction unrolling technique which would work well for pointer intensive applications.

We implemented a prototype of the proposed optimizations in the Open Research Compiler, an open source compiler for the Itanium Processor Family(IPF). This allowed us to study in detail the conditions affecting the effectiveness of the method, and using these studies, we formulated several variants of the algorithm and heuristics that will maximize the efficacy of PEPSE as a data prefetching mechanism. As a result, we have a data prefetching scheme that is (1) highly precise, (ii) efficient and robust in the context of a wide class of applications, (iii) does not require any new hardware support , and (iv) incurs very little overhead. Our implementation of PEPSE on ORC demonstrates that PEPSE is a viable optimization strategy and delivers upto 53% performance improvements compared to ORC optimizations, which includes its own native prefetching technology. We achieved a speedup of 25.6% on an average across nine benchmarks from the SPEC and olden suites. This serves as a concrete evidence that this method is effective.

6.2 Future Research Directions

Combining interprocedural analysis with PEPSE to enhance the PEPSE scheduling decisions using the information available from the inter-procedural analysis remains

an important topic of future research. Another orthogonal direction is to enhance the effectiveness of PEPSE in pointer-based code by disabling the LDG when it incurs misses itself. In pointer code, the result of the load is necessary to further propagate the precomputation. If a load within the LDG results in a cache miss that is not serviced in time for a subsequent LDG operation, the processor stalls and awaits data delivery. To alleviate this problem , we have devised a technique that will selectively disable the LDG instructions if any previous LDG instruction missed the cache. This would make sure that the LDG instructions themselves do not stall the processor. We plan to carry on this work further.

Bibliography

[1] Inside itanium II. http://www.extremetech.com/article2/0,3973,1160107,00.asp.

[2] Performance application programming interface.

[3] Andreas Moshovos Amir Roth and Gurindar S. Sohi. Dependence based prefetching for linked data structures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[4] J-L. Baer and T-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *International Conference on Supercomputing*, 1991.

[5] D. Callahan and A. Porterfield. Data cache performance of supercomputing applications. In *International Conference on supercomputing*, 1990.

[6] Open Research Compiler. http://ipf-orc.sourceforge.net/.

[7] Standard Performance Evaluation Corporation. http://www.spec.org/.

[8] K. Kennedy D. Callahan and A. Porterfield. Software prefetching. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[9] W. Jalby D. Gannon and K. Gallivan. Strategies for cache and local memory management by global program transformation. In *Journal of Parallel and Distributed Computing*, 1988.

[10] J. Elder and M. Hill. Dinero IV trace driven uniprocessor cache simulator.

[11] John Hennessy and David Patterson. *Computer Architecture, A quantitative approach*. Morgan Kaufmann, third edition, 2003.

[12] TI high performance DSPs. http://dspvillage.ti.com/docs/allproducttree.jhtml.

[13] M. D. Hill. *Aspects of cache memory and instruction buffer performance*. PhD thesis, Univ. of California, Berkeley, 1987.

[14] Intel Corporation. *Intel Itanium Architecture. Software Developers Manual. Instruction Set Reference*, October 2002.

[15] Dean M. Tullsen Christopher Hughes Yong-Fong Lee Dan Lavery Jamison D. Collins, Hong Wang and John P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *International Symposium on Computer Architecture*, 2001.

[16] Norman Jouppi. Improving direct mapped cache performance by the addition of small fully-associative cache and prefetch buffers. In *International Symposium on Computer Architecture*, 1990.

[17] Dongkeun Kim and Donald Yeung. Design and evaluation of compiler algorithms for pre-execution. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[18] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *International Symposium on Computer Architecture*, 1981.

[19] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[20] Steven R. Kumkel Mikko H. Lipasti, William J. Schmidt and Robert R. Roediger. Spaid: Software prefetching in pointer and call-intensive environments. In *International Conference on Microarchitecture*, 1995.

[21] Todd. C. Mowry. *Tolerating latency through software-controlled Data Prefetching.* PhD thesis, Stanford University, 1994.

[22] Edward S. Davidson Murali Annavaram, Jignesh M. Patel. Data prefetching by dependence graph precomputation. In *International Symposium on Computer Architecture*, 2001.

[23] G. Brown G. Desoli F. Homewood P. Faraboschi, J.Fisher. Lx: A technology platform for customizable vliw embedded processing. In *International Symposium on Computer Architecture*, 2000.

[24] Intel Itanium processors. http://www.intel.com/products/server/processors/server/itanium2

[25] A. Roth and G. Sohi. Speculative data-driven multithreading. In *International Symposium on High Performance Computer Architecture*, 2001.

[26] B. R. Rau Santosh G. Abraham, Rabin A. Sugumar and Rajiv Gupta. Predictability of load/store instruction latencies. In *International Conference on Microarchitecture*, 1993.

[27] Trimedia technologies and VLIW products. http://www.trimedia.com.

[28] Monica S. Lam Todd C. Mowry and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[29] Shinichiro ishizaki Toshihiro Ozawa, Yasunori Kimura. Cache miss heuristics and preloading techniques for general purpose programs. In *International Conference on Microarchitecture*, 1995.

[30] W.D. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in multiprocessor architecture". In *International Symposium on Computer Architecture*, 1989.

[31] M. E. Wolf and Monica. S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1991.

[32] M. J. Wolfe. More iteration space tiling. In *ACM/IEEE Conference on Supercomputing*, 1989.

[33] Youfeng Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *International Conference on Programming Languages Design and Implementation*, 2002.