Data Prefetching by Dependence Graph Precomputation

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

Abstract

Data cache misses reduce the performance of wide-issue processors by stalling the data supply to the processor. Prefetching data by predicting the miss address is one way to tolerate the cache miss latencies. But current applications with irregular access patterns make it difficult to accurately predict the address sufficiently early to mask large cache miss latencies. This paper explores an alternative to predicting prefetch addresses, namely precomputing them. The Dependence Graph Precomputation *scheme (DGP) introduced in this paper is a novel approach for dynamically identifying and precomputing the instructions that determine the addresses accessed by those load/store instructions marked as being responsible for most data cache misses. DGP's dependence graph generator efficiently generates the required dependence graphs at run time. A separate precomputation engine executes these graphs to generate the data addresses of the marked load/store instructions early enough for accurate prefetching. Our results show that 94% of the prefetches issued by DGP are useful, reducing the D-cache miss stall time by 47%. Thus DGP takes us about half way from an already highly tuned baseline system toward perfect D-cache performance. DGP improves the overall performance of a wide range of applications by 7% over tagged next line prefetching, by 13% over a baseline processor with no prefetching, and is within 15% of the perfect D-cache performance.*

1 Address Prediction vs. Precomputation

Out-of-order execution is the norm in current superscalar processor designs; it is intended to allow processors to tolerate pipeline stalls due to data dependences, 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. Furthermore, processor clock speeds are continuing to outpace memory access speeds, resulting in larger cache miss latencies (measured in processor cycles). Thus the large number of cache misses of today's dominant application domains, coupled with the increasing cache miss latencies of current processor designs cause significant performance degradation, even in aggressive out-of-order processors.

Prefetching data is one way to reduce the high miss rates of these irregular applications, but most existing data prefetch schemes are not effective in predicting prefetch addresses for irregular applications. Furthermore, previous work [11] has shown that for integer applications it is difficult to both accurately predict prefetch addresses and issue the prefetches in a timely fashion, i.e. sufficiently ahead of the corresponding demand reference to mask the large cache miss latencies.

In this paper we explore an alternative to predicting prefetch addresses for irregular access patterns, namely precomputing them. The *Dependence Graph Precomputation* scheme (DGP) introduced in this paper is a novel approach to prefetching. Once an instruction is fetched from the I-cache into the Instruction Fetch Queue (IFQ), its dependences are determined and stored as pointers with the instruction in the IFQ. When a load/store instruction that is deemed likely to cause a cache miss enters the IFQ, a Dependence Graph Generator (DGG) follows the dependence pointers still within the IFQ to generate the dependence graph of those instructions yet to be executed that will determine the address of the load/store instruction. The dependence graphs generated by DGG are fed to a separate Precomputation Engine (PE) that executes them to generate the load/store addresses early enough for timely prefetching. DGG does not remove any instructions from the IFQ; consequently all precomputed instructions will be executed in the normal manner by the main processor pipeline. Furthermore, the results generated by PE are used only for prefetching data; in particular they are not used for updating the architected state of the main processor. Since precomputation of the dependence graphs is speculative, it runs ahead of the normal computation by avoiding fetch queue and reorder buffer delays experienced by the normal computation

due to the in-order decode and the instruction scheduling priorities of the main processor pipeline.

Our results show that 98% of the precomputed addresses match the actual address accessed by the marked load/store instructions, indicating that prefetch addresses generated by DGP are highly accurate. DGP actually issued a prefetch for only 15% of these addresses, since 85% of the precomputed addresses were found to be already in the L1 cache. Of the prefetches issued by DGP, 94% are useful, reducing the D-cache miss stall time by 47%. Thus DGP takes us about half way from an already highly tuned baseline system toward perfect D-cache performance. DGP improves the overall performance of a wide range of applications by 7% over tagged next line prefetching, by 13% over a baseline processor with no prefetching, and is within 15% of the performance obtained with perfect D-cache.

The rest of this paper is organized as follows. Section 2 describes a hardware implementation for generating the dependence graphs of the load/store instructions that are marked as likely to miss. It then describes a mechanism to precompute the instructions in the dependence graphs and generate the prefetch addresses. Section 3 describes the simulation environment and performance analysis tools that we used to assess the effectiveness of DGP. Results of this assessment are presented in Section 4. Section 5 compares DGP with the most relevant prior work in the area of dependence graph generation; it also contrasts DGP with prior data prefetching techniques by highlighting the applicability and constraints of prior schemes that use either address prediction or precomputation for prefetching. Conclusions and some future directions for DGP are presented in Section 6.

2 Dependence Graph Generation and Precomputation

Figure 1 highlights (in bold capital font) the additional hardware added by DGP. For the simulation results of this paper, the set of load/stores that generated 90% of the D-cache misses in a profile run were marked for prefetching during the production execution of each benchmark. The dependence graph of a load/store instruction, I, in the Instruction Fetch Queue (IFQ) is the set of all unexecuted instructions, waiting either in the IFQ or in the Reorder Buffer (ROB), that contribute to the address accessed by I. In this paper we approximate the dependence graph by considering only the set of instructions that are in the IFQ at the time that the dependence graph of I is being generated. Note that the approximate definition ignores the dependence of I on unexecuted instructions waiting in the ROB. To build this approximate dependence graph for a marked load/store instruction, I, in the IFQ, we place I in the graph and recursively find and add to the graph, earlier instructions in the IFQ which define the register operands of instructions already in the graph. In the Alpha instruction set architecture used in this study, all instruction operands are either registers or immediate values. Since an immediate operand value is available in the instruction itself, these operands are not dependent on any other instruction, and hence are not considered while generating dependence graphs.

In our processor model the fetch stage of the processor pipeline fetches instructions from the I-cache and buffers them in the IFQ. The IFQ is a circular FIFO buffer with 2^N entries that store the fetched instructions in program order. We modified the IFQ entry format by adding three new fields, namely the *Index-Tag bit (IT)*, *OP1* and *OP2*. Each instruction, as it enters the IFQ, is assigned the next sequential $N+1$ bit index value (modulo 2^{N+1}). The lower N bits of the index determine the IFQ entry number where the instruction will be buffered; the most significant bit (MSB) of the index is stored in the IT bit of that entry. IT is thus toggled between 0 and 1 each time IFQ wraps around, e.g. the sequence of instructions in the 4th entry of a 16 entry IFQ will have indices that alternate between 00011 and 10011. The IT bit helps prevent stale or incorrect instructions from being included in dependence graphs (see Section 2.1).

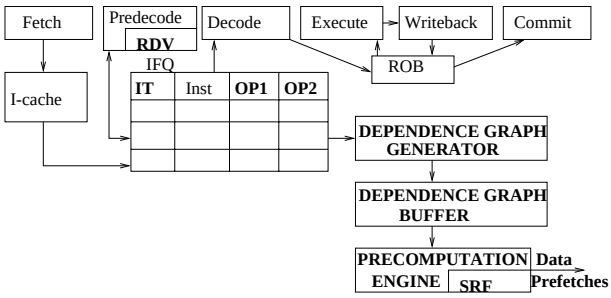

Figure 1. DGP hardware

The *OP1* field of the IFQ entry for instruction I stores the $N+1$ bit index of an earlier instruction in the IFQ that defines the value of input operand 1 of instruction I; if no earlier instruction still residing in the IFQ defines operand 1, or if I has no input operand 1, then *OP1* is set to NULL. *OP2* is defined similarly for operand 2. The Predecode stage of the processor pipeline is responsible for carrying out the dependence analysis. To do so, the predecoder is augmented with a new hardware structure, called the Register Definition Vector (RDV). RDV has one slot for each architected register of the processor. RDV slot j stores the $N+1$ bit index value of the instruction that most recently defined register R_j. When the predecoder decodes instruction I, stored in entry i of the IFQ, it accesses the RDV slots corresponding to the input operand fields of I and sets the *OP1* and *OP2* fields of ith IFQ entry to the corresponding indices read from RDV. The predecoder then considers the result register, R_d, of instruction I and sets RDV slot d to the $N+1$ bit index of instruction I (namely i or $2^N + i$

according to the IT bit of instruction I).

DGP is not concerned with the remaining stages of the pipeline or the ROB shown in Figure 1. They operate as in a traditional out-of-order superscalar processor and hence are not described here.

2.1 What to Precompute

The Dependence Graph Generator (DGG) scans the predecoded instructions in the IFQ in program order. Whenever it finds a load/store instruction, L, that is marked for prefetching, it initiates the process of building a dependence graph for L. DGG begins this process by storing the marked load/store instruction, L, along with its *IT*, *OP1*, and *OP2* fields from the IFQ in a new dependence graph.

Then for each instruction, I, added to the dependence graph, DGG follows the pointer in its *OP1* field, if any, to an earlier instruction , E, in the IFQ that defines the first input operand of I. It is possible, however, that E has left the IFQ by this time. In this case the IFQ entry vacated by E may be empty, or it may contain a new instruction due the circular FIFO ordering of IFQ. If a new instruction, N, occupies the IFQ entry then the *OP1* field of I points to N which does not define the input operand of I; furthermore N is to be executed after I in the program order. DGG therefore adds an instruction to the dependence graph only if *OP1* is not NULL, the IFQ entry it points to is not empty, the register defined by the instruction pointed to by OP1 matches the operand 1 register of I, and the IT bit in the entry pointed to by *OP1* matches the MSB of the *OP1* index. The $OP2$ field of instruction I is then used similarly to determine the instruction, if any, in the IFQ that defines the second input operand of I.

Note that this scheme does not always prevent a newer instruction from being included in the dependence graph. If the RDV slot used to define *OP1* has a stale value, i.e. the IFQ entry it points to has been overwritten by a newer instruction since the RDV slot was last defined. Often in this case the IT bit will not match. However, if the IFQ entry that the RDV slot points to has been overwritten an even number of times since the time the RDV slot was last defined, then the IT bit will match the MSB of the OP field. But even in this case, given 32 registers, it is possible but unlikely that the result register of the new instruction in the IFQ entry will match the input operand register of I. In our precomputation method (see Section 2.2), precomputed instructions from the dependence graphs never modify the architected state of the main processor. Thus although incorrect instructions in the dependence graphs, if any, will almost certainly result in generating a wrong prefetch address, they do not corrupt the architected state of the machine; they simply reduce the DGP performance gain.

DGG builds the dependence graph by repeating the above operations until it finds no new (unanalyzed) instructions in the dependence graph, or the Dependence Graph Buffer (DGB) is full. If the DGB is full, the DGG squashes the dependence graph that it is currently generating, as it is too large to store in DGB and would also be relatively unlikely to result in a timely prefetch due to its size. Otherwise, the graph for L is deemed complete and is stored in DGB. Each entry in DGB holds one dependence graph; the order of the instructions in the entry is the order in which DGG added them to the graph.

DGG then resumes searching the IFQ for the next marked load/store instruction. It is possible that a new marked load/store instruction, M, enters the IFQ while DGG is still in the process of building the dependence graph of an earlier load/store instruction, L. The graph generation for M is delayed until DGG completes building the dependence graph for L. In our evaluations we assume a simple implementation of DGG that takes one cycle for each instruction added to the dependence graph.

As shown in Section 4.2, 92% of the dependence graphs have fewer than 8 instructions, and the largest has only 15 instructions. In our evaluations, the IFQ holds up to 32 (or 64) instructions, the DGB has 2 entries, and the size of each DGB entry is 64 bytes (16 instructions). No dependence graph encountered in our benchmark evaluations exceeded this size. Of the two DGB entries, one is used by DGG to store the dependence graph that it is generating; the other holds a completed dependence graph that is ready for execution or is being executed by the Precomputation Engine.

Since the dependence graph of one load/store instruction may use the register value loaded by another load instruction, dependence graphs may contain intermediate load instructions. The dependence graph generator follows the $OP1$ and $OP2$ pointers of an intermediate load instruction, IL, as usual to identify the earlier instructions in the IFQ that define the address accessed by IL. During the precomputation IL, the processor's cache is accessed using the precomputed address to get IL's operand value. However, if there is an uncommitted store to the same address which precedes IL in program order, this hazard cannot be detected by DGG since the operand addresses of load/store instructions in the IFQ are unknown. Such hazards may cause incorrect addresses to be generated for marked load/store instructions. Nevertheless, as shown in Section 4, 94% of the prefetches issued by DGP are useful. Thus such hazards are not a significant hindrance to generating accurate prefetch addresses using DGP.

2.2 How to Precompute

A separate Precomputation Engine (PE) is used to execute the dependence graphs stored in the Dependence Graph Buffer. Since DGG generates the dependence graphs by tracking dependences backwards from a marked load/store in the IFQ, PE executes the dependence graph by beginning with the last instruction added to the graph and continuing

forward until it executes the marked load/store instruction at the root of the graph.

PE executes instructions speculatively. The results generated by PE are used only for prefetching data, and in particular they never update the architected state of the main processor. When executing a marked instruction, PE generates the load/store address and prefetches the data into the L1 cache without loading a value into the main processor's register (for a load) and without storing a value in the prefetched address (for a store). Since PE is prevented from modifying the architected state of the main processor, it can speculatively run ahead of the actual computation without having to guarantee the correctness of the precomputation. Note that DGG does not remove any instructions from the IFQ; consequently all precomputed instructions will be executed in the normal manner by the main processor pipeline.

The PE is a simple single issue, in-order execution engine that executes one integer instruction every cycle (except that multiply takes 2 cycles, and integer divide takes 4). As soon as it finishes one graph it proceeds to the next buffered graph. Since address computation arithmetic does not involve any floating point operations, and DGG (since it operates on instructions in the IFQ) follows only the predicted control flow path from the branch predictor while building the dependence graphs, the dependence graphs never contain floating point operations or branch instructions. Hence PE needs no floating point or branch execution units or branch prediction hardware. The precomputation engine has a scratch register file (SRF) to store the live results of precomputed instructions. The maximum number of SRF registers needed is equal to the maximum width of the dependence graph, which never exceeded 4 instructions in our benchmarks. Since PE executes at most one instruction every cycle, the SRF needs only two read ports and one write port. PE is thus a simple fixed-point engine that can be implemented with a modest transistor count.

The SRF registers are all set to unassigned before beginning a new graph. Before PE executes an instruction, I, an SRF register is assigned to store its result. To get the necessary operand values for I, PE accesses the SRF only if the corresponding *OP* field of I is not NULL, in which case an already executed instruction in the dependence graph would have generated the operand value and stored it in the SRF. When the *OP* field is NULL, the PE obtains the corresponding operand value by accessing the processor's register file and the ROB for forwarding uncommitted register values. Hence in our implementation, the register file and the ROB of the main processor each need two additional read ports for PE accesses.

Due to the lack of interlocks between PE and the main processor, operand values read by PE from the main processor's register file and ROB could be incorrect. At the time PE accesses a main processor register, the desired value might not yet be defined (because the instruction that defines it has not yet executed), or might have already been overwritten by a later instruction if the main processor has run ahead of the PE. Accessing the ROB as well as the register shifts (and generally reduces) this problem, but does not eliminate it. Thus we note that register hazards may arise due to the lack of interlocks between the main processor pipeline and PE. However, we must be careful not to delay the operation of the main processor pipeline. Furthermore we note that any hazards that do manifest themselves will simply reduce DGP performance, and our experimental results indicate that DGP is quite robust in practice.

The PE accesses the processor's cache to get the values for intermediate load instructions, if any, in the dependence graph; if such an access results in a miss, the PE squashes the remaining precomputation; it does not issue a prefetch, and moves on to begin precomputing the next dependence graph in the dependence graph buffer, or await its arrival. After the last (load/store) instruction in a dependence graph is executed, the address generated is used to prefetch the data into the processor's L1 cache. The prefetch is squashed by L1 if the data is already present.

2.3 Graph Generation and Precomputation Costs

Since DGG takes one cycle to generate each instruction in the dependence graph, one way to reduce this cost is to cache recently generated dependence graphs so as to avoid regenerating a graph every time its root load/store instruction is encountered. Caching would save the time needed to regenerate dependence graphs and might thereby increase prefetch timeliness. One simple caching strategy is to store the most recently generated dependence graph of each marked load/store instruction in a dependence graph cache. DGG could then access the graph cache using the PC of a marked load/store instruction as an index. On a hit, the graph is sent to the dependence graph buffer to await precomputation. But on a graph cache miss, DGG must build the dependence graph as before. Whenever a new dependence graph is built, it is sent to the buffer to await precomputation and is also stored in the dependence graph cache, replacing some previously stored entry. We implemented this simple graph caching scheme, but our results (not presented here) show that it actually degrades the performance. Caching dependence graphs gains better timeliness at the cost of using a cached graph that may be inaccurate because the predicted control path that leads up to the current instance of the marked load/store may be different from the path used to generate the cached graph. The performance loss due to the reduced accuracy of cached graphs outweighed the performance gains due to the more timely prefetching.

In our implementation we did not explore the possibility of reducing the instruction overhead due to precomputation. One way to reduce this overhead is to let PE up-

date the main processor's architected state after executing the instructions in the dependence graphs, thereby eliminating the need to re-execute them in the main processor. In our implementation, instructions executed by PE never update any architected state of the main processor. We chose this approach because it avoids complicating the main processor by almost entirely decoupling the DGG and PE from the main processor pipeline execution. Constraining the PE to update the architected state would significantly increase the control logic and add performance degrading interlocks between the PE and the main processor pipeline to ensure correct updating of the processor state. This would also increase the complexity of the testing and verification process of the design. Moreover the increased communication between the PE and main processor would add new wire delays which may reduce the clock speed.

Thus in our implementation we chose the simplicity of a free running PE without interlocks, despite the often redundant reexecution of precomputed instructions in the main processor. Our evaluations show that on average PE executes about 20% of the main processor's instructions. Furthermore as 98% of the precomputed addresses generated by PE match the addresses generated by the main processor, we infer that the vast majority of the instructions executed by the PE are in fact eventually reexecuted by the main processor without change, as otherwise the addresses would be highly unlikely to match. For future processors with further increased transistor counts [9], the 20% reexecution overhead will be even less of a concern relative to the savings in cache miss stall overhead achieved without slowing down the main processor pipeline [1].

The additional hardware needed for implementing DGP is the Dependence Graph Generator, Dependence Graph Buffer, and the Precomputation Engine, in addition to augmenting the IFQ and adding the RDV to the predecode stage. We assumed a simple Dependence Graph Generator that reads one entry from the IFQ every cycle. The Dependence Graph Buffer uses only 128 bytes of storage. The Precomputation Engine has one general purpose integer functional unit and a small scratch register file with two read ports and one write port. Even with such simple hardware components, DGP reduces the data cache miss stalls quite significantly. Thus we believe that the extra computation and the transistor budget required for DGP is well justified.

3 Simulation Environment and Benchmarks

To evaluate DGP performance we selected seven benchmarks from the CPU2000 integer benchmark suite (*gzip, gcc, crafty, parser, gap, bzip2* and *twolf*) and two sets of database queries generated from eight queries (1 through 7 and 9) from the Wisconsin benchmark [2] and five queries (1,2,3,5, and 6) from the TPC-H benchmark [7]. These benchmarks were selected because they exhibit poor data cache behavior and our existing simulation infrastructure allows us to run them without modifying the benchmark source codes. The *wisc* workload consists of all the eight Wisconsin queries running on a 10MB database; *wisc+tpch* is all eight Wisconsin queries plus the five TPC-H queries running concurrently on a database of size 40MB. The database queries are implemented on top of SHORE [4], a fully functional storage manager which has been used extensively in the database research community and is also used in some commercial database systems.

All the benchmarks were compiled on an Alpha 21264 processor running OSF Version 4.0F using the Compaq C++ compiler, version 6.2, with -O5 -ifo -inline and speed optimization flags turned on. To the O5 optimized binary, we applied the OM [12] tool which reduces I-cache misses by performing feedback-directed code layout. Although OM is targeted toward improving the spatial locality of the code, it's ability to analyze object level code at link time also opens up new opportunities for redoing some traditional compiler optimizations, such as inter-procedural dead code elimination and loop-invariant code motion, which even the O5 optimizations could not perform effectively at compile time. Thus, in addition to reducing I-cache misses, OM also reduces the number of instructions executed in a program. Although the CPU2000 benchmarks do not benefit much from the OM optimizations, OM does provide an 11% performance improvement over O5 code for the database benchmarks. We used this highly optimized binary as a baseline to study the performance impact of DGP.

The CPU2000 benchmarks were first run on the *test* input set provided by SPEC. The set of load/store instructions were sorted according to the number of misses they generated, then those that missed most often were marked in the code down to the level where 90% of the cache misses were covered. Similarly, a set of three queries from the Wisconsin benchmark: query 1 (sequential scan), query 5 (non-clustered index select) and query 9 (two-way join) were run on a small database of 2100 tuples, and the load/store instructions that caused 90% of the cache misses were marked in the *wisc* and *wisc+tpch* workloads. Recall that during actual execution of the full workload on the large datasets, DGG will generate dependence graphs only for the marked load/store instructions.

The CPU2000 benchmarks were then run on the *train* input set, *wisc* was run on a 10MB database, and *wisc+tpch* on a 40MB database. The results presented in this paper were generated by terminating the execution of each benchmark after 2 billion committed instructions.

The SimpleScalar out-of-order processor simulator [3] was modified by adding a predecode stage between the fetch and dispatch stages of the processor pipeline. We also added the DGG and the PE that execute concurrently with the main

processor simulator. This modified simulator was used for detailed cycle-level processor simulation. The microarchitecture parameters were set as shown in Table 1.

4 Simulation Results

In this section we first present the results from our initial feasibility studies. We measure the instruction delays in the processor pipeline of the target microarchitecture (Table 1) and show that instructions spend 23 cycles on average in the IFQ and ROB before entering the execution stage. Our measurements also show that most marked load/store instructions have dependence graphs of fewer than 8 instructions. The 23 cycle delay between the fetch and the beginning of execution, coupled with the small dependence graph sizes encouraged us to consider prefetching by dependence graph precomputation. We then show that DGP did in fact reduce the D-cache miss stall time by 47%, resulting in a 13% average overall performance improvement.

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

Table 1. Microarchitecture Parameter Values

4.1 Delays in the IFQ and Reservation Stations

In our processor model the fetch stage of the pipeline can fetch 4 instructions every cycle and the processor can commit 4 instructions every cycle. But the *IPC* column of Table 2 shows that on average only 1.4 instructions are actually committed per clock cycle by the baseline system without prefetching. (The average figures in Tables 2-4 and Figure 2 are computed as if all 9 benchmarks were combined into one long run, and dividing by 9 for the counts in Tables 3 and 4.) IPC is much less than the maximum fetch rate because of pipeline stalls due primarily to branch mispredictions, cache misses, and resource conflicts. Because of this disparity between the instruction fetch and completion rate, the IFQ will fill quickly and tend to remain near full as can be seen from the *IFQ_full%* column. IFQ is full on average for 69% of the execution time. In fact, a careful analysis of the IFQ occupancy showed that the only time IFQ is not full is immediately after a branch misprediction, during which time the fetch entries are all squashed and instructions from the correct path are being fetched. *Wisc* and *wisc+tpch* suffer from much higher branch misprediction rates than the CPU2000 benchmarks, resulting in their IFQs being full only 39% and 44% of the time, respectively.

An instruction in the processor pipeline spends sometime (*IFQ_Delay*) in the IFQ waiting for both an available decoder and an empty reservation station before it can leave the IFQ; Table 2 shows a 13 cycle IFQ wait for the average instruction. After leaving the IFQ, instructions wait in a reservation station for dependence resolution and functional unit availability (*Exec_Delay*) before beginning execution, this takes the average instruction 10 additional cycles. Thus an average instruction spends 23 cycles in the pipeline before entering the execution stage. After beginning execution, it spends another 8 cycles before it is committed.

Assuming the average 23 cycle delay before beginning execution, to completely mask a 16 cycle L1 D-cache miss latency, a prefetch for the corresponding load/store instruction would have to be issued within 7 cycles after that instruction enters the IFQ. Thus the dependence graph generation plus the precomputation of the address, which both depend on the size of the dependence graph, would have to be completed within 7 cycles from the time the marked load/store instruction enters the IFQ. Of course individual cases will vary from this average and an out-of-order processor should be capable of masking some portion of the miss latency itself. Furthermore, although masking the full miss latency is most desirable, masking a portion of the latency is still helpful.

			Pipeline Delay (cycles)		
	IPC	IFQ_full%	IFQ	Exec	Commit
gcc	1.5	72%	11	7	7
parser	1.0	92%	21	18	12
twolf	1.5	86%	12	12	6
gap	1.3	87%	14	11	7
crafty	2.1	78%	9	8	7
bzip2	2.4	99%	13	10	11
gzip	2.0	91%	11	11	8
wisc	1.1	39%	13	4	8
wisc+tpch	1.0	44%	15	6	10
Avg	1.4	69%	13	10	8

Table 2. Instruction Flow and IFQ occupancy (baseline system)

4.2 Dependence Graph Size

Figure 2 shows the cumulative distribution of dependence graph sizes for each benchmark and the average. On average 92% of the dependence graphs have 8 instructions or fewer. Since DGG takes one cycle to generate each instruction in the dependence graph, 92% of the dependence graphs can be generated within 8 cycles. Likewise PE can execute these dependence graphs and generate the prefetch address in at most 8 cycles. Thus a prefetch can almost always be issued within 16 cycles after a marked load/store enters the IFQ. Since the average instruction spends 23 cycles before entering the execution stage we estimate that at least 7 cycles of the D-cache miss latency would be masked

by these prefetches. Thus a 16 cycle miss stall might be reduced to 9 cycles, a 44% reduction for an 8 instruction graph. Moreover, since 56% of all the dependence graphs have 3 or fewer instructions, 56% of the prefetches can be generated within 6 cycles, leaving 17 cycles to complete the prefetch. Thus over half the prefetches should completely mask the 16 cycle cache miss latency. This observation encouraged us to explore precomputation as a viable technique for prefetching.

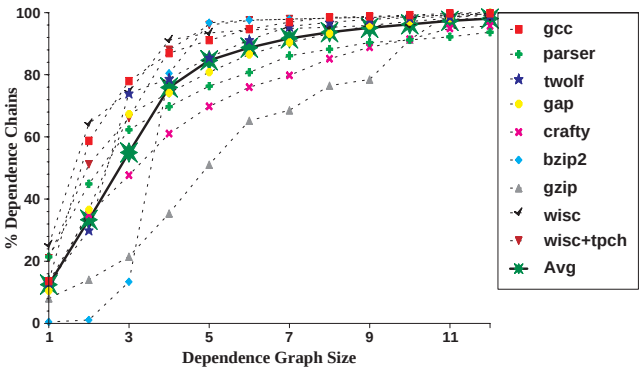

Figure 2. Cumulative Distribution of Dependence Graph Sizes (in instructions)

4.3 Marked Load/Store Characteristics

Table 3 shows the basic characteristics of the load/stores that generated 90% of the cache misses in the profile run, and were marked for precomputation. Out of 57,493 load/stores in the average benchmark, only 166 were marked for precomputation. Thus only a very few load/stores in a program cause the vast majority of D-cache misses. Precomputation can therefore focus only on this small set of load/stores, which in turn greatly reduces the dependence graph generation and precomputation overhead.

	Tot_L/S	Mark_L/S	Mark_Ref%	Over%
gcc	117880	294(0.24%)	13.5%	13.1%
parser	19276	258(1.33%)	25.8%	34.2%
twolf	31360	158(0.50%)	23.5%	24.7%
gap	51668	108(0.20%)	17.4%	23.0%
crafty	19461	153(0.78%)	17.5%	22.5%
bzip2	9382	59(0.62%)	46.2%	45.3%
gzip	10261	45(0.43%)	4.6%	6.8%
wisc	127157	152(0.11%)	4.4%	3.9 %
wisc+tpch	130989	268(0.20%)	10.0%	8.9%
Avg	57493	166(0.28%)	10.8%	19.9%

Table 3. Marked Load/Store Characteristics

The total references generated by the marked load/stores (*Mark_Ref%*) on average account for 11% of all the references during the actual execution. The average instruction overhead (*Over%*) of 20% shows that on average, 1 instruction is executed by PE per 5 instructions executed by the main processor. In *bzip2* the marked load/stores from the profile run accounted for 46% of the total references generated in the actual execution, and 45% of all its instructions were also precomputed by PE. Since DGP generates a dependence graph for every dynamic instance of a marked load/store execution, *bzip2* has significantly higher instruction overhead than the other benchmarks.

4.4 Precomputation Characteristics

Table 4 shows the characteristics of precomputed addresses and the resulting prefetches that were issued by the PE. Of the 110,008,552 precomputed addresses (*Prec*) of the marked load/stores in the average benchmark, 98% matched the address generated by the main processor, indicating that PE is highly accurate in generating prefetch addresses. Thus neither the potential register hazards due to the lack of interlocks between the main processor pipeline and the PE, nor the inability of DGP to detect memory aliases, significantly hinder DGP from generating accurate prefetch addresses. Recall that a prefetch is issued to L2 only if the precomputed address is not already in L1; *Pref_Iss* shows that 19% of the total precomputed addresses resulted in a prefetch being issued. *Pref_Used* indicates that on average 18% of the total precomputations (94% of the issued prefetches) resulted in a useful prefetch, i.e. the prefetched line was used by the main processor before being replaced.

	Prec	Match_Prec	Pref_Iss	Pref_Used
gcc	102.8	101.7	34.2	33.6
parser	140.0	133.0	29.3	27.8
twolf	130.7	127.8	49.3	46.1
gap	121.3	119.6	20.0	19.3
crafty	82.2	80.7	14.7	10.9
bzip2	277.1	276.7	17.4	17.3
gzip	23.1	22.3	2.8	2.2
wisc	33.9	33.6	5.5	5.3
wisc+tpch	79.0	78.1	19.7	19.3
Avg	110.0	108.2(98%)	21.4(19%)	20.2(18%)

Table 4. Precomputations and Prefetches (in millions)

4.5 Performance Improvement with DGP

Figure 3 shows the reduction in execution cycles due to DGP. There are six bars for each benchmark. The leftmost bar shows the base performance with a data cache size of 32KB. The second bar shows performance with tagged next line (NL) prefetching, where the main processor issues a prefetch to the next cache line on every cache miss and also on a first hit to a previously prefetched cache line. The performance of DGP with and without NL, using an IFQ size of 32, is shown in the third and fourth bars, respectively. The fifth shows the execution cycles that would be required with a perfect data cache, wherein every access to the data

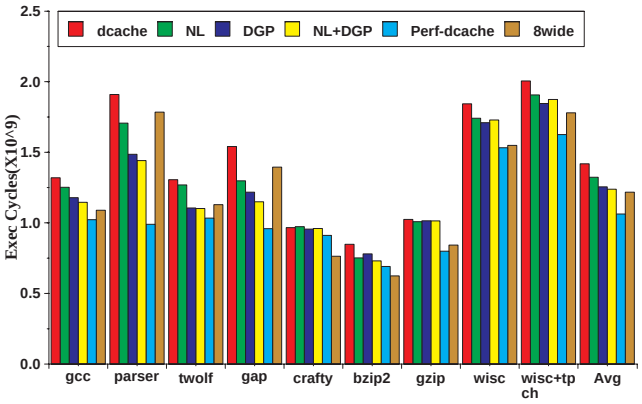

Figure 3. Performance with a 32 entry IFQ

cache is completed in 1 cycle. The rightmost bar shows the performance when the baseline processor's issue width and general purpose functional units are doubled (from 4 to 8), with other parameters remaining as in Table 1.

These results show that NL prefetching improves the performance by 7%. When DGP is used with or without NL it reduces the average cache miss stall time by 47%. This stall time reduction results in a 7% further improvement over NL, and 13% over the baseline processor with no prefetching. Furthermore doubling the issue width and functional units of the main processor gives only a 2% performance improvement over DGP. The extra transistors required to implement DGP is approximately equivalent to adding only one additional issue width and one functional unit to the 4-wide baseline processor. DGP is also within 15% of the perfect data cache performance.

4.6 Prefetch Effectiveness and Bus Traffic

Figure 4 shows the prefetch effectiveness of DGP without NL by categorizing the issued prefetches into three categories. The bottom component, *Pref Hits*, shows the number of times that the next reference to a prefetched line found the referenced instruction already in the L1 cache. The center component, *Delayed Hits*, shows the number of times that the next reference to a prefetched line found the referenced instruction still *en-route* to the L1 cache from lower levels of memory. Finally the upper component, *Useless* prefetches, shows the number of cache lines that were prefetched, but replaced before their next reference. On average, for IFQ size 32 (left bars) only 6% of the prefeches issued by DGP are useless. Thus the extra traffic generated due to DGP is minimal. Of the 94% useful prefetches (Pref Hits+Delayed Hits), 40% are Delayed Hits. Further analysis of the delayed hits showed that 35% of them incurred less than 8 cycles of miss penalty. Another 35% incurred more than 16 cycles of miss penalty because the prefetched line was also missing in the L2 cache. Since DGP is most effective in covering L1 miss penalties, it could benefit from techniques that improve L2 cache performance. However, note that 57% of the prefetches fully masked the corresponding miss penalty which is consistent with the ballpark estimate in Section 4.2.

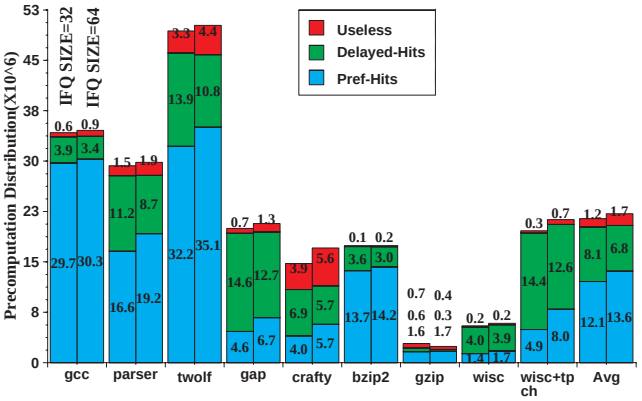

Figure 4. Distribution of DGP prefetches issued with IFQs of 32 and 64 entries

4.7 Increasing Timeliness with a Larger IFQ

One possible way to increase prefetch timeliness is to increase the IFQ size, allowing instructions to be buffered in the IFQ for a longer time, and thereby giving more time for the precomputations to complete. However, to fill a longer IFQ the fetch stage has to run further ahead of the execution stage, which it does by predicting several more branches ahead, which in turn decreases the branch prediction accuracy. Decreased branch prediction accuracy may negate the gains due to increased prefetch timeliness. Moreover increasing the IFQ size may also increase dependence graph sizes, thereby increasing the time required to generate and precompute the dependence graphs. To evaluate these tradeoffs we simulated IFQ sizes of 64, 128, 256 and found that the queue size has a negligible effect on performance; results for IFQ size 64 are shown in Figure 5.

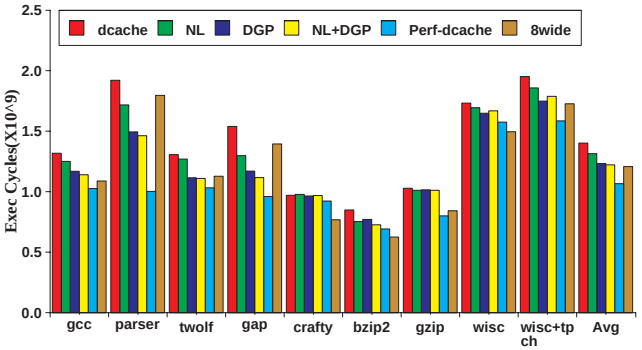

Figure 5. Performance with a 64 entry IFQ

The right bars in Figure 4 show the distribution of precomputations for an IFQ of size 64. As can be seen the useful precomputations increase and delayed precomputations

decrease when the fetch queue size is increased from 32 to 64. However, the number of useless precomputations also increases because of the increase in branch mispredictions.

Although Figure 4 shows that a longer IFQ allows more of the precomputations to be useful, Figure 5 shows that the actual performance of DGP with a 64 entry IFQ, due to a 2% reduction in branch prediction accuracy which negates the gains from better prefetch timeliness, is nearly unchanged from the 32 entry IFQ (third bar in Figure 3). However, the two database benchmarks, *wisc* and *wisc+tpch*, despite having lower branch prediction accuracy than the CPU-2000 benchmarks, do show a modest net gain with the larger IFQ.

5 Comparison of DGP with Previous Work

Program slicing [13] is a more general technique used for focusing on parts of a program that have a direct or indirect effect on the values computed at some points of interest. The concept of a dependence graph, used in this paper, is one way of generating a program slice. Dependence graphs are used extensively by compilers for performing several traditional compiler optimizations and by debuggers in order to focus in on instructions that cause a program bug. Typically these dependence graphs are generated at compile time, but their size is too large to be useful for precomputation. DGP uses hardware to generate its dependence graphs dynamically, which significantly reduces the graph size and makes precomputation quite feasible, as the domain of interest includes only the predicted control path and extends back only as far as the unexecuted instructions that still reside in the IFQ.

Zilles and Sohi [14] studied the characteristics of dependence graphs called "backward slices" for performance degrading instructions such as cache miss generating instructions and hard to predict branches. Their results show that in a window of 512 instructions leading up to a performance degrading instruction, fewer than 10% of the instructions contribute to the data used by the performance degrading instruction. Roth and Sohi [10] proposed speculative data driven multithreading (DDMT) in which backward slices are first generated from a program trace and are then preexecuted on a separate thread in a simultaneous multithreaded (SMT) processor. Speculative threads are spawned when a designated *trigger* instruction from the non-speculative thread starts execution. In their scheme, instructions in the speculative threads are not reexecuted by the main thread and hence speculative threads do update the processor's architected state. To guarantee correct updating of the architected state by the speculative thread, their approach requires complex instruction issue and register renaming logic in the processor core

Collins, et al. [6] use the DDMT approach to generate backward slices for cache miss causing loads from program traces. These slices are preexecuted on a separate thread in an Itanium processor for early generation of load addresses. However, as in DGP, the speculative slices are used only for prefetching data, and in particular do not update the architected state of the processor. This approach has the advantage that it avoids complicating the processor core by almost entirely decoupling speculative thread execution from main thread execution. Collins proposes a *chaining trigger* mechanism whereby speculative threads are also allowed to spawn other speculative threads. This trigger mechanism differs from Roth's approach wherein only non-speculative threads are allowed to spawn speculative threads. While all the approaches above use trace driven analysis to generate their dependence graphs, in this paper we have presented an easily implementable technique to generate these graphs dynamically at runtime, together with a mechanism to precompute the dependence graphs and generate accurate and timely prefetches.

Farcy, et al. [8] use precomputation to generate branch outcomes earlier. To precompute the branch outcome of a loop branch they skip several iterations of the loop and get ahead of the normal execution to determine future branch outcomes of the loop branch. This technique relies on the regularity of the program to gain the required distance for precomputation. But in the case of a cache miss, the miss address needs to be generated further ahead of the demand in order to completely mask today's increasingly large cache miss latencies. The technique presented in [8] is not suitable for masking large L1 cache miss latencies for irregular applications.

6 Conclusions and Future Work

Today's dominant application domains exhibit irregular access patterns resulting in a significant number of cache misses. Furthermore, processor clock speeds are continuing to outpace memory access speeds, resulting in longer cache miss latencies. Thus even aggressive out-of-order processors suffer significant performance degradation when executing these applications due to their frequent cache misses and increasingly long cache miss latencies. Existing data prefetch techniques, do not accurately predict the prefetch addresses for irregular applications in a timely fashion. In this paper we have explored an alternative to predicting prefetch addresses, namely precomputing them.

Dependence Graph Precomputation (DGP) is a novel approach for accurately and dynamically identifying and precomputing the instructions that determine the addresses accessed by those load/store instructions deemed to be responsible for most data cache misses. The Dependence Graph Generator described in this paper efficiently generates the required dependence graphs at run time. A separate Precomputation Engine executes these dependence graphs to generate the data addresses of the designated load/store instructions early enough for timely prefetching. The re-

sults generated by PE are used only for prefetching data; in particular they do not update the architected state of the main processor. Since PE is prevented from modifying the architected state of the main processor, it can speculatively run ahead of the actual computation without having to guarantee the correctness of the precomputation; nevertheless, it is nearly always correct.

We have used a very aggressive out-of-order processor with well tuned application codes as a baseline system for studying the performance impact of DGP. Even with this formidable baseline, our results show that 94% of the prefetches issued by DGP are useful, reducing the D-cache miss stall time by 47%. Thus DGP takes us about half way from an already highly tuned baseline system toward perfect D-cache performance. DGP improves the overall performance of a wide range of applications by 7% over tagged next line prefetching, by 13% over a baseline processor with no prefetching, and is within 15% of the perfect D-cache performance.

The prefetch timeliness of DGP can be improved if the IFQ size is increased. But increasing IFQ size decreases branch prediction accuracy, thereby reducing the net performance. Hence branch prediction accuracy will eventually limit the size of the IFQ that should be used for the purpose of generating the dependence graphs. Although this paper focuses on load/store address precomputation, this technique could also be used for early generation of branch outcomes, thus enabling it to ameliorate both the instruction and data supply problems of current processors. Another alternative to implementing the precomputation engine is to execute the dependence graphs on special threads in a simultaneous multithreaded processor, as suggested by Chappell, et al. [5]. We are currently considering this alternative to implementing the precomputation engine.

The insights gained from this work can also be used to change the scheduling priority among the reservation stations. By scheduling and executing the instructions in the dependence graphs of marked load/store instructions ahead of other ready instructions, results similar to those shown in this paper might be achieved without a special precomputation engine. One of the main advantages of out-of-order processors is their ability to tolerate cache miss latencies by executing other ready instructions in the reservation stations. If a simple in-order main processor were used with several autonomous precomputation engines to simultaneously reduce cache misses and branch mispredictions, it would be interesting to see whether complex out-of-order processors could still be justified.

7 Acknowledgements

This research was supported by a gift from IBM. The simulation facility was provided through an Intel Technology for Education 2000 grant. We would like to thank Josef Burger for providing us a version of SHORE that runs on Alpha machines, and Steve Reinhardt for his suggestions and for graciously allowing us to use his Alpha machines. We would also like to thank Joel Emer for suggesting several future directions for this work.

References

[1] M. Annavaram, G. Tyson, and E. Davidson. Instruction Overhead and Data Locality Effects in Superscalar Processors. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 95–100, April 2000.

[2] D. Bitton, D. DeWitt, and C. Turbyfill. Benchmarking Database Systems A Systematic Approach. In *9th International Conference on Very Large Data Bases*, pages 8–19, October 1983.

[3] D. Burger and T. Austin. The SimpleScalar Tool Set. Technical report, University of Wisconsin-Madison, Computer ScienceDepartment Technical Report #1342, June 1997.

[4] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring Up Persistent Applications. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 383–394, May 1994.

[5] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous Subordinate Microthreading (SSMT). In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 186–195, June 1999.

[6] J. D. Collins, H. Wang, D. M. Tullsen, H. J. Christopher, Y. Lee, D. Lavery, and J. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, page ??, July 2001.

[7] T. P. P. Council. TPC Benchmark H Standard Specification (Decision Support). In *Revision 1.1.0*, June 1999.

[8] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow Analysis of Branch Mispredictions and Its Applications to Early Resolution of Branch Outcomes. In *Proceedings of the 31st International Symposium on Microarchitecture*, pages 59–68, Dec 1998.

[9] Y. Patt, S. Patel, M. Evers, D. Friendly, and J. Stark. One Billion Transistors, One Uniprocessor, One Chip. In *IEEE COMPUTER*, volume 30(9), pages 51–57, Sept. 1997.

[10] A. Roth and G. Sohi. Speculative Data-Driven Multithreading. In *Proceedings of the High Performance Computer Architecture*, pages 37–48, Jan 2001.

[11] C. Selvidge. *Compilation-Based Prefetching for Memory Latency Tolerance*. PhD thesis, MIT, May 1992.

[12] A. Srivastava and D. Wall. A Practical System for Intermodule Code Optimization at Link-Time. Technical Report Technical Report 92/6, Digital Western Research Laboratory, June 1992.

[13] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 11(4):352–357, 1984.

[14] C. Zilles and G. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 172–181, June 2000.