Improving Processor Performance by Dynamically Pre-Processing the Instruction Stream

by

James David Dundas

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical Engineering)
in The University of Michigan
1998

Doctoral Committee:

Professor Trevor Mudge, Chairman

Professor Richard Brown

Professor Ronald Lomax

Professor Karem Sakallah

ABSTRACT

Improving Processor Performance by Dynamically Pre-Processing the Instruction Stream

by

James David Dundas

Chairman: Trevor Mudge

The exponentially increasing gap between processors and off-chip memory, as measured in processor cycles, is rapidly turning memory latency into a major processor performance bottleneck. Traditional solutions, such as employing multiple levels of caches, are expensive and do not work well with some applications. We evaluate a technique, called runahead pre-processing, that can significantly improve processor performance.

The basic idea behind runahead is to use the processor pipeline to pre-process instructions during cache miss cycles, instead of stalling. The pre-processed instructions are used to generate highly accurate instruction and data stream prefetches, while all of the pre-processed instruction results are discarded after the cache miss has been serviced: this allows us to achieve a form of very aggressive speculation with a simple in-order pipeline. The principal hardware cost is a means of checkpointing the sequential state of the register file and

memory hierarchy while instructions are pre-processed. As we discard all pre-processed instruction results, the checkpointing can be accomplished with a modest amount of hardware.

The instruction and data stream prefetches generated during runahead episodes led to a significant performance improvement for all of the benchmarks we examined. We found that runahead typically led to about a 30% reduction in CPI for the four Spec95 integer benchmarks that we simulated, while runahead was able to reduce CPI by 77% for the STREAM benchmark. This is for a five stage pipeline with two levels of split instruction and data caches: 8KB each of L1, and 1MB each of L2. A significant result is that when the latency to off-chip memory increases, or if the caching performance for a particular benchmark is poor, runahead is especially effective as the processor has more opportunities in which to pre-process instructions. Finally, runahead appears particularly well suited for use with high clock-rate in-order processors that employ relatively inexpensive memory hierarchies.

Confutatis maledictis
flammis acribus addictus,
voca me cum benedictus.
Oro supplex et acclinis,
cor contritum quasi cinis,
gere curam mei finis.
Lacrimosa dies illa,
quae resurget ex favilla,
judicandus homo reus.

Excerpted from *Requiem*
— Wolfgang Amadeus Mozart

DEDICATION

To my family.

ACKNOWLEDGEMENTS

I would like to thank Peter Bird, I-Cheng Chen, Brian Davis, Kris Flautner, Bruce Jacob, Viktor Kravets, Chih-Chieh Lee, Charles Lefurgy, Trevor Mudge, Dave Nagle, Yale Patt, Matt Postiff, Mike Riepe, Tim Stanley, Tim Strong, and David Van Campenhout for their many helpful suggestions over the years. I would also like to thank my thesis committee for providing me with advice and perspective on my research. Finally, I would like to thank my father for his many years of encouragement.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Runahead

The severe device count constraint forced upon the design space of a Gallium Arsenide processor led us to consider a number of unorthodox architectural ideas. One of these was runahead pre-processing. The basic idea is to allow a simple, yet very fast, processor pipeline to pre-process instructions during cache miss cycles, instead of stalling.

The pre-processed instructions are used to generate highly accurate instruction and data stream prefetches by detecting cache misses before they would otherwise occur, while all of the pre-processed instruction results are eventually discarded. This allows us to achieve a form of very aggressive speculation with a simple in-order pipeline. The principal hardware cost is a means of checkpointing the sequential state of the register file and memory hierarchy while instructions are pre-processed. Since we discard all results that are computed during runahead episodes after the cache miss has been serviced, the checkpointing can be accomplished with a modest amount of hardware.

By waiting until cache misses occur before generating prefetches, runahead adds a highly responsive feedback component to the memory hierarchy: the greater the cache miss penalty, the more opportunities there are for prefetching, which tend to reduce the frequency of future cache misses. Conversely, if an application enters a phase where its hit rate is high, few prefetches are generated.

1.1 Basic runahead theory

When a runahead processor detects an L1 instruction or data cache miss it records the instruction address of the faulting access and enters runahead mode. A demand fetch request for the missing instruction or data cache line is generated if necessary. The processor also checkpoints the register file, RF, by copying its contents to a backup register file, or BRF. The processor then pre-processes subsequent instructions while the cache miss is serviced. The goal is to generate prefetches for instructions and data that will be needed in the near future. If a pre-processed load or store generates a cache miss the processor can generate a prefetch for the missing line. Similarly, instruction fetch misses in the instruction cache during runahead become instruction stream prefetches. Because the value returned from a cache miss cannot be known ahead of time, it is possible for pre-processed instructions to be dependent upon invalid data. Rather than terminating runahead we allow registers and data cache values to have an explicit "invalid" state during runahead. Denoting this value, INV, requires an extra bit associated with each register in the RF as well as with each word in the L1 data cache (if byte or half-word addressing is allowed, additional bits are required). Pre-processing of most instructions consists of the usual steps of fetch, decode, and execute, with some changes to deal with INV data. Also stores are treated slightly differently. The actions associated with pre-processing can be summarized as follows:

1. **register-to-register** instructions mark their destination register INV if any of their source registers are marked INV. (They can also replace an INV value in their destination register if all sources are VALID).
2. **load** instructions mark their destination register INV if any of three cases arises:
   i. if the base register used to form the effective address is marked INV, or
   ii. a cache miss occurs, or
   iii. the target word in the cache is marked INV as a result of a preceding store during the same runahead episode (see next case).

(They can also replace an INV value in their destination register if none of the preceding three cases apply).

3. **store** instructions do not write data into the cache or main memory. They do, however, mark the referenced cache item INV, if the base register used in address calculation is VALID and the target line is in the cache.
4. **conditional branch** instructions are resolved normally if their branch condition is VALID. If it is marked INV, the outcome is determined by whatever branch prediction strategy the processor employs, and the processor continues to pre-process instructions down the predicted path.
5. **indirect branch** instructions (the target of the branch is obtained from a register) in which the register is marked INV stall the processor pipeline until normal operation resumes.
6. **instruction cache misses** during runahead generate instruction stream prefetches.

The above actions were formulated from straightforward considerations of read-after-write dependencies, however they do not always accurately anticipate what occurs during actual execution. Action 3 does not account for the case when stores cause a cache miss or cannot compute their target addresses because their base register is marked INV. Such stores cannot mark their target word INV. It thus follows that subsequent loads have a small possibility of introducing apparently VALID data into the RF, which should have been INV. Action 4 does not account for the case when an unresolvable conditional branch is mispredicted. Finally, action 5 does not account for the case when an instruction cache miss causes the processor to not pre-process one or more taken branch or ALU instructions, which can insert uncertainty into the RF and PC state during runahead.

To summarize, the above runahead pre-processing actions result in values in the RF that cannot be trusted with certainty: there is a small possibility that a VALID register should be marked INV and vice versa. As the values in the RF during runahead are only used to generate prefetches, they do not affect the sequential state of the machine. Actions taken by the processor during runahead episodes cannot affect program correctness.

After the cache miss that initiated runahead mode is serviced, the processor resumes execution at the PC of the faulting instruction, and the RF is restored from its backup, the BRF. The runahead valid bits in the RF and the L1 data cache are then set to the VALID state. The L1 data cache runahead valid bits for a given line are also set to the VALID state whenever a new line of data is allocated in the cache.

1.2 Some Runahead Examples

An example sequence of code is shown in Figure 1.1. Note that the sub-block runahead valid bits in the L1 data cache are not shown for this example, and that only the first eight general purpose registers are considered. The runahead valid bits for these registers are collectively referred to as the Invalid Register Vector (IRV). A register is marked INV if its corresponding bit in the IRV is a 0.

Figure 1.1 Basic runahead example

Comment	Instruction	IRV State (r0 r1 r2 r3 r4 r5 r6 r7)
dcache miss	`load r1, 0(r2)`	1 0 1 1 1 1 1 1
INV result	`add r1, r2, r3`	1 0 1 0 1 1 1 1
bad address	`load r4, 4(r3)`	1 0 1 0 0 1 1 1
correct result	`sub r6, r2, r5`	1 0 1 0 0 1 1 1
prefetch if miss	`load r5, 0(r5)`	1 0 1 0 0 ? 1 1

The first instruction in the sequence is a load that misses in the L1 data cache, causing the processor to enter runahead. The bit in the IRV corresponding to the destination register of the load (r1) is marked INV (with a zero) in the IRV. The second instruction sources an INV register (r1). Its destination register (r3) is subsequently marked INV. The third instruction is another load. This load cannot properly form its target address, since it sources an INV register (r3). As a result this load cannot generate a runahead prefetch, and has to mark

its destination register (r4) as INV. The fourth instruction can source VALID registers (r6 and r2), which it uses to compute a new value for (r5), which remains VALID. The final instruction in the sequence is another load. This load can compute a VALID address using r5. If the load hits in the L1 data cache, then it marks its destination register (r5) as VALID after reading the data from the cache. If the load misses in the cache, then it marks its destination register (r5) as INV, and generates a prefetch for its target line.

This process continues until the memory hierarchy is able to service the L1 data cache miss corresponding to the first instruction. When this occurs, the processor leaves runahead mode and restarts execution at the instruction that initiated runahead (the first instruction in Figure 1.1). Before the processor can leave runahead mode it has to reset all of the IRV and L1 data cache sub-block valid bits to the VALID state, and perform the 1:1 copy from the BRF registers to their counterparts in the RF.

Once the processor has left runahead mode it restarts execution at the PC of the first instruction shown in Figure 1.1. Since the miss corresponding to the load has already been serviced, it is guaranteed to hit in the L1 data cache. The following add instruction is then able to execute normally. The third instruction may generate a L1 data cache miss. If it does not, then it can execute in the normal fashion. If it does generate a cache miss, then the processor re-enters runahead starting at the second load instruction. Assuming that the second load did not generate a L1 data cache miss, then the fourth instruction (the subtract) can execute normally. Finally, if the last instruction in the example generated a prefetch during the previous runahead episode, and that prefetch has been serviced, then the load will not generate an L1 data cache miss. If the prefetch has not been serviced, then the processor will re-enter runahead mode.

Another runahead example is shown in Figure 1.2.

Figure 1.2 Stores during runahead

Comment	Instruction	IRV State
		r0 r1 r2 r3 r4 r5 r6 r7
dcache miss	`load r1, 0(r2)`	1 0 1 1 1 1 1 1
INV dcache word	`store r2, 0(r1)`	1 0 1 1 1 1 1 1
is r4 valid?	`load r4, 4(r3)`	1 0 1 1 ? 1 1 1

Note that the first instruction in the sequence is the same as that in the previous example. It has generated a runahead-initiating L1 data cache miss, and marked its destination register INV as in the previous example. The second instruction is a store that needs to use an INV register (r1) to calculate its target address. Since it cannot determine its target address, it cannot mark its target word in the L1 data cache as INV, even if it is in the cache. As a result, the subsequent load instruction cannot know for certain if the word it attempts to read from the L1 data cache is INV or not, as it could be read-after-write dependent upon the store instruction (in which case the data in the cache is stale). As these dependencies should be relatively rare, the processor should always assume that any word in the L1 data cache that it reads that is not marked as INV, is actually VALID. However, if a dependence does exist, a load can unknowingly add an “undetectable” INV value to the register file.

A third runahead example is shown in Figure 1.3. This example illustrates the effect of branches during runahead.

Figure 1.3 Dependent branches during runahead

Comment	Instruction	IRV State
		r0 r1 r2 r3 r4 r5 r6 r7
dcache miss	`load r1, 0(r2)`	1 0 1 1 1 1 1 1
dependent branch	`blt r1, loop`	1 0 1 1 1 1 1 1
prefetch if miss?	`load r4, 4(r3)`	1 0 1 1 ? 1 1 1

The first instruction initiates runahead as before. Note that the conditional branch is dependent upon the INV register generated by the load miss (r1). A runahead processor implementing a conservative runahead scheme would simply halt pre-processing at the INV-dependent branch in order to avoid polluting the L1 data cache. A runahead processor implementing a more aggressive runahead scheme would continue to pre-process instructions past the branch, using its branch prediction unit to predict the likely path of future execution. In preliminary tests with our current simulator we found that the aggressive scheme always provided superior performance. For this reason all of the runahead simulation results that we present are for processors models that use the aggressive scheme.

1.3 Dissertation Organization

The dissertation is organized as follows. Chapter 2 discusses previous research work. Chapter 3 describes the methodology. Chapter 4 presents experimental results for a baseline runahead processor model. Chapter 5 presents an analysis of the effectiveness of runahead at both the function and instruction level, as well as a study of register and instruction usage. Chapter 6 describes simulation results for more advanced runahead processor models, in which the effects of instruction cache misses, memory hierarchy bandwidth, and wrong-path effects are eliminated. Chapter 7 describes simulation results for reduced-cost

runahead processor models. Chapter 8 presents a summary of our work. Chapter 9 concludes with a discussion of future work.

Chapter 2

The Memory Latency Problem

Processor clock rates have been increasing at about 40% per year, accounting for much of the 59% per year increase in performance [1]. During this time, the access time of commodity DRAM has been decreasing at only 7% per year [2], resulting in an exponentially increasing cycle time gap between processors and main-memory. Unless clever architectural tricks are employed, Amdahl's Law [2] tells us that memory latency effects will eventually dominate the execution time of applications. There are many different established methods of attacking the problem of memory latency.

2.1 Caches

The classic method of reducing the impact of memory latency is to employ one or more levels of high-speed cache memory [3]. Caches work by exploiting locality of reference, meaning that if a data item is referenced once, then that item, or one near it in the address space, is likely to be referenced in the future. Locality of reference is exploited by holding, or caching, recently referenced data items in a small, fast, memory located close to the processor. While a data item is in the cache, subsequent accesses to it have a much lower latency than a main memory access. Caches work well for applications whose address streams exhibit locality, however many important applications do not. Nevertheless, caches

still form the backbone of high-performance memory systems, and can be made very effective when combined with other latency reduction and toleration techniques.

2.2 Perform other useful work during cache miss cycles

One way to add a measure of memory latency tolerance is to allow a processor to continue to perform useful work while a cache miss is serviced. This can be done in a variety of ways. Non-blocking L1 data caches [4] allow a processor to continue to access the cache while a miss is serviced. Processors that allow the out-of-order completion of instructions [2] can continue to execute instructions while a data cache miss is serviced. This allows a processor to tolerate cache miss latency by attempting to keep the execution units busy while a cache miss is serviced. A conceptually similar approach is taken by coarse-grained multi-threaded processors, which can switch between independent threads of execution when a cache miss is detected [5].

2.3 Use the available memory bandwidth more effectively

Another way to reduce latency is to control the store traffic to the lower levels of the memory hierarchy. Reducing store traffic allows the lower levels of the memory hierarchy to concentrate on servicing demand misses, which must be fulfilled quickly in order to maintain a high rate of execution, unlike store traffic, which is only needed to ensure consistency throughout the memory hierarchy.

The easiest way to reduce store traffic is to use a write-back, as opposed to a write-through data cache [3]. While write-back caches make sense for the lower levels of the memory hierarchy where accesses are less frequent, using a write-back L1 cache can complicate processor design [6].

If the priority of store traffic can be made lower than that of miss fill requests, then the average latency of miss fill requests can be lowered by allowing them to proceed ahead of earlier stores. This requires both a relaxed memory consistency model [2], as well as a means of checking to ensure that miss fill request and store addresses do not conflict. One way of lowering the priority of store traffic is to place them into a buffer, where they can be deferred until the off-chip memory is not busy servicing miss fill requests. The easiest way to do this is to place stores into a FIFO queue called a write buffer [2]. Loads that pass stores in the buffer must compare their target address to the target addresses of the stores in the buffer. The performance of write buffers can be increased by coalescing stores in the buffer that map to the same block of memory. This further reduces the amount of store traffic by combining requests that map to the same line into a single access to the next level of the memory hierarchy. An even more advanced way of controlling store traffic is to employ a write cache [7]. A write cache is essentially a coalescing write buffer with an LRU line replacement policy, as opposed to a FIFO replacement policy, turning the buffer into a cache. This allows the write cache to coalesce even more stores, resulting in an even greater reduction in store traffic.

2.4 Increase the bandwidth of the memory hierarchy

A low latency memory hierarchy is not enough to ensure adequate performance on many applications [8]. Memory bandwidth is also very important. Bandwidth can be increased in many different ways. Cache and main memory bandwidth can be increased by employing wider cache lines or memory buses, pipelining accesses, and interleaving cache and memory banks. Main memory bandwidth can be increased even more by employing

exotic DRAM types, such as RAMBUS, as opposed to traditional commodity-type DRAMs [9].

2.5 Statically schedule loads before stores

One way to reduce the effects of memory latency is to statically schedule code such that loads are moved as far as possible before any subsequent dependent instructions. This will allow an aggressive processor to start loads earlier than usual, increasing the likelihood that some or all of any cache miss latency will be hidden by the time any load-dependent instructions are ready for issue. This approach is limited by the ability of the compiler to statically detect dependencies between load and store instructions, among other things. Although some dependence analysis can be done statically, in the general case it is difficult to move loads before stores due to the problem of memory dependences between them. These potential hazards can only be detected after address calculations have been performed, which themselves, require hazard detection. The solution to this problem is referred to as memory disambiguation [10].

Although pure hardware memory disambiguation techniques, such as the store-buffer, can allow loads to pass stores dynamically, additional performance can be obtained by improving the static code schedule beyond that obtainable with standard compiler optimizations. One way of doing this is to employ hybrid hardware-software memory disambiguation, a recent example of which is [10], which introduced the memory conflict buffer (MCB). The MCB is a hardware device used to detect dependencies between rescheduled load instructions and any subsequent store instructions. This allows the traditional load instructions to be split into two parts: a preload instruction that can be scheduled by the compiler before any prior ambiguous stores, and a check instruction located at the position

of the original (non-rescheduled) load instruction. The preload instruction executes before the ambiguous stores, and records its target address and destination register in an MCB entry. Any stores that execute after the preload, but before the check instruction, access the MCB. If a store target address matches that of any MCB entries, then one or more load-store conflicts exist, and the target registers of the corresponding preload instructions are marked invalid in the MCB. If and when the corresponding check instructions are executed, they access the MCB to see if their corresponding preload conflicted with a store. If a conflict exists, the check instruction causes the processor to branch to a preload-specific fixup routine that fixes the error. This is done by re-executing the load instruction, as well as any load-dependent instructions that were rescheduled by the compiler.

2.6 Executing loads early

The conventional five-stage pipeline [11] uses an adder in the ALU to compute load-store target addresses. This improves performance by reducing the branch misprediction penalty, since the ALU execute stage is the first stage that instructions enter once they are issued. Unfortunately, this comes at the price of a load hazard.

A hardware load-use interlock is control hardware that is added to a pipeline that detects data dependencies, or load hazards, between load destination registers and subsequent dependent instructions [2]. The interlock hardware can stall the pipeline until any data dependencies can be resolved. This results in increased design complexity, however this allows the compiler to ignore the pipeline implementation, if correct operation is all that is needed. An alternative is to require that the compiler to either insert independent instructions, or NOPs, into the slots after a load that can result in a data hazard [11]. This approach

makes the compiler's job harder, as well as that of any future processor designers that may wish to radically change the microarchitecture of a subsequent implementation.

The work in [12] first proposed eliminating the load-use interlock (LUI) in the traditional five-stage pipeline by performing load/store address calculation in a stage before the ALU execute stage. This allows loads to execute in the same stage as ALU operations, which when combined with result forwarding, allows load-data dependent ALU instructions to be issued immediately after a load. This replaces the load-use interlock with an address-generation interlock, which occurs less often in typical applications. For this reason this pipeline organization is often referred to as the address generation interlock (AGI) organization. Unfortunately it delays branch resolution by an additional cycle, resulting in an increased branch misprediction penalty. In [11] this trade-off was evaluated, and concluded that a five stage AGI pipeline had to have a branch misprediction rate no greater than 20% in order to beat the performance of the traditional LUI pipeline. This pipeline organization has been employed in Intel i486, Pentium, Cyrix M1, and R8000 processors [11]. Note that while the AGI organization was initially proposed in [12], they concluded that the extra adder that it required was expensive enough to make the LUI organization more attractive at the time.

Another way to perform loads early in a pipeline is to detect them early, and attempt to access the data cache ahead of time by predicting the load target address. This idea was first explained in [13] where the concept of the load target buffer (LTB) was introduced. The LTB is a table indexed by the processor PC. Each entry of the LTB contains fields that hold the previous target address of a load instruction, the difference between the target addresses of the last two executions of the load (the stride), and some status bits. The LTB is accessed

in parallel with the instruction cache in the fetch stage of a pipeline. If the PC hits in the LTB, then the pipeline is fetching a load instruction. If on a hit, and the status bits indicate that the target address and stride should be trusted based upon the prior history of the load, then a speculative load from the address previous_address + stride is immediately issued to the data cache, when possible. If the speculative load hits in the data cache, and the predicted address is correct, then the load latency is hidden. In [13] it was concluded that the latency to the L1 data cache must be at least five cycles to justify the use of a moderate sized LTB.

In [14] a method is presented to reduce load latency by performing effective address calculation in parallel with data cache access within the cache access stage. They employed a simple carry-free addition circuit that can compute the cache set index portion of the effective address with a single level of logic. Assuming that no carries are generated within, or propagated to, the set index portion of the address, the set index can usually be computed by simply ORing the set index portion of the address operands together. This very fast computation of the set index allows them to perform set index calculation in series with data cache access. The tag portion of the effective address can be computed in parallel with the cache lookup, as well as the actual effective address of the load. This allows them to perform many loads one cycle earlier than would otherwise be the case. If the carry-free addition was incorrect, or if the data cache port is busy, the load is performed in the customary MEM stage of the pipeline.

The authors in [15] went on to combine the idea of the LTB [13] with their earlier work on Fast Address Calculation [14] to reduce load latency even further. They proposed the Base Register and Index Cache (BRIC), a cache of general purpose register values that is

accessed in the fetch stage in parallel with the instruction cache. Each entry in the BRIC contains a register pair consisting of the base and index (if used) register values corresponding to a load instruction. These register values are used in conjunction with the load offset (if any) and some predecode information from the instruction cache to allow load instructions to obtain their address operands by the beginning of the decode stage of the pipeline. The pipeline can then use Fast Address Calculation (FAC) [14] to potentially access the data cache during the decode stage of the pipeline. If this succeeds, then two cycles of load latency are hidden. If the BRIC misses, the register values read during the decode stage can be used to perform the cache access during the execute stage of the pipeline using FAC. This will still hide one cycle of load latency. If the FAC fails, then the data cache is accessed in the usual fashion during the MEM stage of the pipeline.

2.7 Data Prefetching

Data prefetching can reduce average memory latency by bringing data close to the processor before it is needed. The problem of course is selecting which data to prefetch. If data is prefetched into the L1 cache, then useless prefetches will replace other lines. As these other lines were probably fetched as a result of actual misses as opposed to prefetches, replacing these lines with useless prefetches can degrade performance. This is referred to as cache pollution [3]. If data is prefetched into a buffer where it waits for a demand miss before it is moved into the cache, pollution effects can be reduced, although useless prefetches can still reduce performance by increasing memory traffic. Prefetching into a buffer also introduces coherency problems, as the buffer contents must be checked every time that a store is committed.

The most elementary form of data prefetching is provided by having cache lines larger than the largest addressable data item. This brings additional data items into the cache, where they may be used before they are displaced from the cache. Other forms of data prefetching can generally be divided into hardware and software approaches.

2.7.1 Hardware data prefetching

Hardware data prefetching techniques allow the processor to generate prefetch requests without the intervention or knowledge of the compiler. This is more versatile, as it allows prefetching to be employed on legacy code, and across platforms.

The earliest form of hardware prefetching, sequential prefetching [3], generates prefetches for one or more lines located immediately after a referenced line in the address space. When this is done on every access, it is referred to as always-prefetch. Generating sequential prefetches only when a reference to the current line misses is referred to as prefetch-on-miss. It is also possible to perform tagged-prefetch, a variation of always-prefetch, in which sequential prefetches are generated whenever a previously prefetched line is accessed. One of the problems with sequential prefetching is that prefetched lines that are not referenced in the near future can displace useful lines from the cache, resulting in cache pollution. Sequential prefetch can also generate a great deal of additional traffic, which can swamp the connection to memory.

A conceptually similar approach, the data stream buffer, was proposed in [16]. The data stream buffer generates fetch requests for sequential lines after a missing line. These lines are fetched into the FIFO stream buffer, where they can be subsequently placed into the cache if the processor requests them. Since the stream buffer does not place prefetched lines into the cache until they are requested, cache pollution is kept to a minimum. Note that both

the stream buffer and sequential prefetching are conceptually similar to having very large cache lines, however using a stream buffer requires that the cache and buffer be kept coherent at all times. One of the problems with both the stream buffer and sequential prefetching is that prefetches are only generated for lines located after a missing line in the address space. Another problem with using the stream buffer for data stream prefetching is that multiple stream buffers are usually required for adequate performance, multiplying the coherency problem.

In [17] a more advanced sequential prefetching scheme for shared memory multiprocessors was proposed and evaluated. They compared the simple sequential prefetching technique [3], with an adaptive technique of their own design. Their adaptive sequential prefetching technique can dynamically adjust the number of sequential lines that are prefetched after a miss during program execution. This approach can even turn off prefetching if the program is in a region where prefetching is detrimental to performance, and subsequently re-enable prefetching if it detects that it would have benefited from prefetching.

More advanced hardware data prefetching methods attempt to generate prefetches for non-sequential lines. These approaches are particularly suited for processors executing scientific code, which typically access very large sparse matrices. The stride prefetch cache [18] can generate prefetches by using stride information obtained from vector memory operations to generate the prefetch addresses. Unfortunately this only works for vector machines, although one could use the PowerPC load/store with update instructions in a similar fashion under certain circumstances.

Conventional processors can generate strided prefetches by caching information about the history of memory operations. This was first done by [19] with their Reference Predic-

tion Table (RPT). The RPT is a table that contains information about loads and stores executed in the recent past. The table is accessed by a Lookahead Program Counter (LA-PC) which is simply a register that runs ahead of the conventional program counter, using a branch target buffer to (hopefully) stay on the proper path of execution. If at any time the LA-PC "hits" in the RPT, then the LA-PC points to a likely future execution of a load or store instruction. Each entry of the RPT contains the previous target address of the load or store, the difference between the target addresses of the last two executions of the instruction (the stride), and some state information. If the state information for the entry indicates that the reference information can be trusted, then a potential prefetch with the target address previous_address + stride is generated if the target address misses in the cache, and there is no outstanding fetch to that address in progress.

In [20] the Stride Prediction Table (SPT) was proposed. It is conceptually very similar to those developed by [19] and [21]. The primary difference between the SPT and the RPT [19] is that the SPT is indexed by the PC, while the RPT is indexed by the lookahead PC (LA-PC), which can move ahead of the PC. This means that the RPT can generate prefetches earlier than the SPT. The approach taken by [21] is similar to the SPT.

The authors of [19] continued their work with the RPT, and proposed three variants in [22]. The basic scheme uses the PC to index into the RPT, similar to that described in [20]. The lookahead scheme is basically the same as that described earlier in [19], except that an additional iteration count field is added to the RPT that allows the LA-PC to continue to generate prefetches when it has moved multiple loop iterations ahead of the PC. The correlated scheme attempts to correlate prefetches with changes in loop level, unlike the previous table approaches.

The authors of [23] recognized that the reference prediction table (RPT) described in [22] could not keep up with the issue rate of superscalar processors, as it could only scan one instruction at a time. Their basic idea is the same, except that they use a modified branch target buffer (BTB) called a program progress graph (PPG). Instead of outputting the predicted taken target address, the PPG outputs a pointer to the next branch on the predicted path of execution in the PPG. In other words, the PPG provides a way to a way to jump between basic blocks in a rapid fashion. The output of the PPG is applied to their superscalar reference prediction table (SRPT). When the output of the PPG is applied to the SRPT, any cached stride information about any loads or stores in the predicted future basic block of instructions drop out of the SRPT automatically in parallel, and are then used to generate hardware prefetches in the same manner as that described by [22].

2.7.2 Software data prefetching

Software prefetching was first proposed in passing in [3], and expanded upon without evaluation in [24]. The general idea is to add a instruction or instructions to the ISA that can compute a target address using the same addressing modes as the load and store instructions already present in the ISA. These instructions can be used to probe the L1 data cache for the presence of a given block of data. If the desired block is not in the cache, then a prefetch for the missing block is generated. These prefetches are generally non-binding, i.e. they are dropped if they would cause an exception, and they prefetch into the cache, not the register set. In other words, non-binding prefetches cannot cause incorrect program execution. Software prefetching is generally only effective for scientific applications that stride linearly through large matrices in a predictable manner.

In [25] a simple heuristic was proposed to decide where to insert the prefetch instructions. If an innermost loop induction variable is used to generate a load or store address, then the loop step is added to the induction variable and a prefetch instruction for that target address expression is inserted into the code by the compiler. This prefetch instruction can then generate prefetches for data one loop iteration before it is needed. This simple method inserts too many prefetch instructions, resulting in wasted execution slots. In an attempt to solve this problem a heuristic that predicts which accesses are likely to cause cache misses was proposed. The overflow iteration is defined as the maximum number of loop iterations whose data accesses can fit into the cache. Any data dependence that is carried by a loop for more iterations than the overflow iteration is likely to cause a miss. Prefetches are only generated for these accesses. This simple heuristic made software prefetching practical, at least for scientific applications that stride through large matrices in a regular fashion.

Additional software prefetching work was performed by [26]. They developed the concept of the prefetch distance, or the number of loop iterations a data item should be prefetched before its first use. The prefetch distance is based upon system memory latency and the estimated execution time of one loop iteration, and allows them to generate prefetches several iterations before the data will be needed, increasing the likelihood that prefetched data will be available by the time the first use of the data occurs. As the prefetch distance can be fairly large, prefetched data is sent to a buffer where it waits until needed by the cache. This avoids displacing useful data from the cache, however this comes at the cost of keeping data in the buffer and cache coherent.

The authors of [27] considered using software prefetching with a shared-memory multiprocessor. Their scheme was somewhat different as their prefetch instructions could fetch

multiple cache lines. Furthermore, they assume that cache coherence is maintained via software, making all prefetches binding, which forces the compiler to carefully consider control and data dependences before inserting prefetch instructions. This requires the compiler to be very conservative, greatly reducing the performance benefits of prefetching.

In [28] an attempt was made to use software prefetching on integer benchmarks. Very small caches were used with a relatively small miss penalty in conjunction with a prefetch buffer. The prefetch buffer helped to reduce the amount of pollution, which is critical when software prefetching is used for integer codes whose access patterns are typically difficult to predict. Their results were mixed.

The authors of [29] introduced an algorithm for inserting software prefetches that performs two very important tasks. Their algorithm employs software pipelining in order to ensure that the accesses in each iteration of a loop are covered by software prefetches. The algorithm also performs locality and reuse analysis to predict which accesses and loop iterations require prefetches given known cache parameters, leading to the prefetch predicate. For example, if every n-th iteration of a particular access in a loop is predicted to cause a cache miss, then the loop is unrolled by a factor of n, and a single prefetch is generated. This avoids the cost of executing n-1 redundant prefetches.

The general consensus up to this point is that software prefetching is generally only useful for scientific codes that access matrices in a predictable fashion. A number of recently developed software prefetching schemes have been developed to address this shortcoming.

The authors of [30] developed a way to generate useful software prefetches for pointer-intensive integer programs. Their scheme inserts software prefetch instructions at procedure call sites that passed pointers as arguments. The pointers are used to specify the prefetch tar-

get addresses. This takes advantage of the fact that pointer arguments are likely to be dereferenced during a procedure call.

The authors of [31] examined ways to generate software prefetches for pointer-intensive integer programs that employ recursive data structures. A recursive data structure (RDS) is a heap-allocated object, such as a linked-list, tree, or graph, whose individual nodes are linked together via pointers. A node pointer is a pointer within an RDS node that points to an adjacent node. The basic idea is to insert software prefetches for one or more nodes that are likely to be accessed in the future, every time a given node in the RDS is traversed. They came up with two different ways of inserting software prefetches. The first, which they call greedy prefetching, inserts software prefetches for every node pointer in each RDS node. This causes the processor to issue prefetch instructions for all of the possible successor nodes of a given node when that node is traversed. They expanded upon this idea to create history-pointer prefetching. This approach adds a new pointer, called the history pointer, to each node of the RDS. These history pointers point to nodes that are likely to be visited several traversals in the future. Software prefetches that are issued for these pointers are more selective, and are more likely to be serviced by the time the processor visits the future node.

2.7.3 Hybrid hardware-software data prefetching

One of the problems with relying on software prefetch instructions is that even if every dynamic execution of a prefetch instruction generates a useful prefetch, the instructions still waste issue slots. A similar fault can be found with table-based hardware prefetching methods: relying on the hardware to allocate table entries can waste a great deal of hardware. Combining aspects of both methods can result in improved performance.

The authors of [32] took the idea of the reference prediction table [19] one step further by placing their instruction stride table (IST) under compiler control. This allows the use of a much smaller table, on the order of 4-8 entries as opposed to 1K entries or more, as entries are only allocated for accesses for which the compiler is certain will benefit from prefetching. Their IST is also more versatile, since it allows the compiler to not only specify the stride of the prefetches, but the number of prefetches to generate for each IST entry. Unlike the other table-driven approaches, the IST only generates prefetches when the current access misses in the cache. This allows the IST to be placed off-chip.

The authors of [33] combined the basic ideas of the IST [32] and the prefetch predicates of [29] to form the runahead table. The major difference between the IST and the runahead table is that the runahead table can initiate prefetches whenever the processor PC touches an entry in the table. They also eliminate redundant prefetches by adding predicate information to each entry, which is conceptually similar to that employed in [29].

Chapter 3

Runahead Simulation Methodology

The preliminary studies that we reported in [35] did not consider instruction stream, pipeline, or wrong path speculation effects. Wrong path speculation is of particular importance, as instruction pre-processing on the wrong path may generate useless prefetches. Furthermore, the memory hierarchy that the preliminary simulations assumed was rather simple.

Unfortunately, creating a simulator that can accurately model a runahead pipeline is a non-trivial task. Trace driven simulation methods cannot be used to model wrong path effects or, just as important, the state of the register file, caches, and main memory during runahead episodes. This led us to conclude that a processor model that could actually execute code was needed.

We added a significant amount of code on top of an existing simulation tool, ATOM [34], to allow us to obtain both a cycle-accurate model of a pipeline, including wrong path effects during both runahead episodes and normal operation, as well as a way to deal with O/S calls and the like.

3.1 The basic idea behind the simulator

The simulator models a runahead processor consisting of a pipeline and several caches. During normal (non-runahead) operation the simulated pipeline retires benchmark instruc-

tions in lockstep with the workstation that is executing the instrumented benchmark. Immediately after the simulated pipeline retires a given instruction, the workstation retires the same instruction. If, for example, the instruction is a store, then the workstation executes the store immediately after the simulated pipeline retires the store. Subsequent load instructions in the simulated pipeline can access the store data, as well as the rest of main memory, by computing the load target address using simulated register values and using a pointer containing this address to access main memory. This allows the simulated pipeline to treat the workstation's main memory as its main memory, allowing us to avoid having to deal with the problem of modeling the state of main memory.

ALU instructions are simulated exactly as they would behave in a real pipeline. Their operands are read from a simulated register file, the actual result is computed and subsequently committed to the simulated register file, if the instruction reaches the writeback stage of the pipeline without getting squashed. Branch instructions are treated in a similar manner.

3.2 How the workstation and simulator interact

The retiring of instructions in lockstep with the workstation processor is performed by adding a function call before every instruction in the benchmark using ATOM. These function calls form a one-way bridge between the workstation and simulator. Enough information is passed to the simulator by each function call to allow the simulated pipeline to emulate unimplemented instructions, such as system calls and floating point instructions. When the simulated pipeline retires an instruction, the function call has completed, and control is passed back to the workstation processor. The workstation processor then immediately executes the instruction in the benchmark corresponding to that which the simulated

pipeline just retired, thus keeping the workstation and simulated processors in lockstep. Unimplemented instructions are emulated by syncing the state of the simulated processor's register file and PC to that of the workstation processor after unimplemented instructions have been executed by the workstation processor.

3.2.1 Simulation example

Suppose that we have a very short benchmark consisting of the four instructions shown in Figure 3.1:

Figure 3.1 Uninstrumented benchmark

```
PC          Instruction
0           add         r1, r2, r3
4           load        r2, 0(r1)
8           store       r4, 32(r2)
c           store       r2, 36(r2)
```

ATOM is used to insert function calls to two different functions at various points in the benchmark. The first function call, `InitializeSimulator`, synchronizes the register file and program counter state of the simulated and workstation processors at the start of the simulation. This function call is inserted before the first instruction of the benchmark. This ensures that the simulated and workstation processors start execution at the same point in the benchmark, and that their register files contain the same values.

A second function call, `EmulationBridge`, is the link between the sequential state of the workstation and the simulated processor. Every time that `EmulationBridge` is called, the simulated processor "clocks" itself until it retires an instruction in the instrumented benchmark corresponding to its twin in the uninstrumented benchmark. When the call is finished, the state of the simulated processor's register file and updated PC are the

same as that of the workstation processor. Also, the state of main memory, which is accessed by both the simulated processor and the workstation processor, is updated when the workstation processor executes the instruction once the `EmulationBridge` call is finished.

The instrumented benchmark is shown in Figure 3.2. Note that the instructions in the instrumented benchmark that correspond to their counterparts in the original uninstrumented benchmark are shown in bold text. The simulator function calls are also shown. The values under the PC heading correspond to the instruction addresses of the instructions in the uninstrumented benchmark. The IPC, or instrumented-PC, values are the actual addresses of the instructions in the instrumented benchmark. The function calls that are inserted between the instrumented instructions force "sequential" instrumented instructions to be separated by a value of 4+δ bytes in the address space, as opposed to 4.

Figure 3.2 Instrumented benchmark

PC	IPC	Instruction
–	–	`InitializeSimulator(r0, r1, ..., r31, IPC=0, PC=0)`
–	–	`EmulationBridge(r0, r1, ..., r31, IPC=0)`
0	0	**`add r1, r2, r3`**
		`EmulationBridge(r0, r1, ..., r31, IPC=4+δ)`
4	4+δ	**`load r2, 0(r1)`**
		`EmulationBridge(r0, r1, ..., r31, IPC=8+2δ)`
8	8+2δ	**`store r4, 32(r2)`**
		`EmulationBridge(r0, r1, ..., r31, IPC=c+3δ)`
c	c+3δ	**`store r2, 36(r2)`**

Note that every call to `EmulationBridge` has as arguments the contents of the entire integer register set, as well as the instrumented program counter value, as they exist before the workstation processor executes each instruction in the instrumented benchmark

corresponding to the instructions in the uninstrumented benchmark. These values are only used to detect simulator bugs, and to provide a work-around for instructions that the simulator does not implement. These values are simply ignored when the simulated processor is executing user-level integer instructions. The simulated processor actually fetches, decodes, executes, and retires the fixed-point user-level instructions in the instrumented benchmark corresponding to their counterparts in the uninstrumented benchmark. The `Emulation-Bridge` function handles runahead episodes by ending the function call only when the pipeline retires a non-runahead instruction. This allows the simulated pipeline to go down wrong paths, etc., during runahead episodes exactly as a real pipeline would.

3.2.2 Modeling the instruction stream

One of the consequences of relying upon instrumentation is that the instructions that the simulator executes are no longer located at sequential addresses due to the function calls inserted before each instruction. Therefore, the simulated processor maintains a value, IPC, or Instrumented Program Counter, that is used instead of the conventional PC, or Program Counter. Whenever the simulated processor wants to update its IPC in order to determine the fetch address of the next instruction, it has to scan through the text portion of the address space, starting at the un-incremented value of the IPC, until it finds the next "sequential" instruction in the instrumented benchmark corresponding to the next sequential instruction in the uninstrumented benchmark. This is done by looking for a branch-to-subroutine (unconditional branch) instruction to the known starting address of the `Emulation-Bridge` function. The next "sequential" instruction is located a fixed number of instructions after this branch-to-subroutine instruction. This IPC value is used to fetch instructions

using a pointer into the main memory of the workstation. Taken branches and jumps are handled by computing the new value of the IPC in the conventional manner.

As the IPC does not point to sequential addresses, due to the intervening instructions used to perform the `EmulationBridge` function calls, it cannot be used to simulate instruction stream effects such as cache misses. To get around this we need a way to translate the non-sequential IPC values into sequential PC values that can be applied to an instruction cache model. Note that simply dividing the IPC by a fixed integer will not suffice as ATOM frequently inserts a random number of NOPs between function calls. We created a table that can be used to quickly translate IPC values to their PC counterparts in the fetch stage of the simulated processor. This table is created during the `InitializeSimulator` call before the simulator starts execution. These translated PC values can then be applied to a simulated instruction cache in order to accurately model instruction cache miss effects. While these translated PC values are more than accurate enough to use to measure instruction stream statistics, they cannot be used to access the contents of the workstation's memory. This is why our simulated processor has separate L2 instruction and data caches.

3.2.3 Unimplemented instructions

When our simulated processor fetches an unimplemented instruction (floating point or PALCODE instruction), it simply tags that instruction as unimplementable, and allows it to flow down the simulated pipeline as a NOP. If the instruction reaches the writeback stage of the pipeline, the writeback stage detects that it is an unimplemented instruction. It then squashes all of the instructions in the pipeline, including the unimplemented instruction. However, before it can continue, it has to deal with the effects of the unimplemented instruction. As the simulator does not actually execute these unimplemented instructions, it

takes a copy of the updated register file contents as they would be if the instruction had actually executed, as passed to the simulator via the next `EmulationBridge` call, and places it into the simulated register file. This next `EmulationBridge` call also supplies the IPC of the next sequential instruction, which is used to restart the simulated instruction fetch. This is one of the few places where the simulator actually needs ATOM in order to function properly. Note that unimplemented instructions are rarely encountered in the benchmarks that were simulated.

3.2.4 Side effects of instrumentation

Another problem with relying upon instrumentation is that adding the function calls before every instruction in a benchmark expands the size of the benchmark by about a factor of 40. For most benchmarks this is not a problem. However, if a benchmark executable is sufficiently large before instrumentation, the code expansion caused by instrumentation can "break" branches in the benchmark by placing the instrumented branch target out of the portion of the address space that the branch instruction can jump to using an immediate offset obtained from the instruction word. ATOM automatically fixes "broken" branches by trapping and using fixup code to obtain the proper branch target. Unfortunately it is not practical for our simulator to use this approach. We use the processor state passed to the simulator via the `EmulationBridge` calls as a work-around.

The simulated processor automatically compares the register file and IPC state of the simulated processor to that of the workstation processor at the beginning of every `EmulationBridge` call. This was done originally to detect errors while we were debugging the simulator, however we can also use this functionality to fix the branch problem. If the simulated processor executes a "broken" branch it will jump to the wrong target. This will be

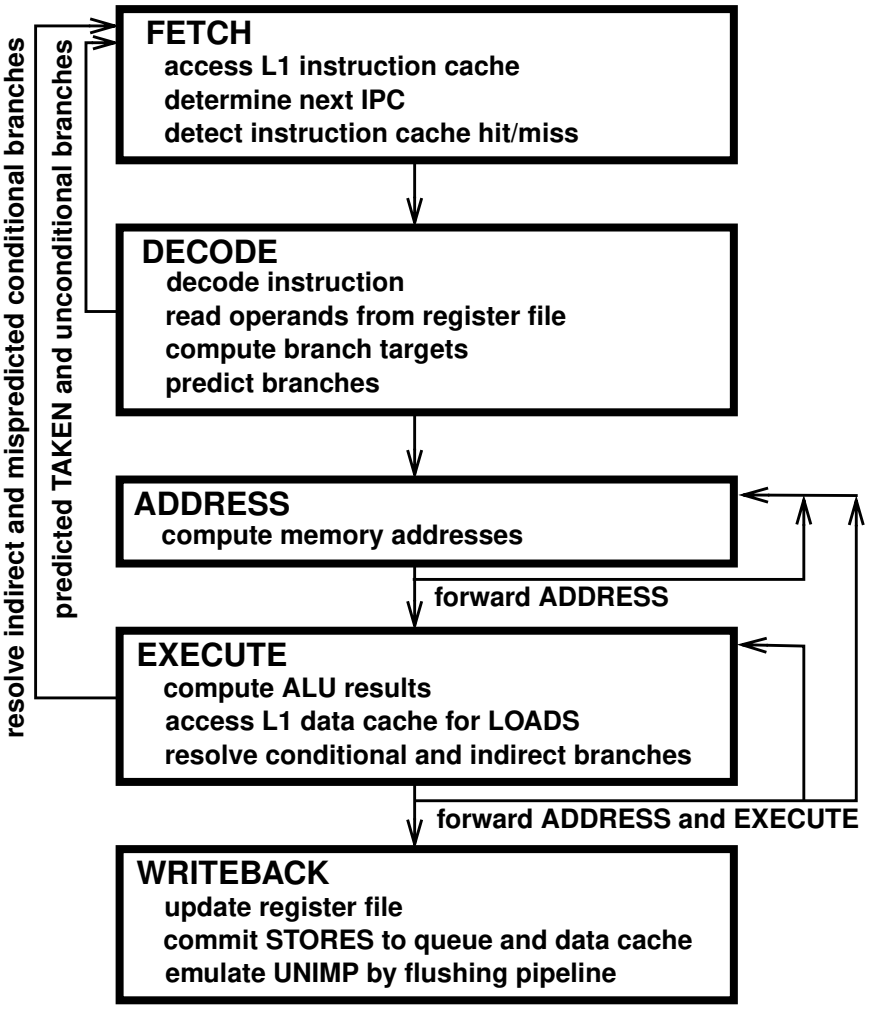
immediately detected after the first `EmulationBridge` call after the branch retires, as a mismatch between the workstation IPC and the simulated processor updated IPC. When this occurs, the simulator flushes the pipeline, and restarts instruction fetch at the updated IPC supplied by the `EmulationBridge` call. This results in a 5 cycle "branch fixup" penalty, which is not significantly greater than our simulated pipeline's 2 cycle mispredicted branch penalty.

Most of the SPEC benchmarks that we examined do not contain any broken branches as a result of instrumentation. The benchmarks that do have this problem only have to perform this fixup a few thousand times, at most, during simulations that last for 100 million instructions. This cannot have a noticeable effect on our simulation results.

3.3 Pipeline Model

As runahead is ideally suited to simple, yet fast, pipelines, we decided to use an in-order five-stage Address Generation Interlock, or AGI [12] pipeline with our simulator. A block diagram of the AGI pipeline is shown in Figure 3.3. We assume that our simulated pipeline is run at a clock speed of 1 GHz.

Figure 3.3 Block Diagram of the AGI Pipeline



NOTE: Address results are for instructions that compute an address and place the result in a register. These are neither load nor store instructions.

The FETCH stage of the pipeline attempts to fetch a single instruction each cycle from the L1 instruction cache. The FETCH stage stalls on L1 instruction cache misses if runahead is not enabled. A branch predictor is also read out of a tagless array of 1024 2-bit counters which employ the Smith Algorithm [37]. This 2-bit prediction is used during the DECODE stage to predict conditional branches, and is updated during the WRITEBACK stage.

The DECODE stage decodes a single instruction each cycle. It also reads instruction operands from the simulated register file, predicts branch outcomes using the output of the branch prediction unit, and computes branch targets. If a conditional branch is predicted TAKEN, or if an unconditional branch is decoded, then the FETCH stage is immediately notified so that it can fetch the branch target instruction. If this prediction is correct, then there is no branch penalty. All branch mispredictions are detected in the EXECUTE stage, resulting in a two cycle misprediction penalty. Indirect branches (branches which obtain their target address from a register) are resolved in the EXECUTE stage, and their target addresses are not predicted.

The ADDRESS stage computes load and store addresses using register values that were read in the DECODE stage. This stage also executes load-address instructions, which simply compute an address and place the result in a register. Note that since we perform address computations in a stage before the EXECUTE stage, we have to provide an address-generation interlock. Basically if a load, store, or load-address instruction is address operand dependent upon an immediately previous ALU or load instruction result, then it must stall in the ADDRESS stage until the EXECUTE result can be forwarded via the forwarding paths. This is a one cycle stall in this pipeline, assuming that the EXECUTE stage itself does not stall.

The EXECUTE stage executes ALU instructions using register operands. It also redirects the FETCH stage when it detects that conditional branches have been mispredicted or when an indirect branch is executed, resulting in a two-cycle penalty. Loads are also executed in this stage by accessing the contents of the workstation's memory using the load target address as a pointer.

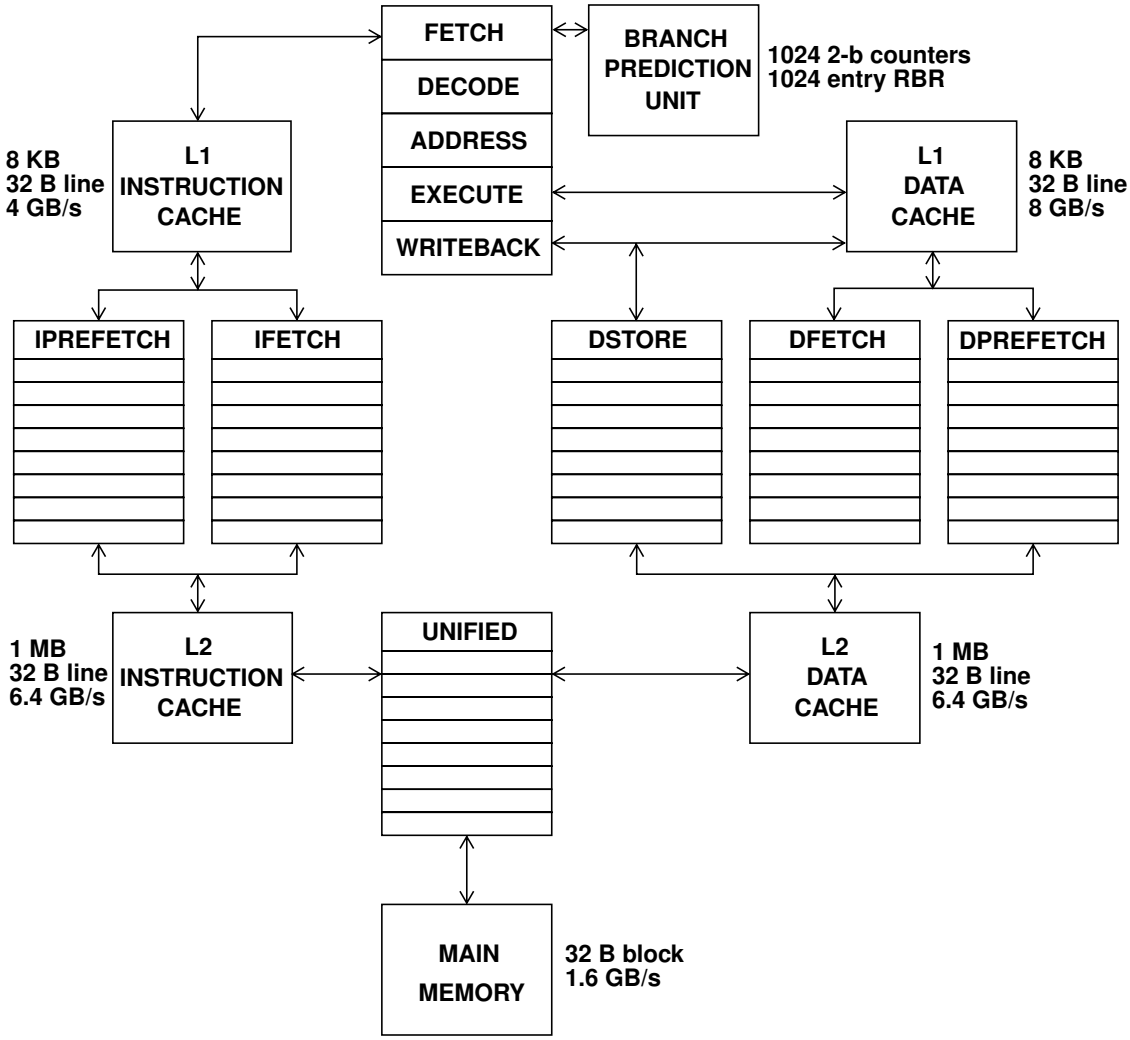
The WRITEBACK stage is where simulated instruction results are committed to the simulated register file. Unimplemented instructions (floating point and PALCODE instructions) are emulated in this stage by flushing the pipeline and syncing the simulated processor IPC and register file contents with the state of the workstation processor at the next `EmulationBridge` call, resulting in a 5 cycle penalty. This pipeline model is slightly unusual in that store instructions commit in the WRITEBACK stage, rather than in EXECUTE with the load instructions. This is a consequence of our simulation methodology, in which the state of main memory is not updated (by the workstation) until simulated instructions actually exit the pipeline. As a consequence, loads that are dependent upon stores must stall in the EXECUTE stage until their dependent store retires. This rarely happens, so we can ignore the occasional stall cycle introduced. An interlock is provided to check for loads that map to the same doubleword as stores. As the L1 data cache is a write-through allocate-on-store-miss cache, stores retire when they update the L1 data cache and add a store-through request to the L2 data cache store-through queue. The 2-bit counter array is also updated with conditional branch outcomes during WRITEBACK.

Pipeline hazards are handled by stalling the pipeline if necessary, and forwarding results where possible via a full complement of forwarding paths. Back-to-back dependent ALU instructions do not require any stalls, nor do back-to-back dependent LOAD-ALU (the ALU instruction is dependent upon the LOAD result) instructions. As was mentioned earlier, an address-generation interlock is provided to ensure that load, store, and load-address instructions obtain the proper operands for their address computations in the ADDRESS stage.

3.4 Processor model including the memory hierarchy

A block diagram of the simulated processor is shown in Figure 3.4.

Figure 3.4 Block Diagram of the Baseline Processor Model



3.4.1 L1 instruction cache

The L1 instruction cache is an 8KB direct-mapped virtual cache with a 32B line size. Each access takes a single cycle, and reads a single 4B instruction word. As the processor model assumes a 1 GHz processor clock frequency, this corresponds to a peak L1 instruction cache bandwidth of 4 GB/s. L1 instruction cache misses are serviced by placing a fetch request into the L1 instruction fetch queue. This is an 8 entry queue that communicates

instruction fetch requests to the L2 instruction cache. The instruction fetch queue also provides the functionality of Miss Status Holding Registers, or MSHRs, making the L1 instruction cache non-blocking, which allows the cache to continue to service accesses during runahead episodes.

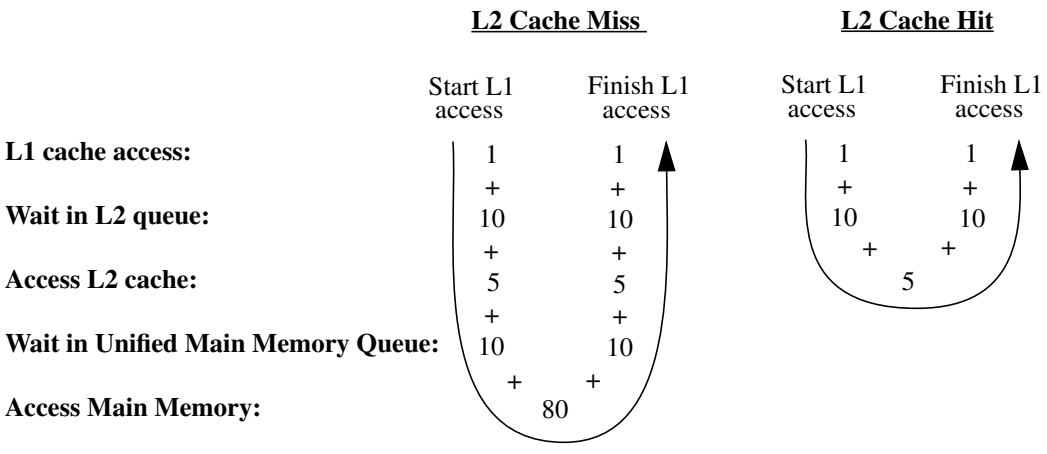
An 8 entry instruction prefetch queue is also provided that communicates instruction prefetch requests to the L2 instruction cache. This queue is identical to the instruction fetch queue in all respects.

3.4.2 L2 instruction cache

The L2 instruction cache is a 1MB direct-mapped virtual cache with a 32B line size. In the baseline model each instruction fetch or prefetch request takes a minimum of 25 cycles to complete after a request has been placed into the instruction fetch or prefetch request queue. It is assumed that L2 instruction cache accesses are pipelined such that fetch and prefetch requests can be pipelined through the L2 cache 5 cycles apart, and that each access of the L2 instruction cache has a latency of 5 cycles. The amount of time required to get requests from the L1 instruction cache to the L2 queue, plus any time it takes to get them to the off-chip L2 instruction cache from the L2 queue is emulated by requiring requests to sit in the L2 queue for at least 10 cycles after the L1 instruction cache places them in the L2 queue. Once the request has been in the L2 queue for at least 10 cycles, it takes 5 cycles to access the L2 instruction cache, assuming that no other higher priority accesses are arbitrating for access to the cache. If the request hits in the L2 instruction cache, then it has to sit in the L2 queue for 10 additional cycles to model the time it takes to transfer the L2 instruction cache line to the L1 instruction cache. Adding all of these times together gives us a minimum time to service an L1 instruction cache miss of 25 cycles. If a fetch or prefetch request

misses in the L2 instruction cache a request is added to the unified main memory queue. Once the request has sat in the unified main memory queue for at least 10 cycles it is eligible to access the main memory, if the memory is available. Accessing the main memory takes 80 cycles, after which the request must sit in the unified main memory queue for another 10 cycles. Once the request has sat in the unified queue for the final 10 cycles the main memory unified queue entry is freed, and the L2 instruction cache is filled. The corresponding request in the L2 queue then has top priority for access to the L2 instruction cache, which takes an additional 5 cycles. After the second L2 instruction cache access occurs, the request is freed in the L2 queue after an additional 10 cycles. When this final 10 cycles is completed, the L1 instruction cache is filled, completing the instruction fetch/prefetch cycle. The timing for an L1 instruction cache access that misses in the L2 instruction cache is therefore: 1 + 10 + 5 + 10 + 80 + 10 + 5 + 10 + 1 = 132 cycles. A timing diagram that illustrates the minimum timing for L2 instruction cache accesses is provided in Figure 3.5. Note that this timing also applies to the L2 data cache.

Figure 3.5 Minimum L2 cache access timing



Other fetch or prefetch requests behind the original request in the L2 instruction fetch or prefetch queue can access the L2 instruction cache while the original request waits on main memory. This makes the L2 instruction cache non-blocking. As 32B fetch or prefetch requests can access the L2 instruction cache 5 cycles apart, the L2 instruction cache has a peak bandwidth of 6.4 GB/s. Demand fetch requests in the fetch queue are given priority over prefetch requests in the prefetch queue in order to improve performance.

3.4.3 L1 data cache

The L1 data cache is an 8KB direct mapped virtual cache with a 32B line size. Each access reads either a 32b word or a 64b doubleword. This results in a peak L1 data cache bandwidth of 8 GB/s. Note that our simulation methodology requires that we perform loads and stores in separate stages. We do not model contention for the L1 data cache between loads and stores in separate stages as a real implementation would perform loads and stores in the same stage. L1 data cache misses are serviced by placing a fetch request in the L1 data cache fetch queue. L1 data cache prefetch requests are handled by the L1 data cache prefetch queue.

As the L1 data cache is a store-through cache, we have to provide a means of sending store-throughs to the L2 data cache. This is done with an 8 entry non-coalescing store queue. Stores place their store data into the store queue when they retire. Stores must stall in the WRITEBACK stage if a store queue entry is not available. This queue does not allow store-dependent fetch or prefetch requests to forward data out of the queue. This is not a significant performance problem as the L1 data cache performs store miss allocation, which makes it unlikely that a store dependent demand fetch or prefetch request will occur soon after a store-through is placed in the store queue.

3.4.4 L2 data cache

The L2 data cache is very similar to the L2 instruction cache, with the exception that it must also handle store traffic. Access latency, size, bandwidth, and queue timing are the same as the L2 instruction cache. A diagram that illustrates the minimum timing for L2 data cache fetch and prefetch accesses is provided in Figure 3.5. Store-throughs are taken off of the store queue one at a time and committed to the L2 data cache strictly in queue order. If a miss occurs, a main memory fetch request is made for the missing line. Unlike the L1 data cache, the L2 data cache is a writeback cache. If a miss in the L2 data cache causes new data to map to a line that has been modified by a store through, then the dirty line is read out of the L2 data cache and written back to main-memory.

The L2 data cache must also respect read-after-write dependencies between store-throughs and fetch and prefetch requests. This must be done to prevent the L1 data cache from obtaining stale data due to a fetch or prefetch "passing" a dependent store-through. This is done by checking the store-throughs already in the store queue for dependencies with new fetch or prefetch requests. A fetch or prefetch request is assumed to have a dependency upon an outstanding store-through if they map into the same cache line address. If there is a dependency between an outstanding store-through and a demand fetch, then the fetch is provided with a pointer (tag) to the dependent store-through. The L2 data cache will not service the dependent fetch until the store-through that it is dependent upon has updated the L2 data cache. The processor handles dependencies between prefetches and store-throughs by dropping prefetches that map to lines for which there is an outstanding store-through in the store queue. This is done before prefetches are placed into the prefetch queue. Note that fetch and prefetch store-through dependencies are rare as the L1 data cache

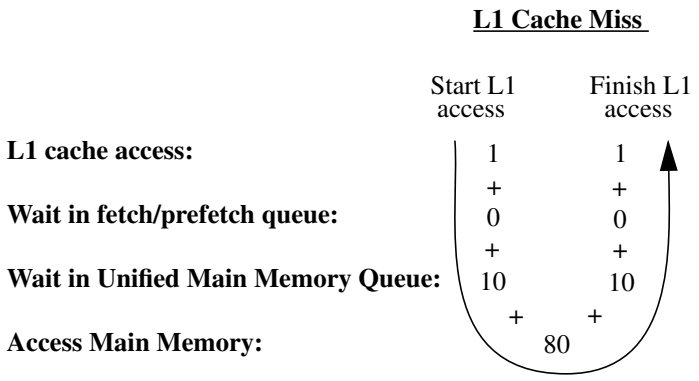
performs store-miss allocation. This makes it unlikely that a demand fetch or prefetch will be generated soon after a store-through to the same cache line address is placed in the store queue.

The L2 data cache attempts to maximize performance by prioritizing accesses: demand fetches are allowed to proceed before prefetches and store-throughs, while prefetches are given priority over store-throughs. This can lead to a deadlock condition if the highest priority fetch request is dependent (as indicated by the tags attached to each fetch request) upon a store-through in the store queue. If this is the case, the requests in the store queue are given priority until the store-through that the fetch in question is dependent upon commits itself to the L2 data cache. Also, prefetches and fetches in the queues can access the L2 data cache while earlier requests that missed in the L2 data cache are serviced by the main memory.

Simulations that do not include L2 Data Cache

Some simulation models do not include an L2 data cache in order to increase the average access time to off-chip memory. When this is done the simulator simply uses the L2 data cache simulation code as a means to arbitrate between L1-L2 data fetch, prefetch, and store-through requests for access to the unified main memory queue. This was done by making two simple modifications to the L2 data cache simulation code: the L2 data cache access time was set to zero and no lines are allocated in the L2 data cache. Also, requests do not have to sit in the fetch or prefetch queues for any length of time before they can be placed in the unified main memory queue. Store-throughs are sent directly from the store queue to the unified main memory queue. The minimum time for an off-chip access is then 102 cycles. A diagram that illustrates the minimum timing for main memory accesses is provided in Figure 3.6.

Figure 3.6 Minimum L1 data cache miss timing for processors without L2 data cache



3.4.5 Main Memory

The simulated main memory is assumed to be a high bandwidth pipelined memory. Each main memory access either fetches or writes back 32B at a time. New accesses can be started 20 cycles apart, resulting in a peak bandwidth of 1.6 GB/s. The main memory access time is 80 cycles, but this does not take into account getting requests to and from main memory itself.

A single 8 entry queue is provided to communicate fetch, prefetch, and writeback requests from the L2 instruction and data caches to main memory. Requests are added to the queue by the L2 caches and are serviced by the main memory strictly in the order in which they are placed in the queue. As with the other queues, requests must sit in the main memory queue for a minimum of 10 cycles to simulate the amount of time that it takes for requests to travel to the main memory. Once fetch and prefetch requests are serviced by the main memory they must sit in the queue for an additional 10 cycles to simulate the amount of time that it takes to transfer their data from the main memory to the L2 cache. The access time of the main memory is 80 cycles, which results in a minimum time to access main

memory from the L2 caches of 100 cycles, when the time that requests must sit in the queues behind other requests is taken into account. Writebacks from the L2 data cache to the main memory only require 90 cycles to complete (minimum) as no data is returned to the L2 data cache, although the queue entry that is occupied by the writeback is not freed for an additional 10 cycles. Access ordering is maintained by using main memory queue entries whose fetch or prefetch accesses eject dirty lines from the L2 data cache to perform write-backs.

3.5 What the simulator does not model

As was mentioned earlier, the simulator emulates, rather than simulates floating point code. This is not felt to be a problem because we intended from the start to only consider integer benchmarks, which contain little or no floating point code. The simulator also emulates, rather than simulates PALCODE (system calls, etc.). The side effects of not simulating O/S calls are minimal, as the SPEC benchmarks do not spend much time in the O/S. Finally, our simulation models do not include virtual memory effects, such as TLB misses. Page faults are modeled in a limited sense, as we drop prefetches that correspond to pages that have not been demand fetched from.

3.6 Processor modifications to support runahead

The most important difference between a runahead pipeline and a normal pipeline is that runahead instructions must be able to exist in the pipeline at the same time as normal instructions, without affecting program correctness. This requires a number of minor modifications to the pipeline control logic.

3.6.1 Hazard logic modifications

Runahead instructions can be read-after-write (RAW) dependent upon earlier non-runahead instructions that are in the pipeline when the processor enters runahead mode. These dependencies should be detected in the usual fashion, and the forwarding paths employed where possible.

However the reverse is not true. Non-runahead instructions cannot be read-after-write dependent upon runahead instructions for two reasons. First, and most important, runahead instructions are highly speculative by nature, and their results (in the general case) must be discarded if the processor is to maintain program correctness. Second, runahead instructions that are in execution represent an attempt to anticipate future events (hence the term "pre-processing") with respect to any subsequently issued non-runahead instructions, and as such these non-runahead instructions cannot be RAW dependent upon them. The pipeline control logic must keep this in mind. This requires that each instruction in the machine be provided with a "runahead valid" (RV) bit that travels with it through the pipeline, in a fashion similar to the usual "valid" (V) bit that allows the pipeline to flag bubbles. If a given instruction in the pipeline has both its RV and V bits set to TRUE, then it is a runahead instruction. If the RV bit is FALSE but the V bit is TRUE, then the instruction is a "normal" (non-runahead) instruction. If the V bit is FALSE, then the "instruction" is a pipeline bubble.

3.6.2 Runahead instruction source and destination valid bits

Each instruction must also have a "runahead destination valid" (RDV) bit that travels with it through the pipeline. The RDV bit is set to indicate if its result is (believed to be) VALID during runahead episodes, and is used to update the IRV when and if the runahead instruction reaches the writeback stage without getting squashed. In order to allow an

instruction to determine if its result is runahead VALID when it executes, the instruction must be able to know if its operands are VALID. This is done by providing a Runahead Operand Valid (ROV) bit for each operand of every instruction in the pipeline. This bit is read out of the IRV at the same time that the register file is accessed when the runahead instruction is in the DECODE stage of the pipeline. If a runahead instruction obtains an operand from a forwarding path, then the ROV bit of the forwarded value is obtained from RDV bit of the instruction providing the forwarded value.

In general, if all of an instructions operands ROV bits are VALID, then its result has its RDV bit set to VALID. If not, then its RDV bit is set to INV. This is true for ALU instructions and branches that update registers (unconditional and indirect branches). Load instructions are handled slightly differently. If a load whose address operand register ROV bit is VALID misses in the cache, then its load destination register has its RDV bit set to the INV state. The load destination register RDV bit is also set to INV if its address operand register's ROV bit is set to the INV state. If the operand register's ROV bit is marked VALID, and the load hits in the cache, then load destination register is marked VALID if the target word in the L1 data cache was not the target of a preceding runahead store. If the word was the target of a preceding runahead store during that runahead episode, as indicated by the L1 data cache runahead valid bit for the target word, then the load destination register has its RDV bit set to the INV state, indicating that was a runahead load-store dependency.

3.6.3 Entering runahead mode

A runahead pipeline treats load and store instructions slightly differently than a normal pipeline. During normal operation a runahead pipeline behaves exactly like a normal pipeline when loads and stores hit in the L1 data cache. When a load or store misses in the

cache, a runahead pipeline makes a demand fetch request for the missing L1 data cache line (if there is no outstanding fetch or prefetch request for the line) and enters the runahead pre-processing mode of operation. If the instruction that missed in the L1 data cache is a load instruction, then the pipeline sets the load destination register RDV bit to the INVALID state. The RDV value is not allowed to update the IRV until the instruction reaches the writeback stage of the pipeline without getting squashed. If the instruction is a store, then there is no destination register to mark. However, the store instruction, as well as all subsequent store instructions during the runahead episode, are not allowed to update any level of the memory hierarchy.

The next thing that happens is that the pipeline sets the RV bits of any non-runahead instructions in the pipeline that were issued after the runahead-initiating instruction to the TRUE state, making them runahead instructions. The FETCH logic also has a global RV bit that is set to the TRUE state. This bit is used to set the RV bits of instructions that are fetched to the proper state. The FETCH logic also saves the IPC of the load or store instruction so that it can re-fetch the instruction once the runahead episode has completed.

3.6.4 Pipeline operation during runahead mode

At this point the pipeline is in runahead mode. If the load or store instruction that initiated runahead reaches the WRITEBACK stage of the pipeline without getting squashed by an earlier non-runahead instruction, then several things happen.

First, the WRITEBACK stage saves the contents of the register file (RF) in the backup register file (BRF). This is done to checkpoint the sequential state of the architected register file. This is assumed to happen in a single cycle. At the same time, if the instruction is a

load, then its RDV bit is used to update the IRV. Store instructions do not update the IRV as they do not write to the RF.

After the runahead-initiating instruction completes in the WRITEBACK stage, subsequent runahead instructions commit their results, if any, to the RF and IRV. The sequential state of the machine is safe as the IPC and RF state were checkpointed when the runahead-initiating instruction reached the WRITEBACK stage.

3.6.5 Resuming normal operation

When the memory hierarchy has serviced the demand fetch (or prefetch if there was an outstanding prefetch for the line in question) corresponding to the load or store miss that initiated runahead mode, the pipeline has to resume normal operation at the point in the instruction stream where it entered runahead mode. This is done by performing the following tasks.

First, it has to notify the FETCH stage to resume instruction fetch at the IPC of the load or store instruction that initiated runahead mode. Second, it copies the checkpointed state of the RF that was saved in the BRF back into the RF. Both of these actions are assumed to take place in a single cycle. The global RV bit in the FETCH stage is also set to FALSE.

Once this is done, the FETCH stage will re-fetch the runahead initiating instruction. However, ignoring any prior instruction cache misses, the later stages of the pipeline will still contain runahead instructions. These instructions can still generate useful prefetches, so they should not be squashed in the general case. However, they must not be allowed to corrupt the newly restored sequential state of the processor. The newly restored RF state is protected by not allowing any remaining runahead instructions in the pipeline to update the RF. The newly restored IPC of the machine is protected by not allowing any remaining

runahead branches or jumps in the pipeline to redirect the FETCH stage. This is done by squashing the first, if any, runahead branch instruction and subsequently fetched runahead instructions that are in the pipeline when the processor resumes normal operation.

3.6.6 Instruction cache miss initiated runahead

In addition to the data stream runahead episodes initiated by load or store data cache misses, a runahead processor can also enter runahead mode on an L1 instruction cache miss. This instruction stream runahead is identical to load- and store-miss initiated runahead episodes, with the exception that the instructions in the missing instruction cache line cannot be pre-processed during the runahead episode. Instead of simply stalling on an L1 instruction cache miss during normal operation, the runahead processor enters runahead mode as described in the previous sections. Instruction fetch is redirected to that portion of the address space that is likely to be close to the proper fetch target after the instructions in the missing instruction cache line. The simplest approach is to attempt to pre-process the instructions located in the cache line sequentially after the missing instruction cache line. If these instructions are in the L1 instruction cache, then they can be pre-processed, starting with the first instruction in the cache line, in an attempt to generate data stream prefetches in the usual runahead fashion. If this line is not in the instruction cache, then an instruction stream prefetch can be generated for the missing line, and an attempt to pre-process the next sequential line can be attempted. This process continues until the instruction cache miss corresponding to the runahead initiating instruction fetch has been serviced, at which time the processor resumes normal operation in the same fashion as for load- and store-miss initiated runahead episodes.

One potential drawback of instruction-miss initiated runahead is that the instructions that are not pre-processed in missing instruction cache lines introduce undetectable INV values to the register file and data cache. If these values are used to generate prefetches then some number of erroneous prefetches can be generated, which may affect performance. The processor can also go down a wrong path if a skipped instruction cache line contained a taken branch, or an instruction that should have modified a branch condition.

3.6.7 Instruction cache misses during load and store miss initiated runahead

It is possible for instruction cache misses to occur during load- and store-miss initiated runahead episodes. If this occurs, the runahead processor redirects its fetch stream to the next sequential line in the instruction stream, which may or may not be in the cache. If it is in the cache, then pre-processing continues at the first instruction in this line. If not, then an instruction stream prefetch can be generated. This approach is similar to instruction-miss initiated runahead, and has the same drawbacks.

3.6.8 Branch prediction

In keeping with the small and simple, yet fast, processor model we chose to simulate a rather simple branch prediction scheme. We use an array of 1024 tagless 2-bit counters using the Smith Algorithm [37] for our dynamic predictors. In our branch prediction scheme, conditional branches always update the dynamic predictor array when they reach the WRITEBACK stage, whether they are runahead branches or non-runahead branches. This allows runahead episodes to train the 2-bit counters for branches that have not yet been encountered dynamically during normal operation, which can potentially improve their accuracy.

Unfortunately, this can cause branch mispredictions if the state of the 2-bit counters get too far ahead of the sequential state of the program. This can happen if a static branch is pre-processed several times in rapid succession, during which the behavior of the branch changes. This can be alleviated by saving runahead branch outcomes in a register, and forcing the processor to use these predictors, when available, instead of the 2-bit counters. This register forms a bridge between the sequential state of the program and the "future" state of the runahead-trained 2-bit counters. We call this register the Runahead Branch Register (RBR).

The operation of the RBR is rather simple, which acts as a 1-bit wide circular queue. Three modulo counters are used to access the RBR. The HEAD counter points to the entry after the last predictor added to the RBR during runahead. The RA_TAIL counter points to either the next predictor to be used during runahead (if RA_TAIL != HEAD), or the location in the RBR that will hold the next conditional branch outcome resolved during runahead (if RA_TAIL == HEAD). The NORA_TAIL counter points to the next predictor to be used during normal operation. If a tail counter is equal to the HEAD counter, then no prediction can be made with the RBR during the mode in question (RA_TAIL during runahead and NORA_TAIL during normal operation).

Note that in this simple scheme it is possible for the HEAD counter to pass the NORA_TAIL counter, causing the RBR to "overflow" and provide incorrect predictions. This possibility is handled by flushing the RBR whenever it mispredicts a branch during either runahead or normal operation. Flushing is performed from the tail pointer that predicted the mispredicted branch to the head pointer: flushing the entire RBR on a misprediction during normal operation, but only the known wrong path portion on a misprediction

during runahead. RBR overflows can be reduced by making the RBR larger than the number of basic blocks pre-processed during a typical runahead episode. An eight entry RBR should provide a comfortable margin of safety. We assume a 1024 entry RBR for all of our studies in an attempt to determine the absolute best performance that can be obtained via this method.

A pseudo-code listing of the behavior of the RBR is shown in Figure 3.7, while a short example of RBR operation is given in Figure 3.8.

Figure 3.7 Pseudo code description of Runahead Branch Register Behavior

- When the processor is reset:

```
HEAD = 0;
RA_TAIL= 0;
NORA_TAIL= 0;
```

- When the processor resumes normal operation after leaving runahead mode:

```
RA_TAIL = NORA_TAIL;
```

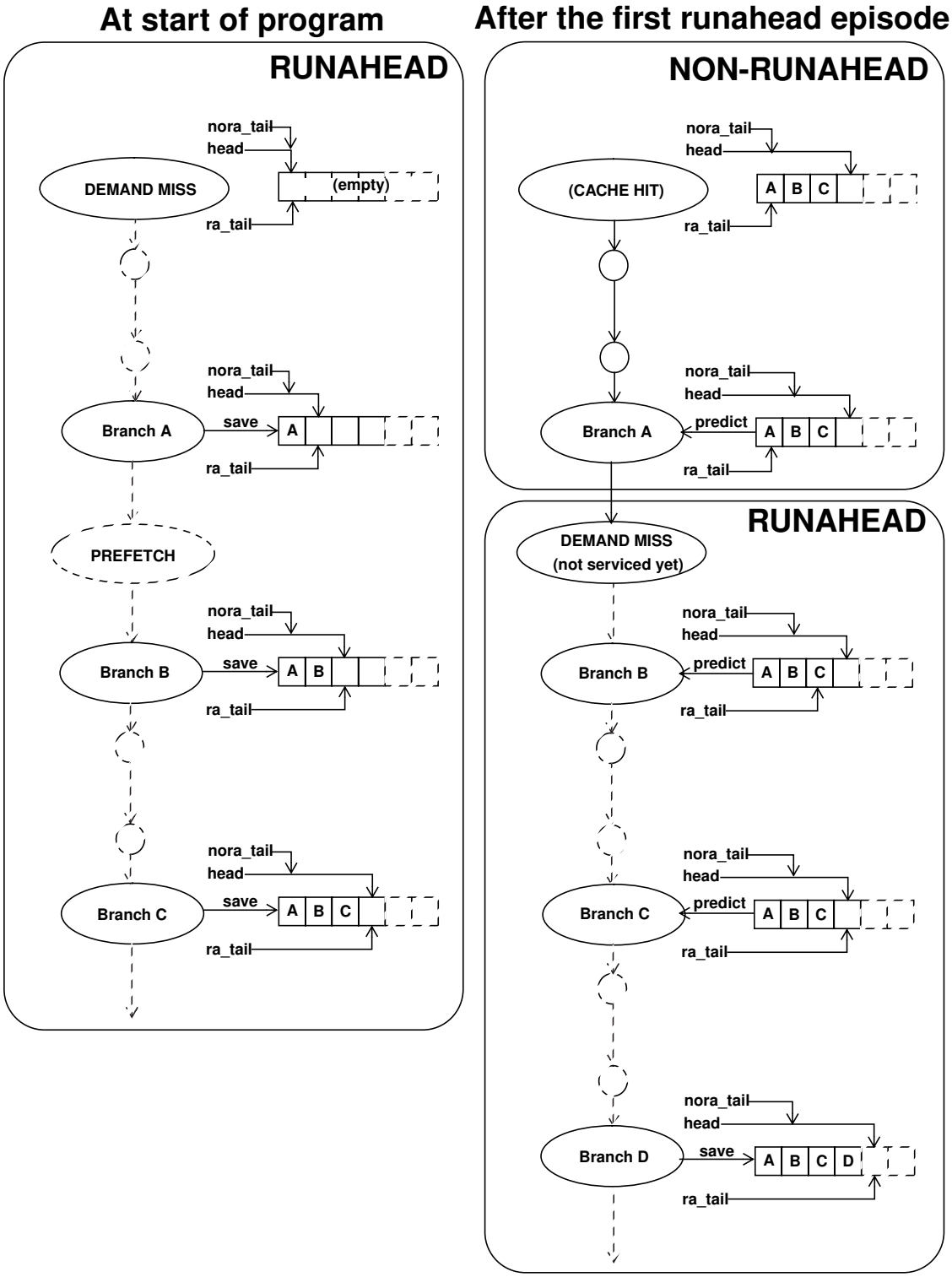
- When branches are predicted:

```
if(instr->runahead == TRUE && RA_TAIL != HEAD) {
    instr->pred_branch_outcome = RBR[RA_TAIL];
    instr->rbr_used_to_predict = TRUE;
    instr->tail_used  = RA_TAIL;
    RA_TAIL++;
}
else if(instr->runahead == FALSE && NORA_TAIL != HEAD) {
    instr->pred_outcome = RBR[NORA_TAIL];
    instr->rbr_used_to_predict = TRUE;
    instr->tail_used = NORA_TAIL;
    NORA_TAIL++;
    RA_TAIL++;
}
else {
    instr->pred_outcome = TWO_BIT_COUNTERS[instr->PC];
    instr->rbr_used_to_predict = FALSE;
}
```

- When branches are retired

```
if(instr->rbr_used_to_pred == TRUE) {
    if(
        instr->pred_outcome != instr->actual_outcome &&
        instr->condition->runahead_valid == TRUE
    ) {
        if(instr->runahead == TRUE) {
            HEAD   = instr->tail_used; // some entries get flushed
            RA_TAIL = instr->tail_used;
        }
        else {
            HEAD   = instr->tail_used; // all entries get flushed
            RA_TAIL = instr->tail_used;
            NORA_TAIL = instr->tail_used;
        }
    }

    if(runahead == TRUE) {
        if(instr->tail_used == HEAD) {
            RBR[HEAD] = instr->actual_outcome; // save result in RBR
            HEAD++;
        }
    }
}
```

Figure 3.8 Runahead Branch Register Example



The example shown in Figure 3.8 represents the state of the RBR and its counters at the start of the execution of a program. The topmost oval on the left hand side of Figure 3.8 represents a load or store instruction that has missed in the L1 data cache, causing the processor to enter runahead mode. The pipeline generates a demand data stream fetch for the missing line in the usual fashion, and continues to pre-process instructions. At some point after entering runahead, the pipeline retires a branch instruction (branch A), and saves the outcome of the branch in the RBR at the location pointed to by RA_TAIL, which is then incremented. A data stream prefetch is generated at some point after branch A, and two more branch outcomes are saved in the RBR (branches B and C). At some point the demand fetch that triggered the runahead episode is serviced, and the pipeline re-fetches the runahead initiating instruction. This is shown on the right hand side of Figure 3.8 with the re-issued load or store instruction represented by the topmost oval. As the demand fetch has been serviced, this instruction retires in the normal fashion, and the pipeline continues to execute instructions. At some point the pipeline reaches branch A, and is able to predict it using the predictor A in the RBR. Eventually, the pipeline reaches the instruction that generated the data stream prefetch in the previous runahead episode. Unfortunately, this prefetch has not yet been serviced, and the pipeline enters runahead again. Branch B is subsequently pre-processed a second time during runahead, only this time there is a predictor in the RBR that can be used by the pipeline due to the overlap between the runahead episodes. The same happens for branch C. After using predictor C to predict branch C, the pipeline continues to pre-process instructions, and eventually pre-processes branch D. Unfortunately, this dynamic branch has not been pre-processed before, so there is no predictor in the RBR that can be used. Therefore, the pipeline predicts the branch using the 2-bit counters. When the

branch is retired, the pipeline saves the branch outcome D in the RBR and continues. At some point when the unserviced prefetch that initiated runahead has been serviced, the pipeline resumes normal execution at the point that the prefetch was generated. When this occurs the RBR contains the outcomes of three branches, B, C, and D, that are still useful. Note that this example ignores the effects of instruction cache misses: they are simply ignored in our baseline runahead branch prediction scheme. As with any branch misprediction, if a skipped branch caused by an instruction cache miss results in an RBR branch misprediction, the affected portions of the RBR are flushed. Also, as our runahead processor continues to pre-process instructions past conditional branches that cannot be resolved with VALID registers, we need a strategy to keep the RBR in sync with the aggressive pipeline. This is done by simply assuming that the branch prediction scheme used is good enough to keep the pipeline on the right path past unresolvable branches most of the time. When an unresolvable conditional branch is retired during runahead the predicted outcome of the branch, obtained from the 2-bit counters, is saved in the RBR.

3.6.9 Unimplemented instructions during runahead

While the pipeline can emulate unimplemented instructions during normal operation by flushing the pipeline and re-syncing the state of the simulated processor to that of the workstation processor, this cannot be done during runahead episodes. During runahead episodes the simulator is no longer retiring instructions in "lockstep" with the workstation processor, so there is no way for it to use the workstation as an emulation work-around for unimplemented instructions.

This leaves the runahead processor with two possible courses of action when it attempts to retire an unimplemented instruction during runahead. It could simply treat the unimple-

mented instruction as a bubble, and assume that the instruction would not re-direct the instruction stream. This would work fine as long as the instruction is a floating point operation, or a similar more-or-less innocuous instruction. We have chosen a simpler approach, in which our pipeline halts if it attempts to retire an unimplemented instruction. If this occurs during runahead the pipeline is flushed when normal operation resumes. If this occurs during normal operation the next call to `EmulationBridge` detects the unimplemented instruction in the WRITEBACK stage and performs the pipeline flush and state synchronization. As unimplemented instructions are rare in SPEC integer benchmarks this approach should not significantly affect performance.

3.6.10 Avoiding segmentation faults during runahead simulation

Due to the highly speculative nature of runahead it is possible for the processor to generate erroneous prefetches. As our simulated pipeline actually reads load data values from the workstation's main memory when it simulates loads during both runahead and normal operation, it is possible for our simulator to generate a segmentation fault if it attempts to load a value from a speculative address that is incorrect. These accesses would quickly kill the simulation if they were not detected before being allowed to access the contents of memory.

We solved this problem in a two-fold fashion. First, we observed that if a runahead load target address hits in either of the simulated data caches, then we know for certain that it is safe for the simulator to access that address in the workstation's memory as that line must have been successfully allocated in the simulated cache at some point.

Segmentation faults can still occur for prefetches that in a real machine would have to go all the way out to main memory. We detect potential segmentation faults for these prefetches by maintaining a linked list of all of the virtual page numbers that have been

demand fetched by the simulated processor during normal operation. This list is checked whenever a data stream prefetch reaches the simulated main memory. If the virtual page number corresponding to the prefetch does not exist in the linked list, then there is a very good chance that a segmentation fault will occur if we allow the runahead prefetch to access the contents of the workstation's memory. These prefetches are dropped.

The performance of the linked list access is improved by "bumping" page number entries that hit in the list to the head of the list. This reduces the average number of list nodes that have to be examined. Note that this list acts as an infinite TLB used with a main memory that never swaps out pages.

A simpler approach is used to detect instruction fetches or prefetches that could cause a segmentation fault. We avoid checking for bad addresses as long as instruction stream fetches hit in the L1 instruction cache. If an instruction fetch or prefetch misses in the simulated instruction caches, we can determine if it is an illegal instruction address by checking to see if it maps into the known limits of the text space for the instrumented benchmark. These limits are obtained via ATOM when the simulator is initialized. If an attempted instruction fetch or prefetch address falls outside of these limits, then the instruction fetch or prefetch is dropped, and the FETCH stage halts. This can happen during normal operation on a mispredicted branch or jump, in which case fetch is restarted when the misprediction is detected in the EXECUTE stage of the pipeline. This can also happen during runahead if the pipeline goes down a wrong path. When this happens, the FETCH stage is restarted if the fetch attempt was caused by a mispredicted branch during runahead, or when normal operation resumes when the runahead-initiating fetch has been serviced.

3.7 Benchmarks

We chose to use the GO, PERL, VORTEX, and IJPEG benchmarks from the SPEC'95 Integer Suite [38] for the bulk of our simulation studies. The STREAM benchmark [39] was also used in order to stress the memory hierarchy. As STREAM is a floating point benchmark we had to modify the code such that all of the data types were type `unsigned long` before we could use it with our simulator. The arguments that are passed to each benchmark are provided in Table 3.1, "Benchmark Arguments," on page 60. Descriptions of each SPEC benchmark follow, all of which are assembled from [38]. Simulations of the SPEC benchmarks were run for 100M instructions, while STREAM was run for 10M instructions. All of the caches were cold-started, and statistics gathering was initiated at the beginning of each simulation.

VORTEX

VORTEX is a single-user object-oriented database transaction benchmark which exercises a system kernel coded in integer C. The VORTEX benchmark is a derivative of a full OODBMS that has been customized to conform to SPEC CINT95 guidelines.

Transactions to and from the database are translated though a schema in which the benchmark is pre configured to manipulate three different databases: mailing list, parts list, and geometric data. The benchmark builds and manipulates three separate, but inter-related databases based on the schema. The size of the database is scalable, and for CINT95 guidelines has been restricted to about 40MB. VORTEX been modified to not commit transactions to memory in order to remove input-output activity from this CINT95 (component) benchmark.

The workload is modeled after common object-oriented database benchmarks with modifications to vary the mix of transactions.

GO

Go is a cpu-bound integer benchmark. It is an example of the use of artificial intelligence in game playing. Go plays the game of go against itself. The benchmark is stripped down version of a successful go-playing computer program. The benchmark is implemented in ANSI C (with function prototypes). There is a great deal of pattern matching and look-ahead logic. As is common in this type of program, up to a third of the run-time can be spent in the data-management routines.

IJPEG

The IJPEG benchmark performs jpeg image compression and decompression with various parameters. This is a cpu intensive benchmark.

PERL

The PERL benchmark performs text and numeric manipulations consisting of anagrams and prime number factoring. As much as 10% of the time can be spent in routines commonly found in libc.a: malloc, free, memcpy, etc.

3.7.1 System Issues

The benchmarks were compiled using the DEC OSF/1 AXP C Compiler version 3.11 supplied with OSF/1 version 3.2. The "-std1 -O2 -non_shared -r" flags were used as arguments to the compiler. Instrumentation was performed with version 2.29 of ATOM running on DEC 3000 workstations employing ALPHA 21064 CPUs. No flags were passed to ATOM.

Table 3.1 Benchmark Arguments

Benchmark	Data Set
VORTEX	"vortex vortex.raw" **reference data set** vortex.raw contains: MESSAGE_FILE vortex.msg OUTPUT_FILE vortex.out DISK_CACHE bmt.dsk RENV_FILE lendian.rnv WENV_FILE lendian.wnv PRIMAL_FILE vortex.pml PARTS_DB_FILE parts.db DRAW_DB_FILE draw.db EMP_DB_FILE emp.db PERSONS_FILE persons.1k PART_COUNT 16000 OUTER_LOOP 1 INNER_LOOP 14 LOOKUPS 250 DELETES 8000 STUFF_PARTS 8000 PCT_NEWPARTS 0 PCT_LOOKUPS 0 PCT_DELETES 0 PCT_STUFFPARTS 0 TRAVERSE_DEPTH 3 FREEZE_GRP 1 ALLOC_CHUNKS 10000 EXTEND_CHUNKS 5000 DELETE_DRAWS 1 DELETE_PARTS 0 QUE_BUG 1000 VOID_BOUNDARY 67108864 VOID_RESERVE 1048576
GO	"go 10 13" **requires just over 100M instructions to run to completion**
IJPEG	":ijpeg -image_file specmun.test.ppm -compression.quality 25 -compression.optimize_coding 0 -compression.smoothing_factor 90 -difference.image 1 -difference.x_stride 10 -difference.y_stride 10 -verbose 1 -GO.findoptcomp" **requires just over 100M instructions to run to completion**
PERL	"perl scrabbl.pl < scrabbl.in" **reference data set**
STREAM	N = 400000 NTIMES = 1 OFFSET = 0 **requires just over 10M instructions to run to completion**

Chapter 4

Baseline Runahead Experiments

In this chapter we evaluate the runahead processor model described in Chapter 3. The intent is to provide an overview of the general characteristics of runahead. Subsequent chapters provide a more detailed analysis of runahead, as well as an examination of low-cost implementations.

4.1 CPI results

The most important result that we can present for these preliminary simulations is the improvement in CPI (Cycles Per Instruction) that can be obtained via runahead. Bar charts that illustrate the CPI obtained for our baseline runahead processor model are shown in Figures 4.1 through 4.5. There are three stacked bars in each plot. The height of each bar, as enumerated on the y-axis, represents the number of processor cycles that each 100M instruction simulation required to complete. Each bar is broken down in to 9 parts, representing the contribution to processor CPI that each instruction class plus instruction cache misses had to the overall CPI of the simulation. These classes include the following:

- L1 instruction cache misses
- Load instructions
- Store instructions
- Load-address instructions (these compute an address and place it into a register)
- ALU instructions

- Indirect branch instructions
- Unconditional branch instructions
- Conditional branch instructions
- Unimplemented instructions (floating point and PALCODE)

The left-most of the three stacked bars represents the overall CPI of a baseline processor that does not employ either runahead or prefetching of any kind. The center stacked bar represents the CPI of the runahead processor. The right-most stacked bar does not represent a specific simulation, rather it represents the number of instructions (execute counts) of each class that were executed during normal operation by the processor. Cycles that the processor spends in load- and store-miss initiated runahead episodes are attributed to load and store instructions in the CPI plots.

The overall CPI of each simulation is provided at the top of each bar. In order to ease the comparison between the runahead and non-runahead simulation results, the load, store, and instruction cache miss portions of the runahead processor stacked bar have their percentage reductions over their equivalents in the non-runahead processor located next to them. Note that the percentage reduction in CPI for loads and stores do not include cycles in which loads and stores are retired: the latter are represented as the load and store portions of the execute counts stacked bar. These percentage reduction figures include both data cache miss, store queue, load-store dependency, and address generation interlock stalls. The load-store dependency stalls are a by-product of our simulation methodology, which requires us to execute loads and stores in separate stages of the pipeline.

The first plot, shown in Figure 4.1, is for the GO benchmark. The overall CPI of the runahead processor is a moderate 3.02, while the CPI of the runahead processor is 2.24. Runahead was able to reduce overall CPI by a respectable 26%. About one-half of the CPI

improvement is attributable to reductions in instruction cache misses (75% reduction in instruction cache miss CPI) due to instruction stream prefetches generated by runahead. The rest of the CPI reduction came about as a result of data stream prefetching for loads and stores. The load portion of pipeline CPI was reduced by 26%, while the store portion was reduced by 35%.

The second plot, shown in Figure 4.2, is for the VORTEX benchmark. The overall CPI for VORTEX is rather high at 4.00, while the CPI of the runahead processor is 2.51, corresponding to a 37% reduction in CPI. As was true for the GO benchmark, about half of the improvement came from reductions in instruction cache misses, with runahead reducing the instruction cache miss portion of CPI by 69%. The load portion of pipeline CPI was reduced by 27%, and the store portion was reduced by 43%.

The third plot, shown in Figure 4.3, is for the STREAM benchmark. Runahead was able to reduce the overall CPI from 18.96 to 4.41, a 77% reduction. Runahead is able to improve performance a great deal, with the load contribution towards pipeline CPI reduced by 83%, and the store portion reduced by 79%. This superior performance is the result of four factors. First, the data cache miss rate is high, providing many opportunities for the processor to enter runahead mode. Second, the benchmark strides linearly through memory, allowing the runahead processor to nearly always compute load and store addresses correctly during runahead. Third, the benchmark itself is very small, resulting in a very small instruction cache miss rate. This effectively eliminates instruction cache miss effects during runahead episodes. Fourth, the processor does not suffer from branch mispredictions for STREAM, as the benchmark consists of a sequence of loops that are executed many times.

The fourth plot, shown in Figure 4.4, is for the IJPEG benchmark. Runahead was able to reduce the overall CPI from 1.55 to 1.30, corresponding to a reduction in CPI of 16%. In spite of this relatively small improvement, runahead was able to reduce the load portion of pipeline CPI by 44%, and the store portion by 27%. These excellent load and store CPI improvements do not translate into a correspondingly large improvement in overall CPI as IJPEG is ALU intensive, which can be seen in the execute counts bar. The situation on the instruction stream side was similar for IJPEG: runahead was able to reduce the instruction cache miss portion of CPI by 78%, but as the instruction cache miss rate for IJPEG is low, this does not translate into a major performance improvement.

The results for the PERL benchmark are shown in Figure 4.5. Runahead was able to reduce overall CPI 34% to 2.86 from 4.30. The store contribution to pipeline CPI was reduced 42%, while that of loads was reduced 38%. The largest component of the overall CPI for the non-runahead processor is attributable to instruction cache misses. Runahead was able to reduce instruction cache CPI by 52%, the least of the instruction cache miss CPI improvements of all of the benchmarks that were simulated.

Figure 4.1

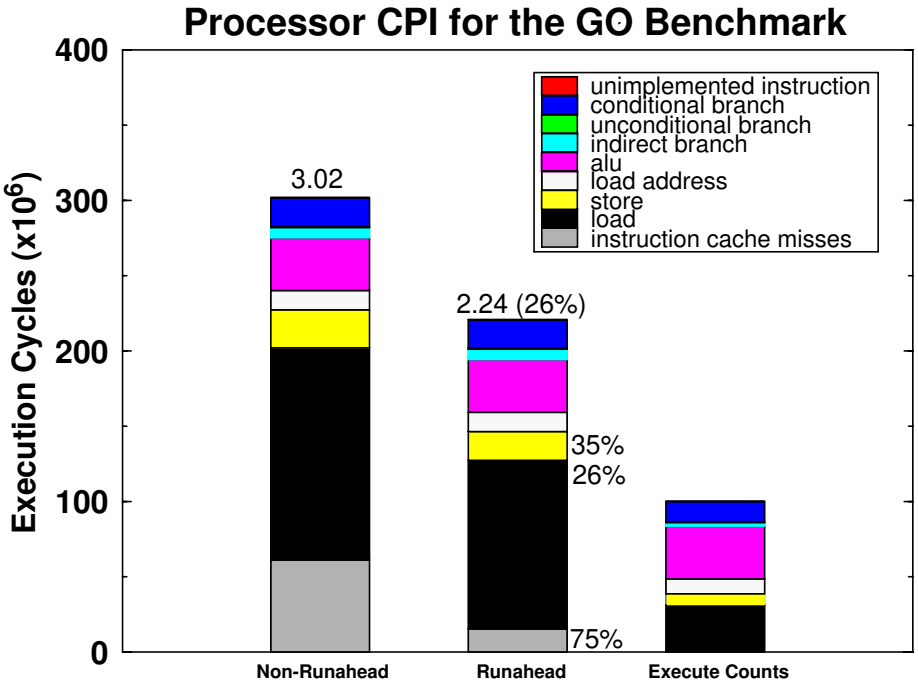


Figure 4.2

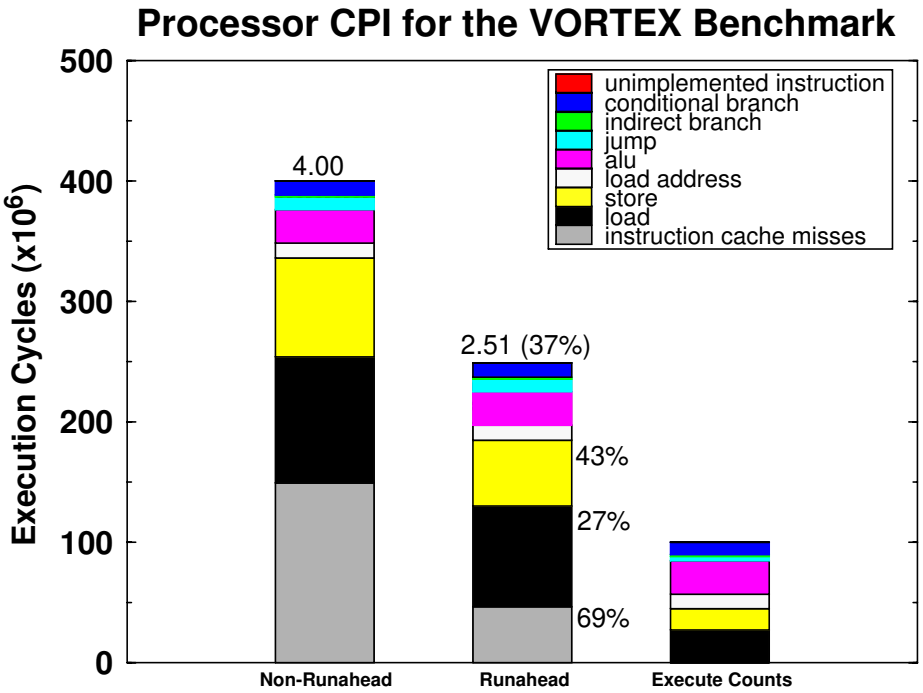


Figure 4.3

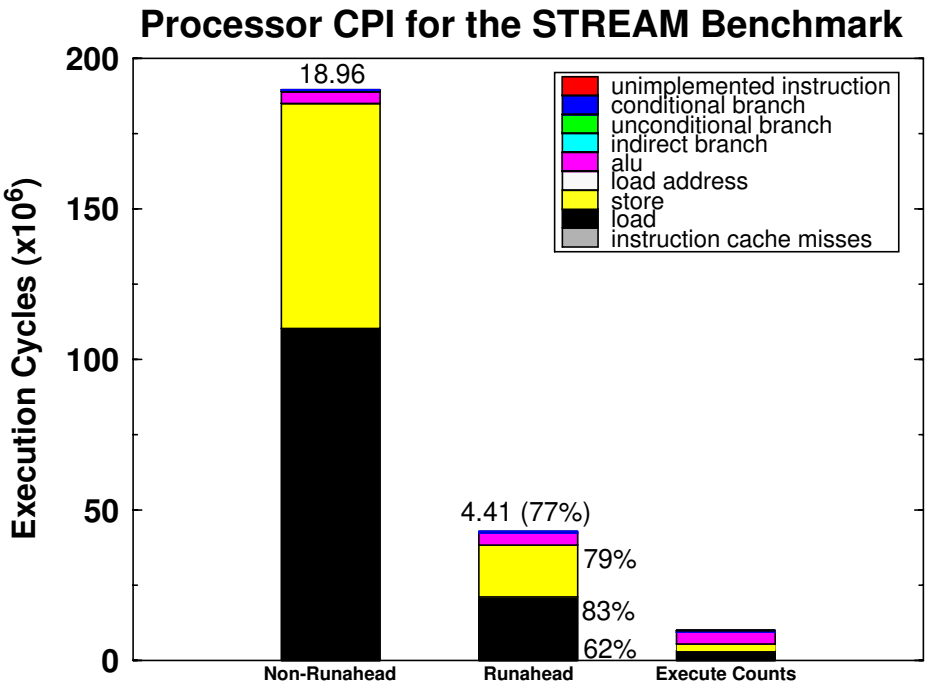


Figure 4.4

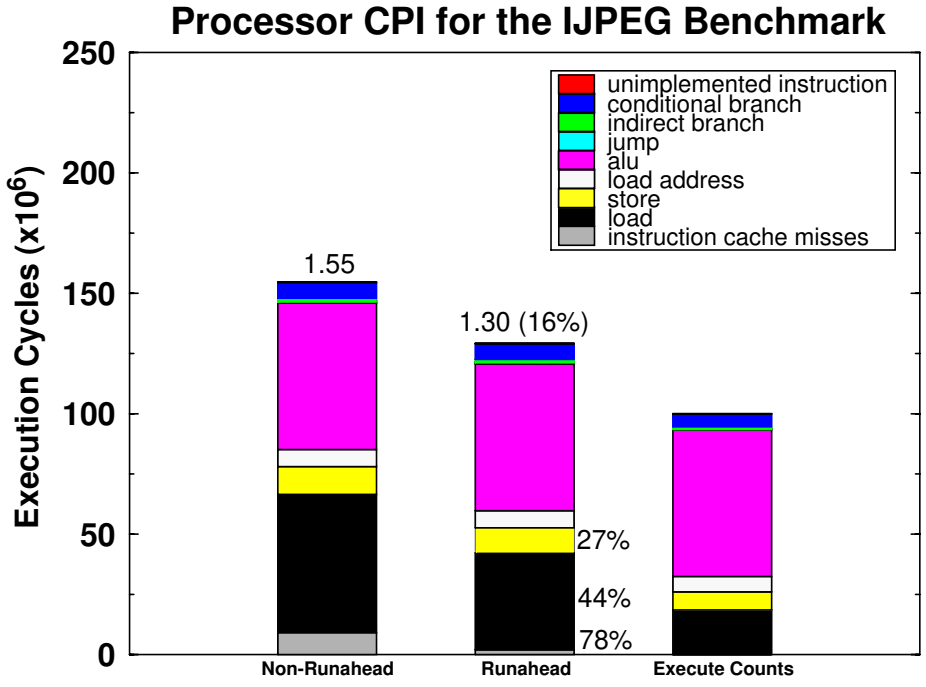
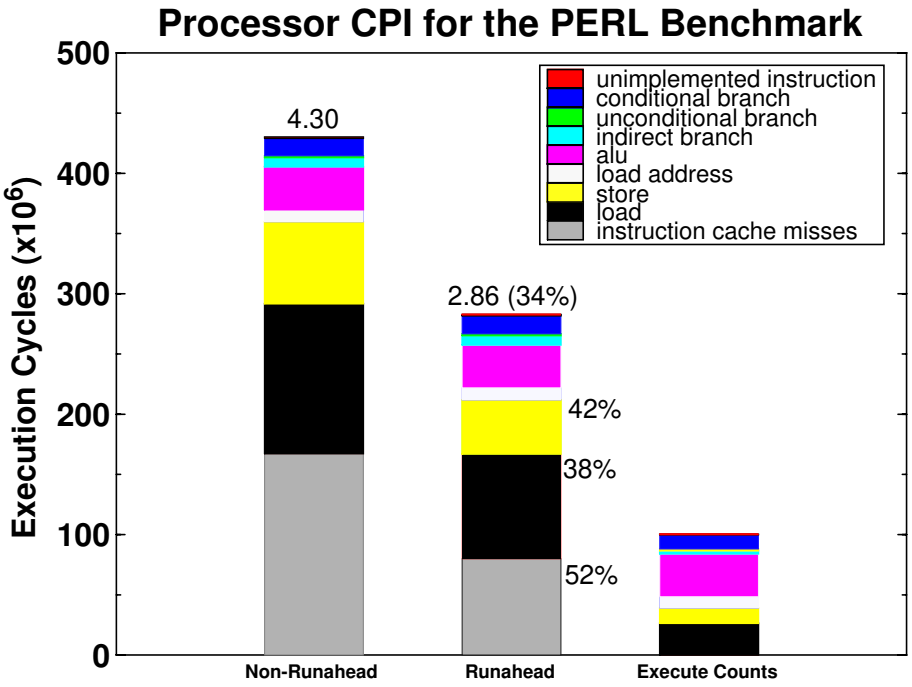


Figure 4.5



4.2 Memory Bandwidth

We measured the average bandwidth at various levels of the memory hierarchy during our simulations and present the results in this section. Plots of the average memory system bandwidth are shown in Figures 4.6 through 4.20. Each plot consists of two stacked bars: one for the runahead processor, and one for a non-runahead processor. Each of these bars has its bandwidth broken down into its component parts. Data stream bandwidths are broken down into store-through/writeback, prefetch, and demand fetch components. Instruction stream bandwidths are broken down into prefetch and demand fetch components. Note that we do not present bandwidths for the L1 caches, as these numbers are not particularly enlightening for a runahead processor which more or less constantly accesses the L1 caches. Also, note that our bandwidth numbers are for accesses that actually supply data: cache

accesses that result in misses do not count towards our bandwidth numbers. Our memory hierarchy requires all L2 cache accesses that miss to access the cache a second time; in other words we do not simulate fetch-around. All main memory accesses count of course, as misses do not occur at that level of the memory hierarchy.

4.2.1 Main memory bandwidth

The average main memory bandwidth for the GO benchmark is shown in Figure 4.6. Note that both the runahead and non-runahead bandwidths are rather low, at 21.49 and 1.98 MB/s respectively. This is normal for the SPEC benchmarks, which tend to fit in large L2 caches. Even so, the runahead processor was able to increase the average main memory bandwidth by a factor of ten over that of the non-runahead processor. This is a consequence of prefetch traffic and the resulting CPI improvements, which reduce the amount of time available for the main memory to service a given number of requests. Note that relatively few instruction prefetches made it out to the main memory. This is a consequence of the benchmark largely fitting in the 1MB L2 instruction cache. Interestingly, there was very little writeback traffic from the L2 data cache to the main memory for the GO benchmark.

The average main memory bandwidth for the VORTEX benchmark is shown in Figure 4.7. This benchmark requires significantly more main memory bandwidth (40.4 MB/s and 92.7 MB/s for the non-runahead and runahead processors respectively) than GO. VORTEX generates a significant number of prefetch requests that reach main memory, enough of which are useful enough to significantly reduce the number of demand data fetches. Instruction prefetches are kept to a minimum, while there is a significant amount of store traffic.

The main memory bandwidth for the PERL benchmark is shown in Figure 4.8. As with VORTEX a significant amount of the main-memory bandwidth is used by both the

runahead and non-runahead processors, however the average bandwidths are much lower than the available bandwidth. The runahead main-memory bandwidth (80.3 MB/s) is slightly more than twice that of the non-runahead main-memory bandwidth (39.6 MB/s). Note that more than half of the demand fetch bandwidth is eliminated when runahead is employed.

The main memory bandwidth plot for the STREAM benchmark is particularly interesting, and is shown in Figure 4.9. The non-runahead processor produces a respectable average bandwidth of 323 MB/s, however this is greatly overshadowed by the bandwidth of the runahead processor at 1.39 GB/s. This huge average runahead main memory bandwidth for STREAM is very close to the peak 1.6 GB/s bandwidth that our simulated main memory can provide. Although STREAM is a somewhat contrived benchmark, it is does indicate that runahead can readily use a high bandwidth memory hierarchy when an application exhibits particularly poor caching behavior. Note that most of the traffic for the runahead processor are data stream prefetches, while the majority of the remainder is store traffic. Few demand instruction or data fetches are presented to the main memory.

The main memory bandwidth for the IJPEG benchmark is shown in Figure 4.10. Note that this benchmark requires relatively little main memory bandwidth, as do most of the SPEC benchmarks. The non-runahead processor only produces an average bandwidth of 1.81 MB/s, while the runahead processor produces a somewhat higher bandwidth of 5.19 MB/s. Most of the bandwidth consists of data fetches and prefetches, with most of the remainder consisting of instruction prefetch and fetch traffic. There is virtually no writeback traffic from the L2 data cache.

Figure 4.6

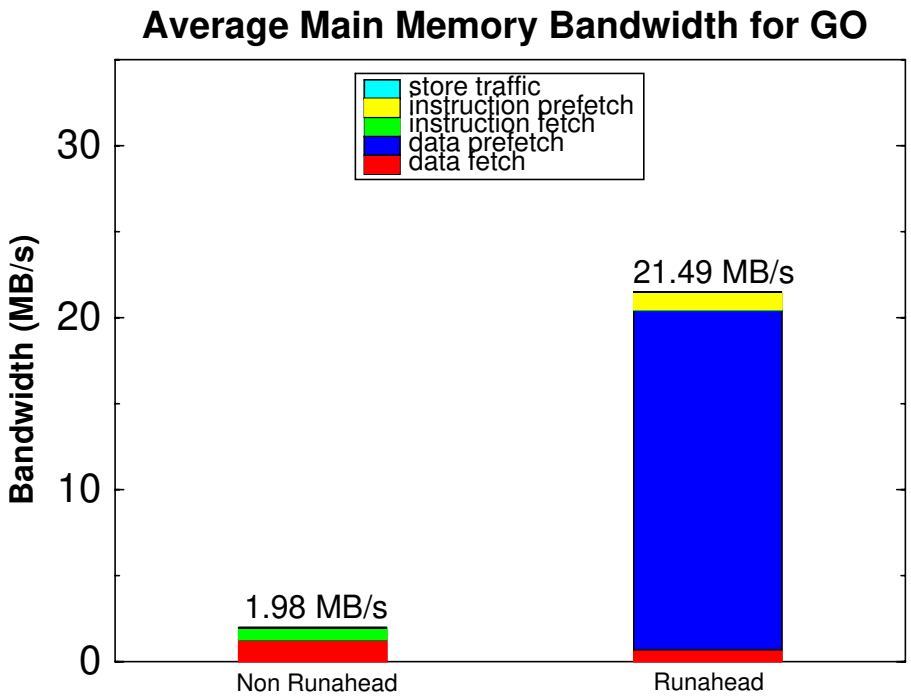
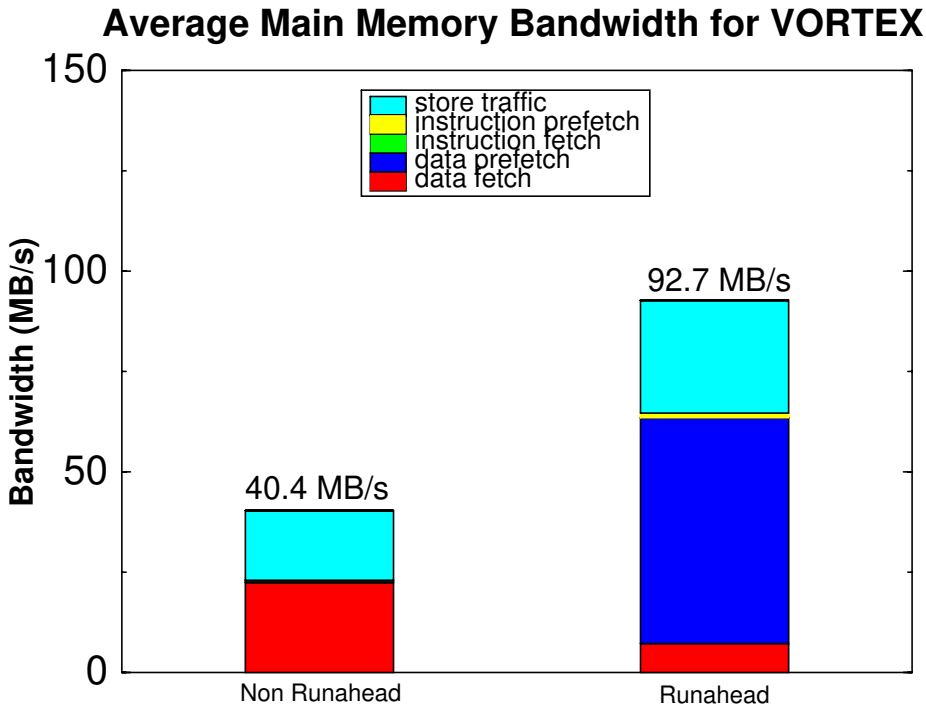


Figure 4.7



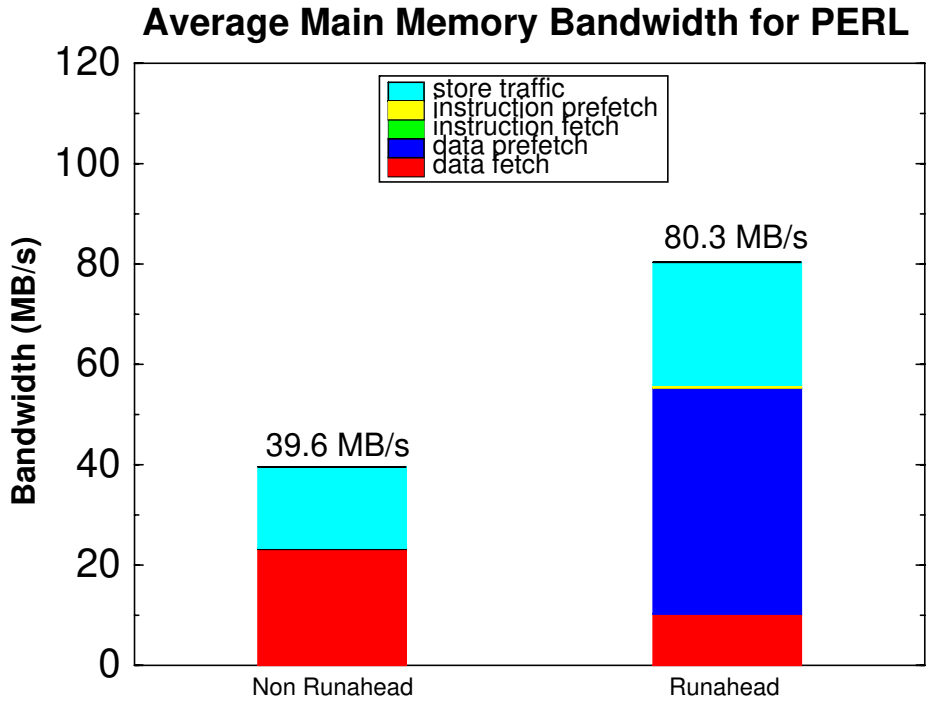


Figure 4.8

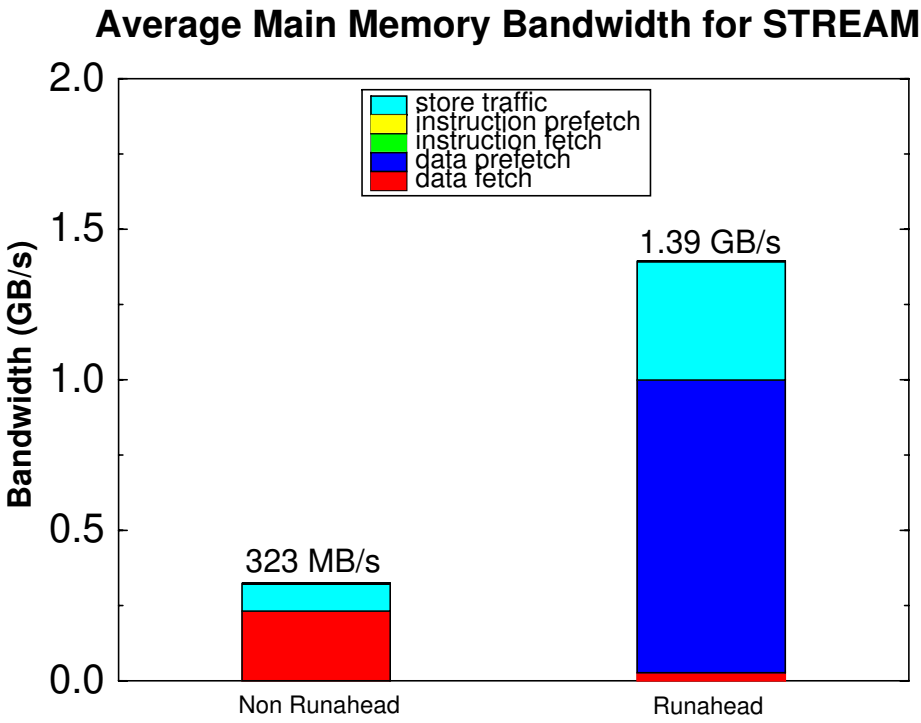
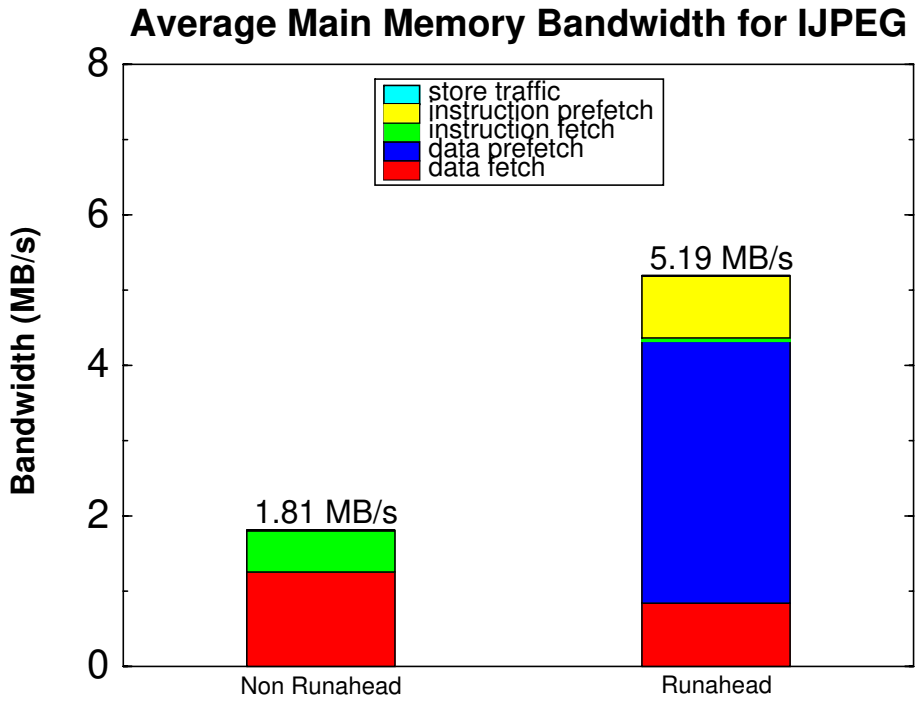


Figure 4.9

Figure 4.10



4.2.2 L2 data cache bandwidth

Unlike the relatively low main memory bandwidth produced for most of the benchmarks, the small 8KB L1 data cache used in our simulations requires a large amount of traffic between the L1 and L2 data caches. Store traffic was a significant source of bandwidth for all of the benchmarks, but did not even come close to saturating the L2 data cache as none of the benchmarks used even half of the available bandwidth.

The average L2 data cache bandwidth for the GO benchmark is shown in Figure 4.11. The non-runahead processor used an average of 1.27 GB/s, of which about two-thirds of which were store-throughs from the write-through L1 data cache to the writeback L2 data cache. The remaining traffic consists of demand data fetches. The runahead processor used a significantly greater average bandwidth of 1.77 GB/s. The store traffic still makes up about

two-thirds of the total, however nearly half of the remainder is made up of data prefetches. These relatively high average bandwidth figures imply that the GO benchmark is able to make good use of the 6.4 GB/s peak bandwidth of the L2 data cache.

The L2 data cache bandwidth for the VORTEX benchmark, shown in Figure 4.12, has even higher average bandwidths than those obtained for GO. The non-runahead processor uses an average of 1.72 GB/s, most of which consists of store-throughs. The runahead processor used a significantly larger average bandwidth of 2.77 GB/s. As with the GO benchmark, nearly half of the non-store bandwidth consisted of prefetch traffic.

As was the case for the main memory bandwidth, the L2 data cache bandwidth for the STREAM benchmark, shown in Figure 4.13, is particularly interesting. The non-runahead processor used a relatively low average bandwidth of 608 MB/s, most of which consists of store-through traffic. The data fetch and prefetch L2 data cache bandwidths for the runahead and non-runahead processors are virtually the same as their main memory counterparts, which is intuitive given the lack of locality in the benchmark. The runahead processor generated an average L2 data cache bandwidth of 2.61 GB/s. Note that more than one-third of the traffic was for data prefetch requests, and that very few demand fetches were requested. The bulk of the traffic is of course store traffic. The total L2 data cache bandwidths for both processor models are higher than their main memory counterparts due to the writeback L2 data cache, which coalesces multiple L2 data cache store-throughs into a single main memory writeback transaction.

The L2 data cache bandwidth for the IJPEG benchmark, shown in Figure 4.14, has nearly the same bandwidths for both the non-runahead and runahead processors, at 1.87 and

2.23 GB/s respectively. Nearly all of the traffic consists of store-throughs, however more than half of the non-store traffic for the runahead processor consists of prefetches.

The runahead processor was able to significantly increase the L2 data cache bandwidth for the PERL benchmark, shown in Figure 4.15, with an average runahead bandwidth of 1.82 GB/s as compared with 1.23 GB/s for the non-runahead processor. As was the case with the IJPEG benchmark, most of the L2 data cache traffic consists of store-throughs.

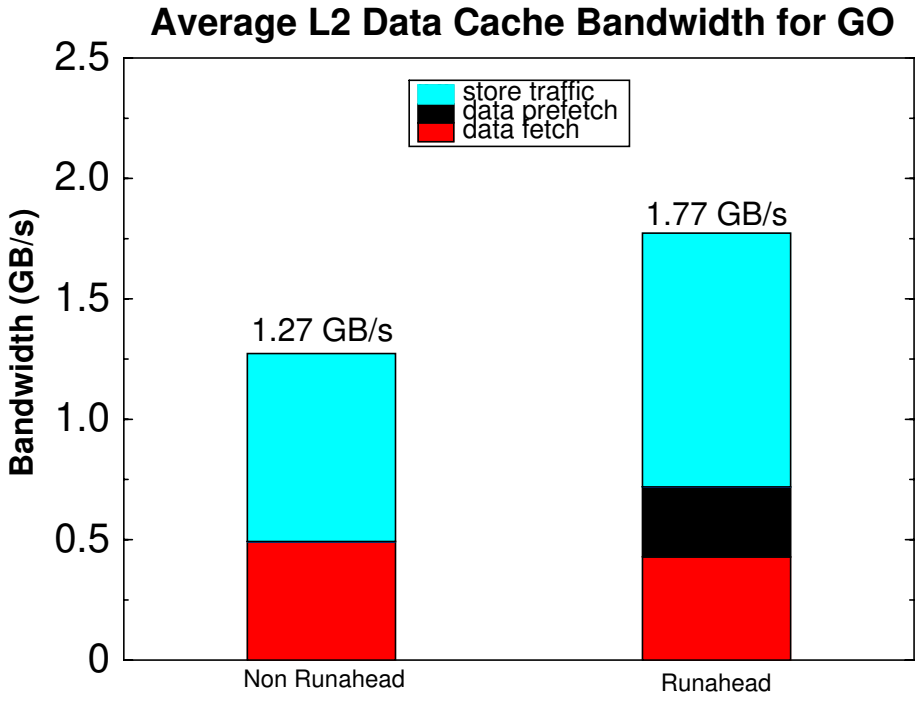


Figure 4.11

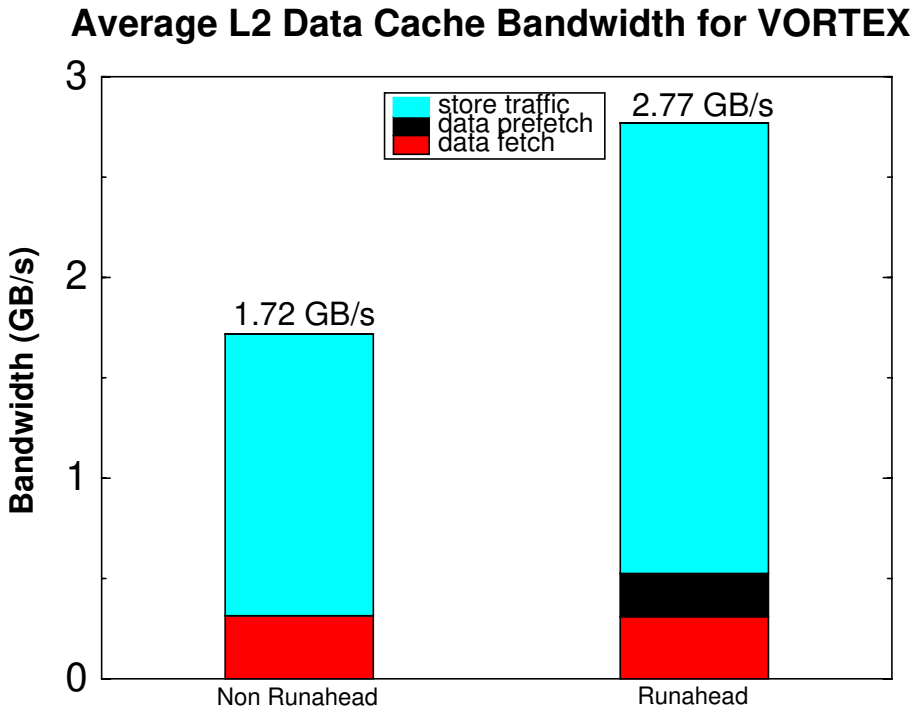


Figure 4.12

Figure 4.13

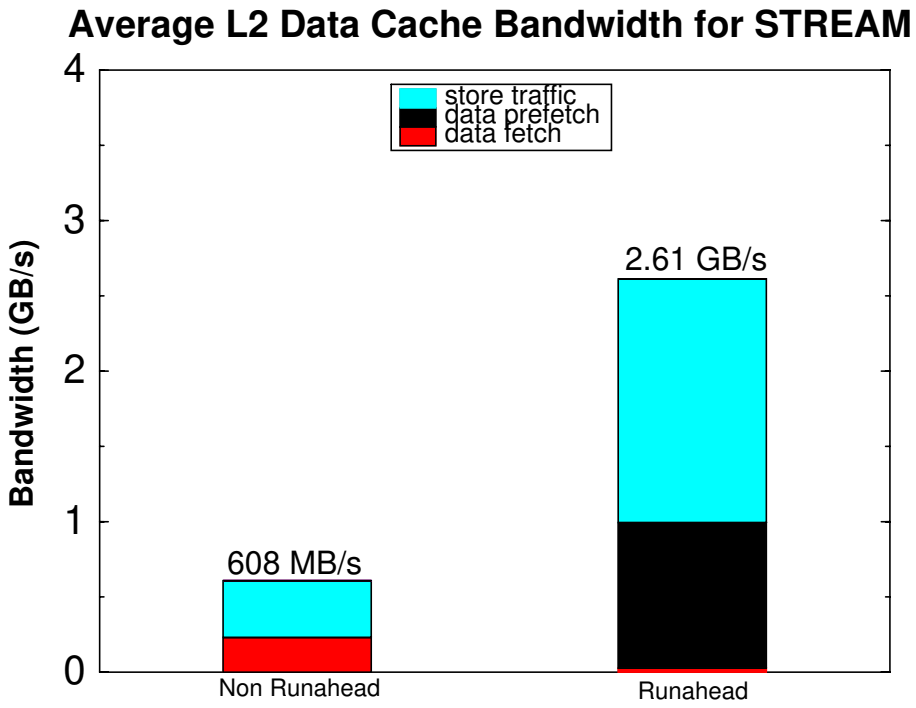


Figure 4.14

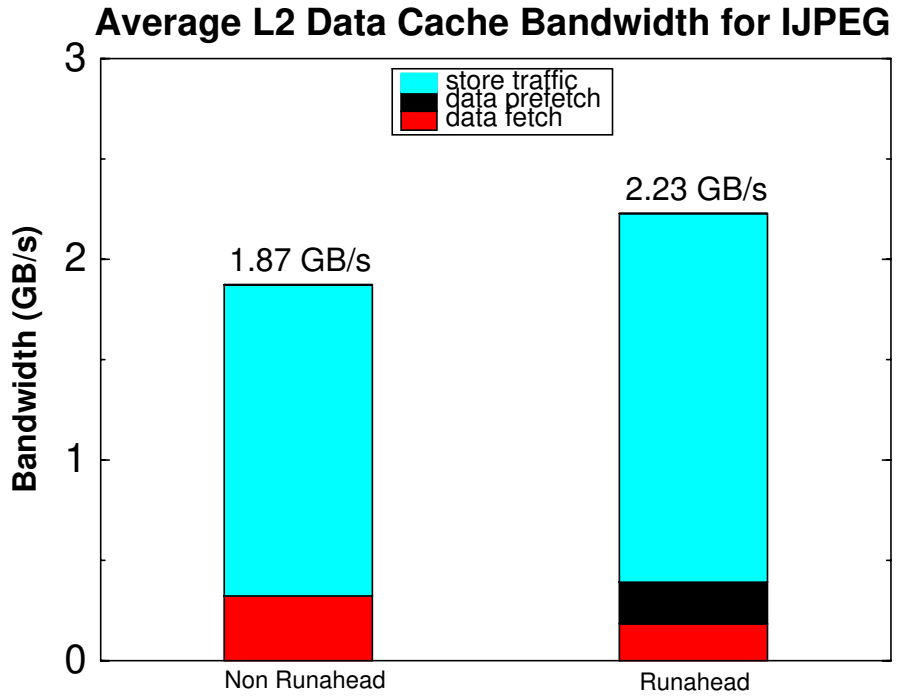
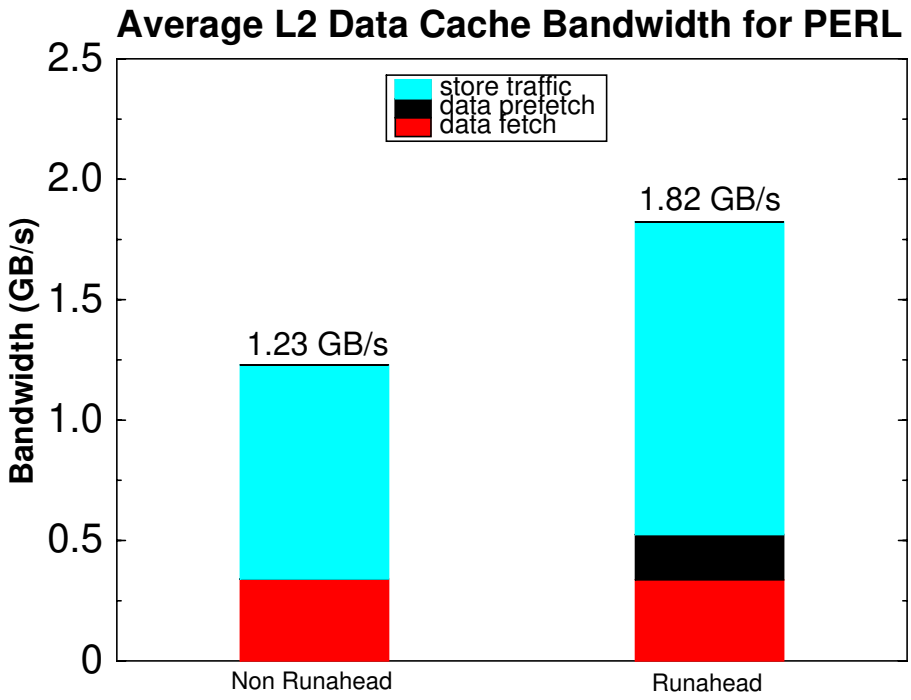


Figure 4.15



4.2.3 L2 instruction cache bandwidth

The L2 instruction cache bandwidths are not nearly as interesting as those for the data stream side of the memory hierarchy. This is a consequence of the relatively good instruction stream caching behavior of most of the SPEC benchmarks. STREAM of course easily fits into the L1 instruction cache.

The L2 instruction cache bandwidth for the GO benchmark is shown in Figure 4.16. The non-runahead processor has a low average bandwidth of 352 MB/s, while the runahead processor comes in at 858 MB/s. The overwhelming majority of the runahead processor's L2 instruction cache bandwidth is made up of prefetch requests.

The results for the VORTEX benchmark, shown in Figure 4.17, exhibit significantly greater bandwidths. The non-runahead processor uses an average of 583 MB/s, while the

runahead processor uses 2.0 GB/s of bandwidth, the majority of which is for prefetch requests.

The L2 instruction cache bandwidth for the STREAM benchmark, shown in Figure 4.18, is rather uninteresting. The non-runahead processor uses an average of 3.72 KB/s of bandwidth for demand fetches, while the runahead processor uses 23.3 KB/s, most of which consists of prefetch requests.

The runahead processor produced a large increase in L2 instruction cache bandwidth for the PERL benchmark, shown in Figure 4.19. The bandwidth increases from 580 MB/s for the non-runahead processor to 2.61 GB/s for the runahead processor. Virtually all of these accesses hit in the L2 instruction cache, as instruction stream fetch and prefetch requests comprise a rather small proportion of total main memory bandwidth (Figure 4.8).

The instruction stream bandwidth for the IJPEG benchmark, shown in Figure 4.20, is relatively low, even when runahead is employed, when compared to the figures obtained for the other SPEC benchmarks. This is a result of the relatively low L1 instruction cache miss rate for the IJPEG benchmark.

Figure 4.16

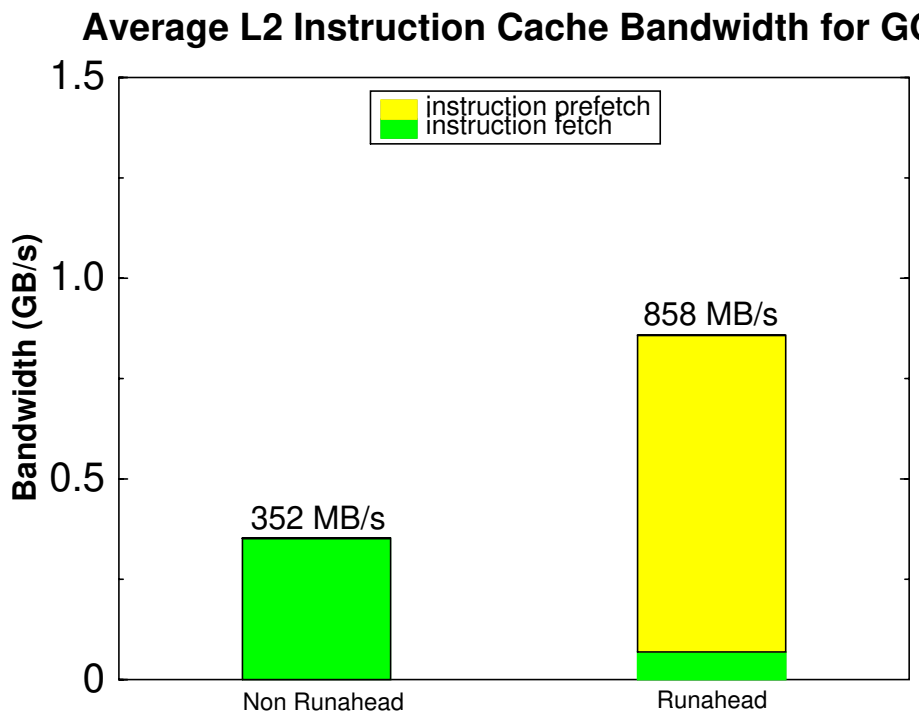


Figure 4.17

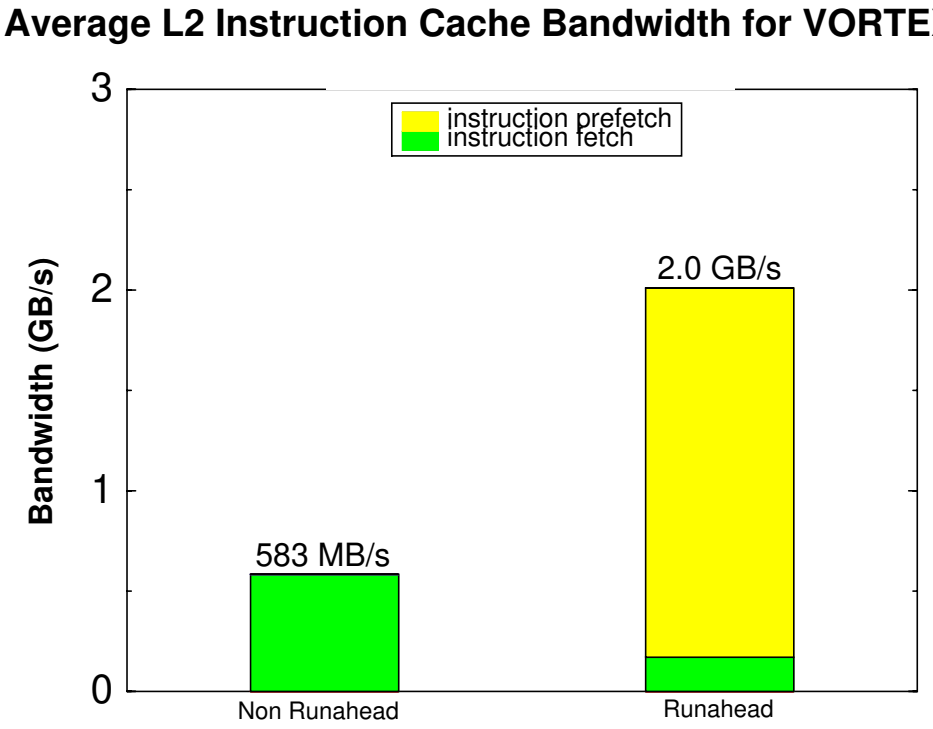


Figure 4.18

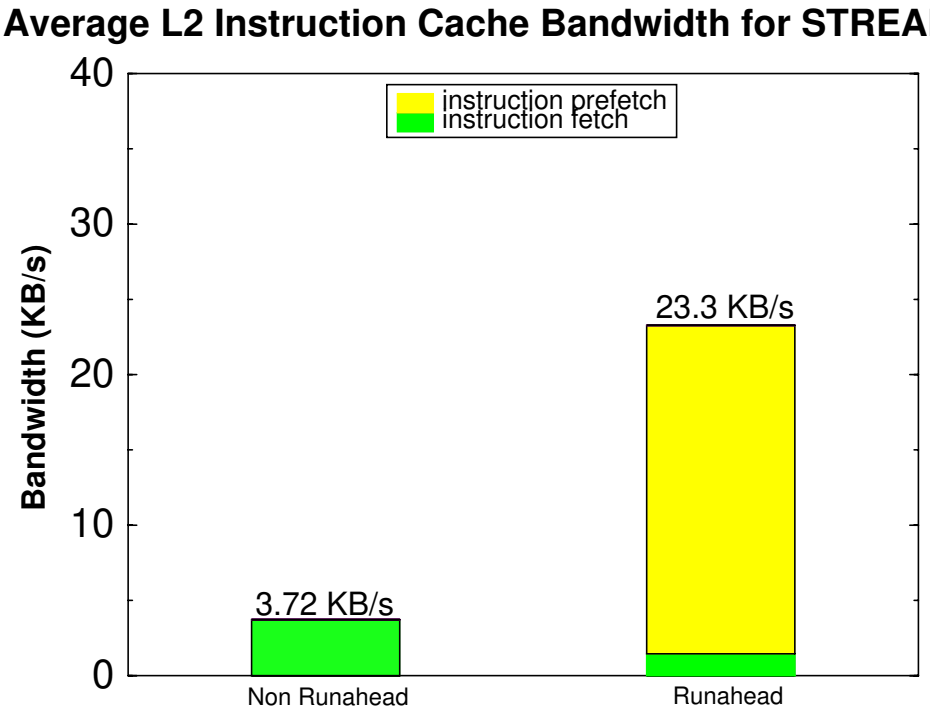
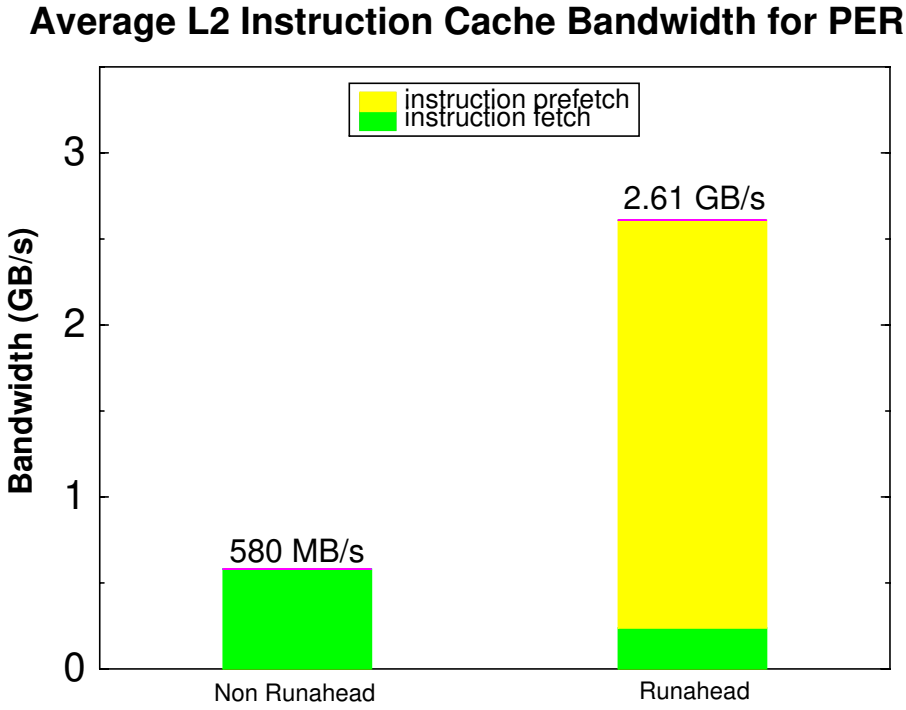


Figure 4.19



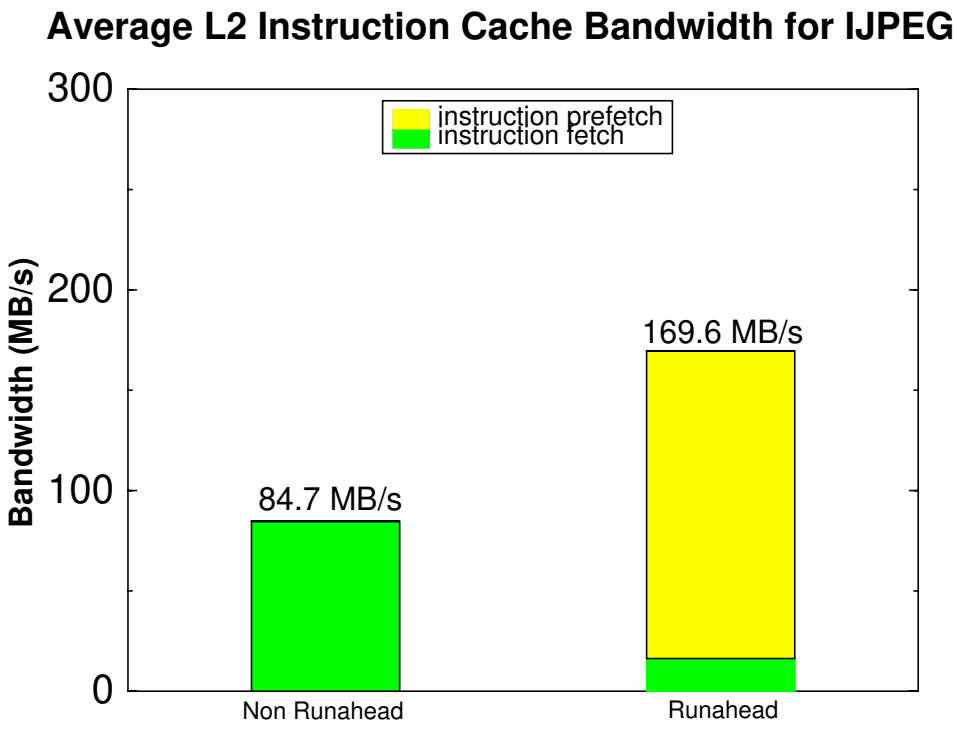


Figure 4.20

4.3 Prefetching effectiveness over the course of runahead episodes

As these runahead simulations allow the processor to speculatively pre-process instructions past branches that cannot be resolved, there will likely be a significant number of instructions that are pre-processed on wrong paths. These instructions can generate erroneous prefetches, even if their register values are VALID.

4.3.1 Probability of remaining on the correct path during runahead

An instruction is on the proper path in a runahead episode if all of the instructions in the runahead episode up to and including the instruction in question are executed during normal operation at the conclusion of runahead, starting with the first instruction in the runahead episode. Also, note that it is possible for the processor to re-enter runahead mode multiple

times before all of the instructions in a given runahead episode are executed during normal operation.

The x-axis of each plot represents the distance into the average runahead episode, in instructions, with the x = 0 point indicating the instructions that initiate runahead mode. The y-axis represents the probability that an instruction at a given point in the runahead episode is on the right path. Measuring this was complicated somewhat by the fact that the processor can execute the same dynamic instance of an instruction multiple times in successive runahead episodes before the instruction is actually executed, if ever, during normal operation. Note that instructions skipped due to instruction cache misses are still considered to have been pre-processed. The PCs of these skipped instructions are compared to those executed when normal operation resumes in the same manner that instructions that are actually pre-processed are. Each plot has three data sets, representing averages for load-, store-, and instruction-miss initiated runahead episodes. We do not present STREAM results as the simple nature of this benchmark virtually guarantees that the processor remains on the correct path during load- and store-miss initiated runahead episodes.

The first plot, shown in Figure 4.21, is for the GO benchmark. Note that the first two instructions in both load and store-miss initiated episodes are always on the correct path. The first instruction in a data stream runahead episode is always, by definition, a load or store instruction. These instructions cannot of course redirect the instruction stream. The second instruction in these episodes can be a conditional branch, which can of course be mispredicted. Instructions after the branch can be on the wrong path. The store-miss initiated runahead episodes are by far the most likely to stay on the right path during the average runahead episode. This is a consequence of their being initiated by store instructions, which

do not introduce INV values into the processor register file. Any INV registers can of course cause a branch to be mispredicted. In fact, for the GO Benchmark there is about an 80% probability that a store-miss initiated episode is on the right path after 25 instructions. The effect of INV registers can be seen in the data set for load-miss initiated episodes. There is an immediate drop off in the probability after the first few instructions into the average load-miss initiated episode. Even so, there is still about a 50% probability of runahead remaining on the right path after 25 instructions. The instruction-miss initiated runahead data set is interesting as it starts out at the x = 0 position with about a 75% probability of being on the right path. This is a consequence of the fact that instruction miss initiated episodes are initiated in the FETCH stage, where they can get squashed by instructions farther down the pipeline. These brief instruction-miss initiated episodes still count however, because they can generate a large number of erroneous prefetches. Each instruction stream prefetch can generate up to 8 skipped instructions. Instruction miss initiated episodes are also hampered as they skip instructions, some of which may have redirected the fetch stream. This is apparent in the data set for the GO Benchmark, where the average instruction miss initiated episode only has about a 15% chance of being on the right path after 25 instructions have been pre-processed.

The second plot, shown in Figure 4.22, is for the VORTEX benchmark. This plot appears rather similar to the GO plot shown in Figure 4.21. As before, store-miss initiated episodes are significantly more likely to stay on the right path during the average runahead episode, with about a 93% probability after 25 instructions. The load-miss initiated data set doesn't drop off quite as fast as that for the GO benchmark. Even so, there is still a greater than 50% probability of load-miss initiated episodes being on the right path after 25 instruc-

tions. The instruction miss initiated data set is exhibits worse performance than that for GO, with only about a 10% chance of being on the right path after 25 instructions have been pre-processed.

The results for the PERL benchmark are shown in Figure 4.23. As was the case for the VORTEX benchmark, this plot appears similar to that obtained for the GO benchmark.

The data obtained for the IJPEG benchmark, shown in Figure 4.24, is particularly interesting. Note that both load and store-miss initiated episodes have a very high probability of being on the right path, even after 25 instructions have been pre-processed. This is a result of the relatively few branches in the benchmark that are dependent upon load-miss data. Note that this result confirms our preliminary results reported in [36] where we found that the aggressive and conservative runahead models reported virtually identical performance improvements for IJPEG. IJPEG instruction miss initiated episodes also perform comparatively better than the other benchmarks, with a nearly 50% probability of being on the right path after 25 instructions have been pre-processed.

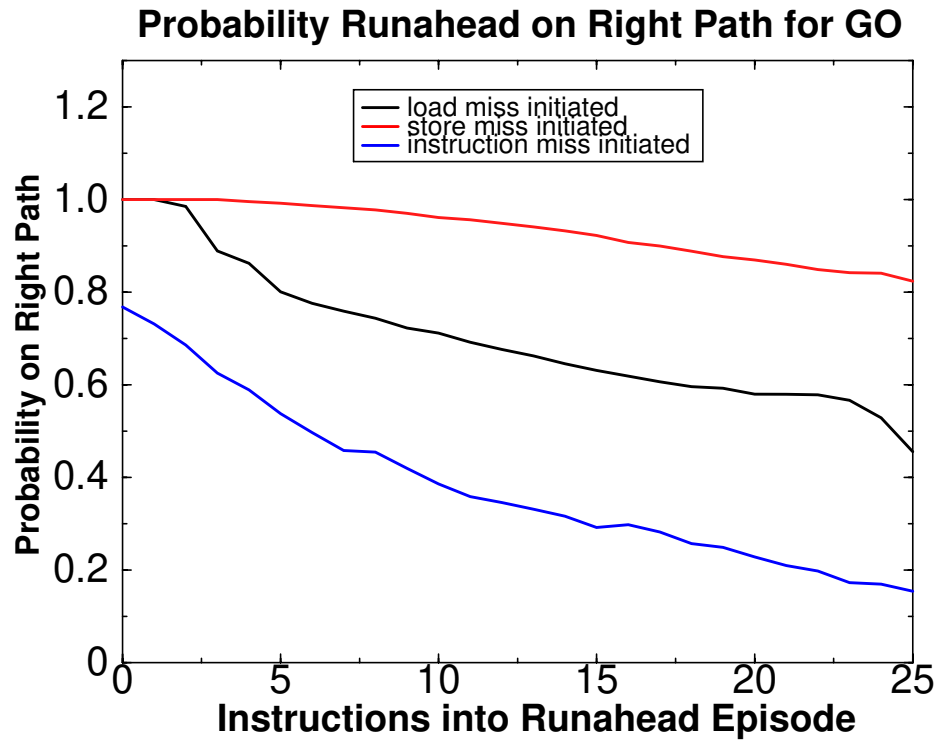


Figure 4.21

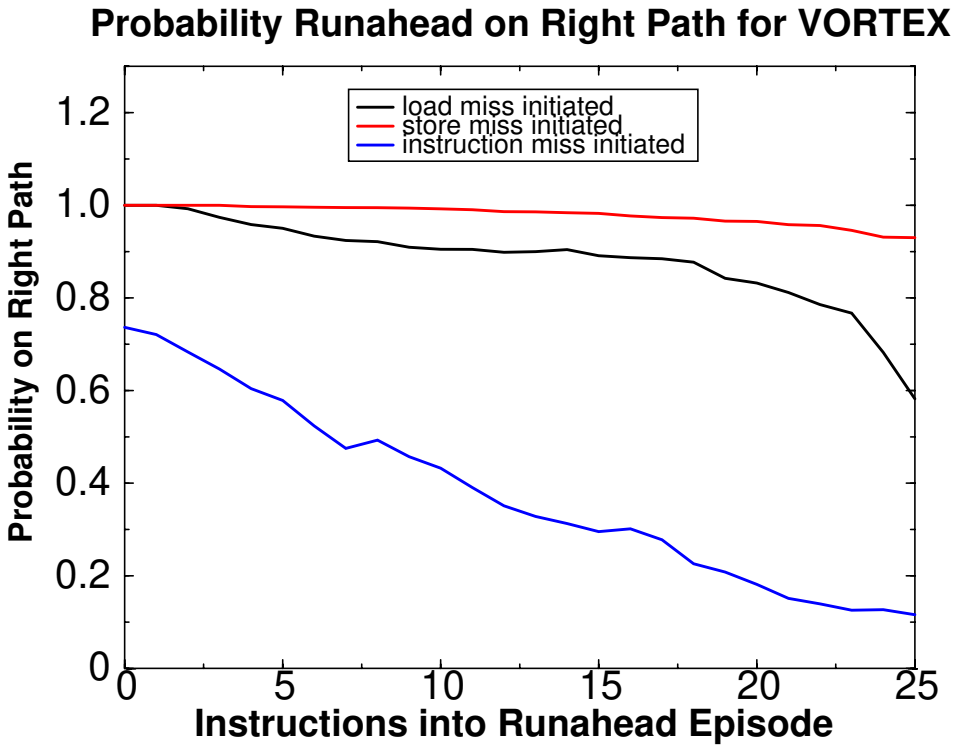


Figure 4.22

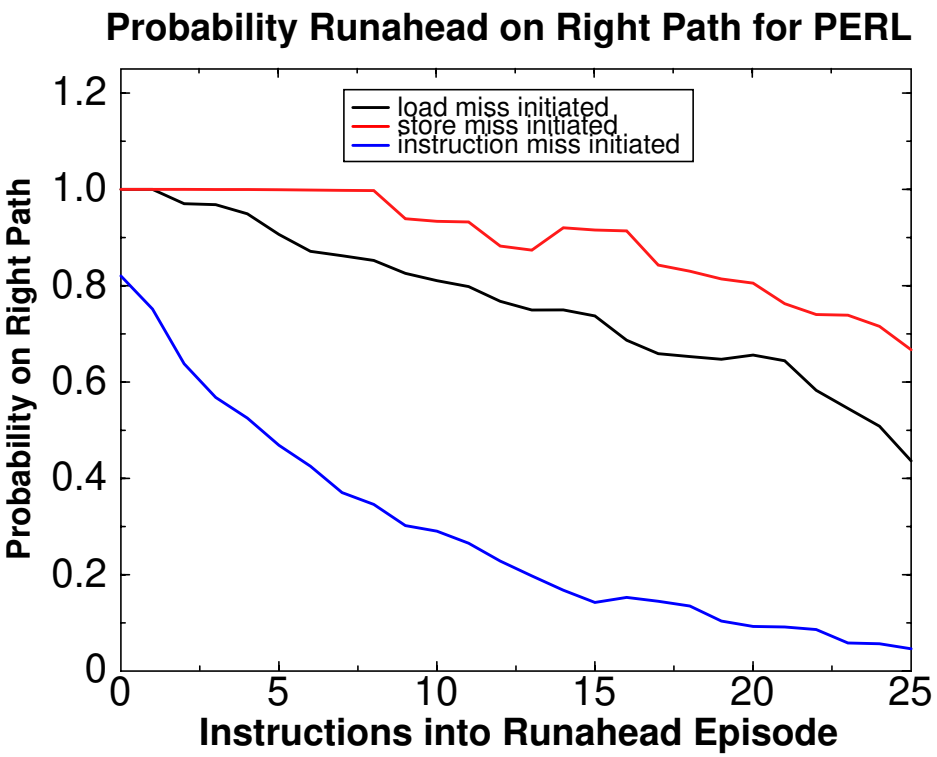


Figure 4.23

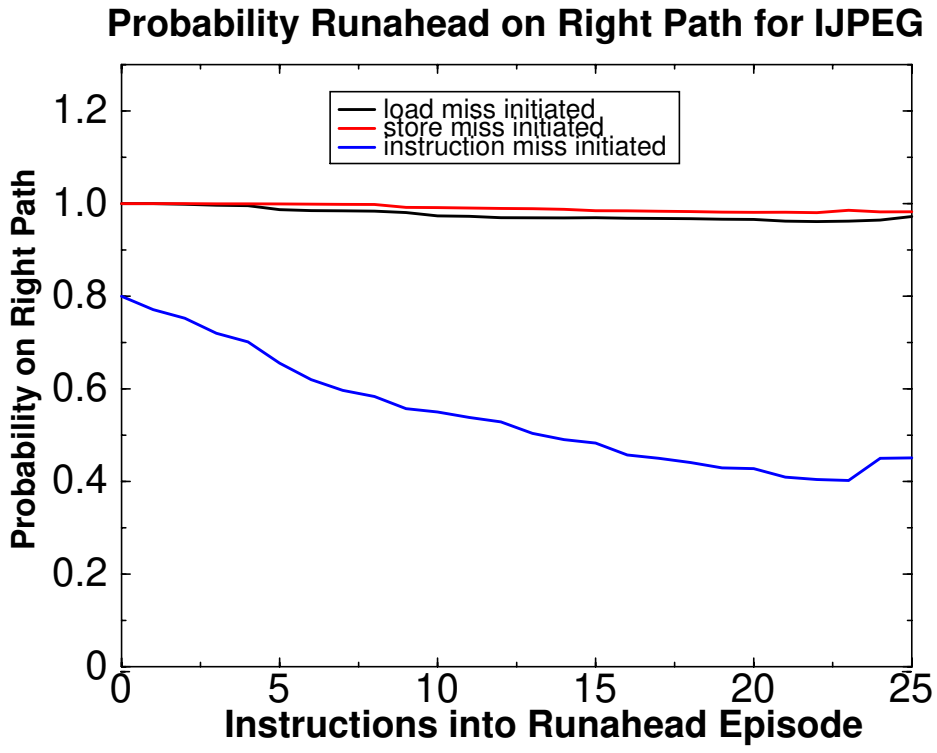


Figure 4.24

4.3.2 Number of runahead episodes of each type

The total number of runahead episodes that are initiated for each type of runahead is shown in Figure 4.25. Note that load-miss initiated episodes outnumber store-miss episodes: this was expected as there are more load than store instructions executed in each benchmark. Recall that the STREAM benchmark was simulated for 10M instructions, while the SPEC benchmarks were simulated for 100M instructions.

4.3.3 Average number of prefetches generated per runahead episode

The average number of load, store, and instruction prefetches generated during each type of runahead episode are shown in Figure 4.26. Note that store-miss initiated runahead episodes always generate more data stream prefetches per episode than either load- or instruction-miss episodes. This is a consequence of store misses not immediately seeding the RF with INV values at the initiation of runahead. Interestingly, STREAM generates the least data stream prefetches per episode of all of the benchmarks, yet it derives the most benefit from runahead. This is a result of its high miss rate, and correspondingly larger number of runahead episodes. This can be seen in Figure 4.25. While STREAM generates the most instruction prefetches on average, all of them are generated during instruction-miss initiated runahead episodes, of which there are very few due to the extremely low instruction cache miss rate.

Figure 4.25

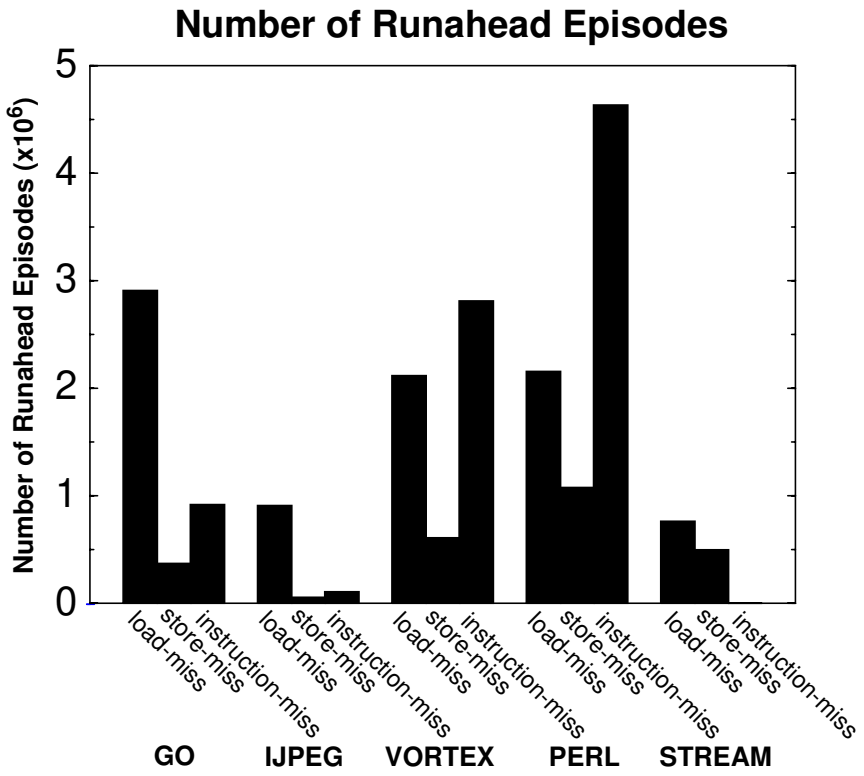
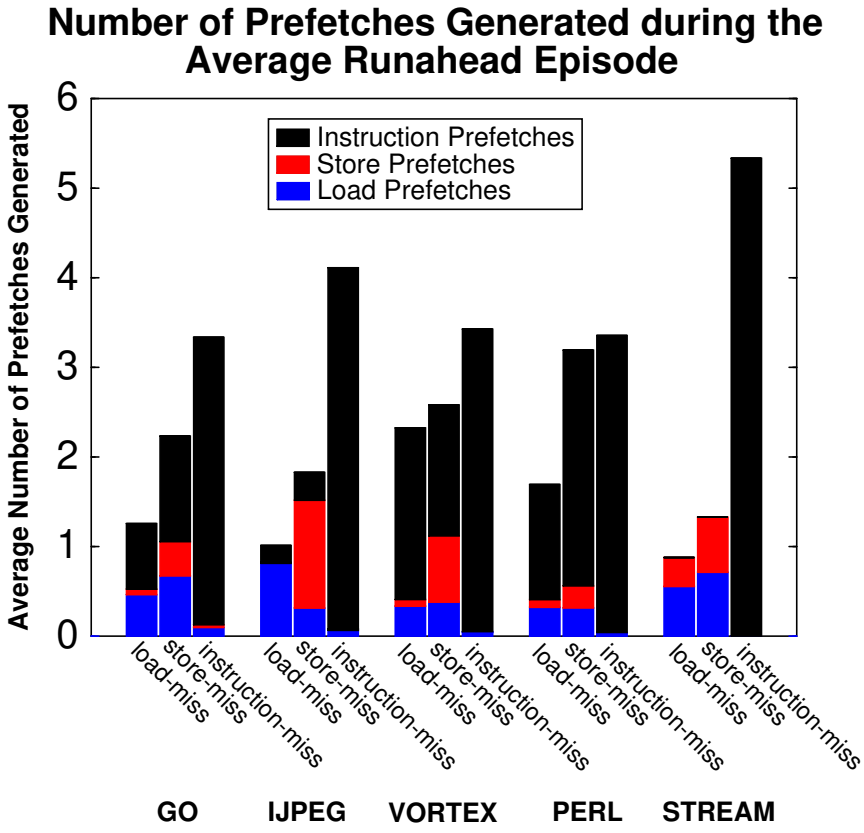


Figure 4.26



4.3.4 Prefetching utility over the course of runahead episodes

We were particularly interested in finding out how effective runahead was over the course of the average episode. We define a useful runahead prefetch as a prefetched line that is accessed at least once during normal operation before it is either ejected from the L1 instruction or data cache, or the simulation ends. Accesses that occur during runahead do not count, as these can be on the wrong path, and at any rate do not directly perform useful work. Useless prefetches are of course the opposite of useful prefetches: they are never accessed during normal operation before they are ejected from the L1 instruction or data cache, or the simulation ends.

Having defined what we mean by useful and useless prefetches, we can now describe the plots that we will use to illustrate the utility of runahead over the course of the average episode. We want to be able to illustrate the number of useful vs. useless prefetches that have been generated as a function of the number of instructions that have been pre-processed. Note that these plots are cumulative as we want to show the utility of runahead up to a certain number of instructions into the average episode. Due to the different nature of load, store, and instruction miss initiated runahead episodes, we have chosen to present separate plots for each runahead episode type. These plots are shown in Figures 4.27 through 4.66. Each plot contains 4 data sets, which allow us to break the useful and useless prefetches down into prefetches generated on the right and wrong paths. Finally, as noted earlier the average runahead episode is shorter than 100 instructions in length. Because of this we limit the x axis to a maximum of 100 instructions. Prefetches generated past this point are taken care of by recording the total number of prefetches as spikes at x = 100.

Load-miss initiated runahead

The first set of plots, shown in Figures 4.27 through 4.29 is for the GO benchmark. The first of these, shown in Figure 4.27, is for load prefetches generated during load-miss initiated runahead episodes. Note that the majority of the prefetches fall into the useful/right-path category, and that virtually all prefetches generated during the first five or so instructions into runahead fall into this category. After about five instructions significant numbers of wrong path prefetches start to be generated. However, note that most of these are still useful prefetches. Very few prefetches are generated past the 25th instruction into the average runahead episode, as indicated by the flat curves past that point. This indicates that the overwhelming majority of the runahead-initiating misses hit in the L2 data cache. At that point the overwhelming majority of prefetches are useful, and of these most are on the right path. Useless/wrong-path prefetches make up less than 10% of the total, while very few useless/right-path prefetches are generated. Note that most useless/right-path prefetches should have been useful, however they were ejected from the cache by other prefetches before they could be accessed during normal operation. Figure 4.28 shows the breakdown for store prefetches generated during load-miss initiated runahead episodes for GO. Note that this plot is very similar to the load prefetch plot, the only noteworthy difference being that load-miss initiated episodes tend to generate many more load prefetches than store prefetches for the GO benchmark. Figure 4.29 shows the breakdown for instruction stream prefetches. Note that both right- and wrong path useful prefetches predominate for the first 99 instructions of the average episode, and that nearly all prefetches fall into either useful prefetch category during the first twenty or so instructions into the average episode. The number of useful/right-path and useless/right-path prefetches spikes at 100 instructions in to the average episode. As was mentioned earlier, the points at x = 100 represent the total number of

prefetches generated. These spikes are a result of a succession of instruction cache misses that occur late during runahead episodes that last much longer than average. The plots for the VORTEX, PERL, and IJPEG benchmarks are very similar to those obtained for GO, and can be found in Figures 4.30 through 4.38.

The plots for the STREAM benchmark, shown in Figures 4.39 and 4.40, are particularly interesting. Note that all of the prefetches fall into the useful/right-path category. The access pattern of STREAM is such that prefetched lines are not ejected until the data stream "wraps around" the cache. As this takes quite a few cycles, even for a relatively small 8KB cache, virtually all prefetched lines are accessed during normal operation before they are ejected. Also, STREAM does not exhibit the "knees" that the other benchmarks have: virtually all L1 data cache misses for STREAM have to go out to main memory, which pushes the average runahead episode length out to close to 100 instructions. We do not present

instruction prefetch results for STREAM due to its extremely low instruction cache miss rate.

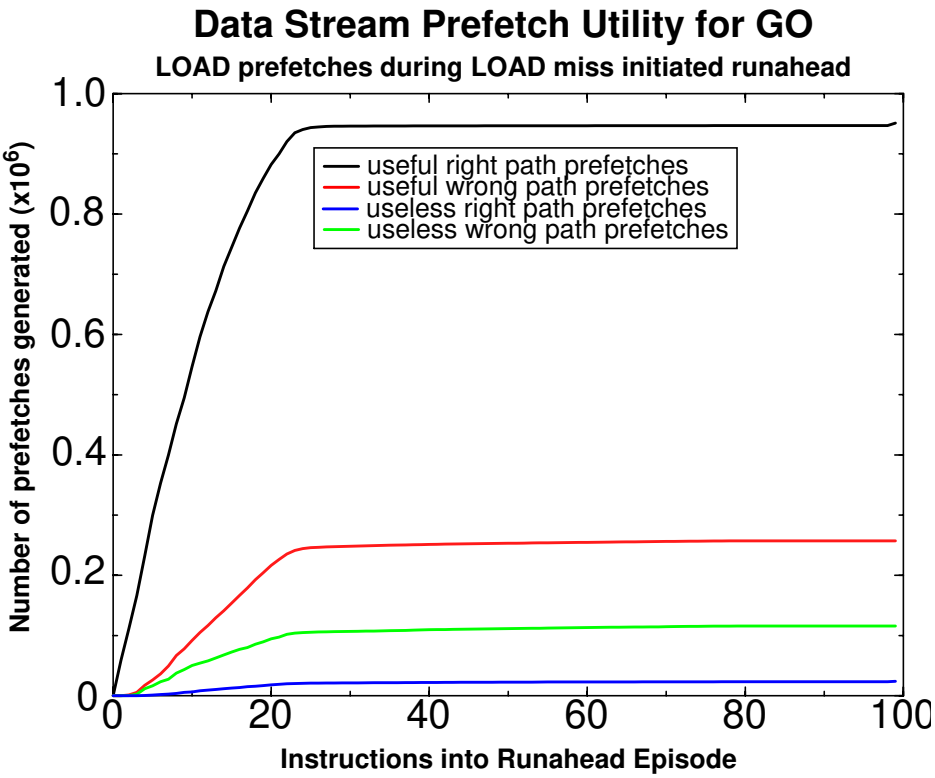


Figure 4.27

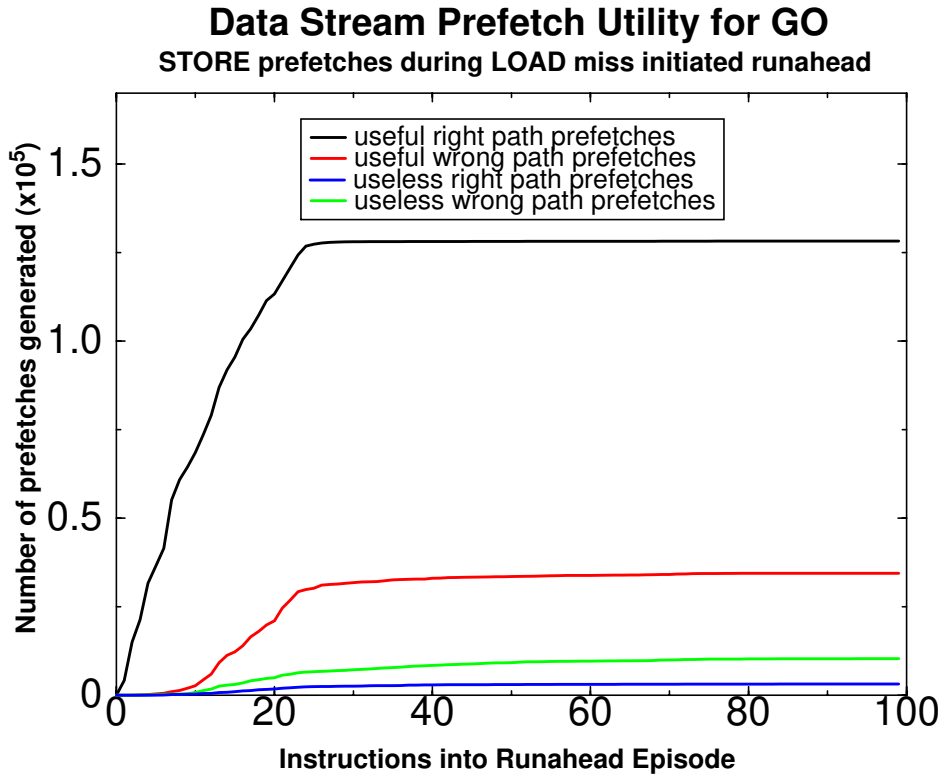


Figure 4.28

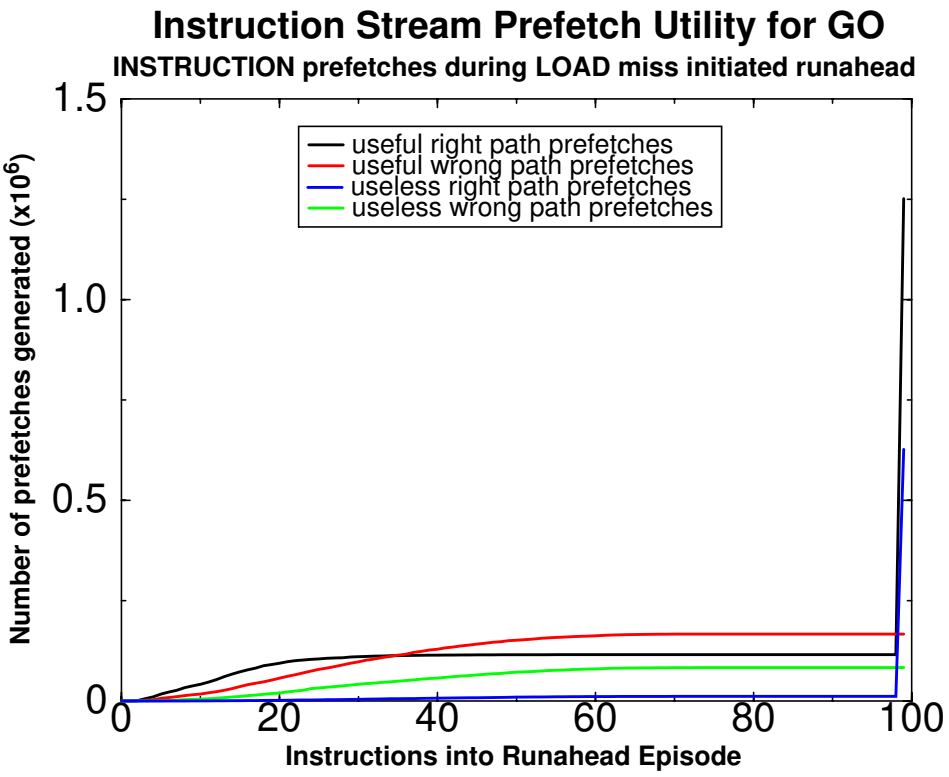


Figure 4.29

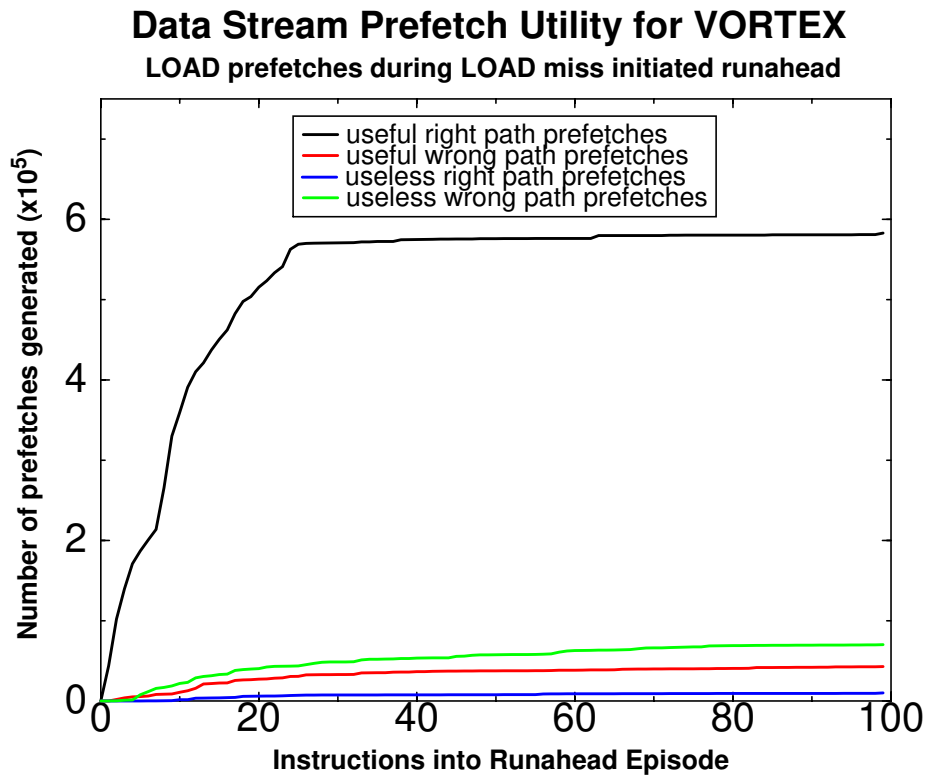


Figure 4.30

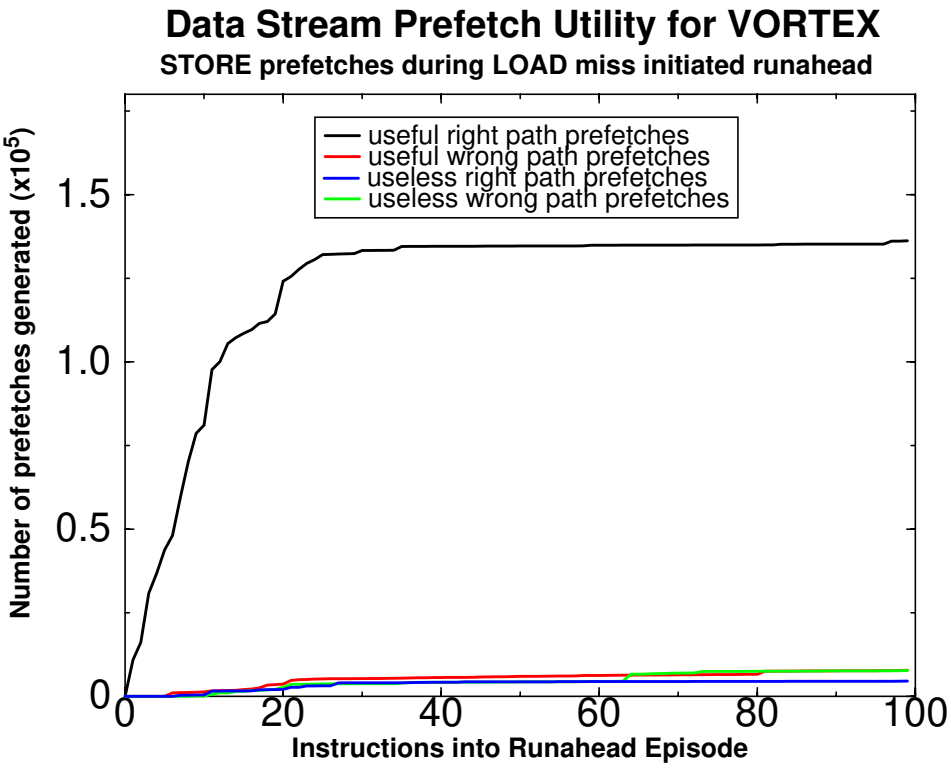


Figure 4.31

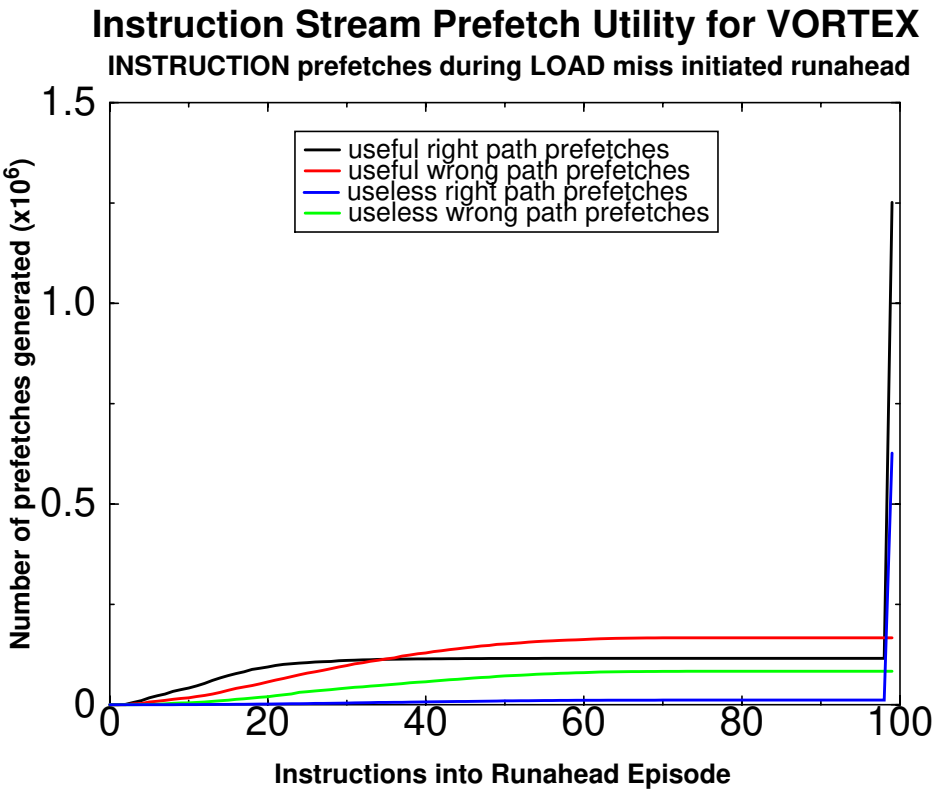


Figure 4.32

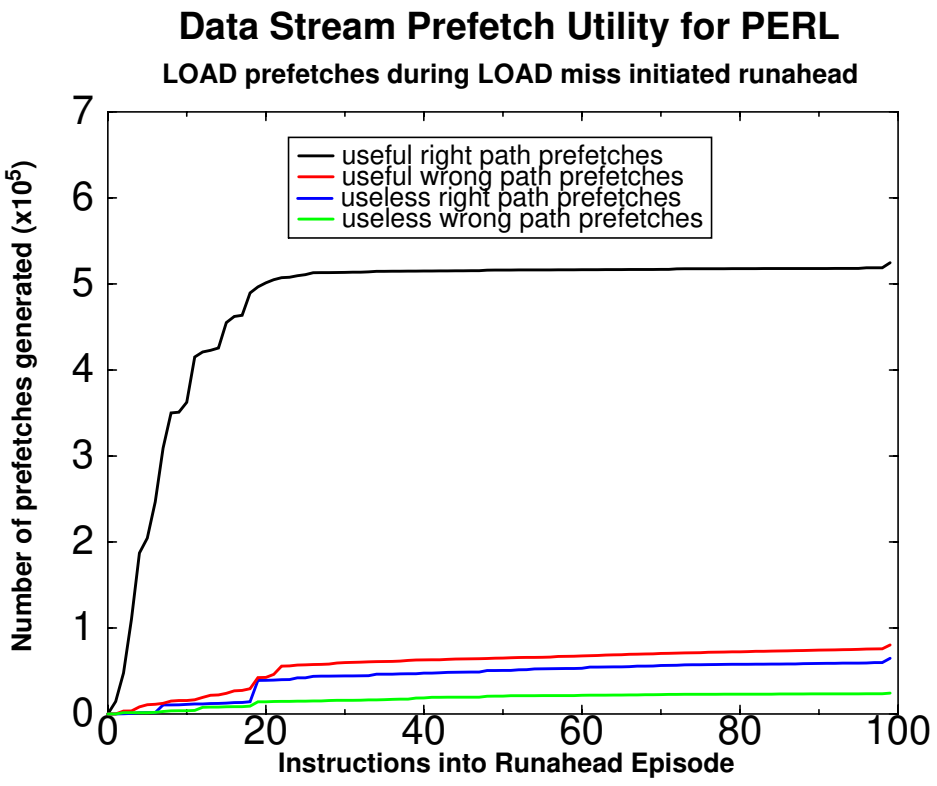


Figure 4.33

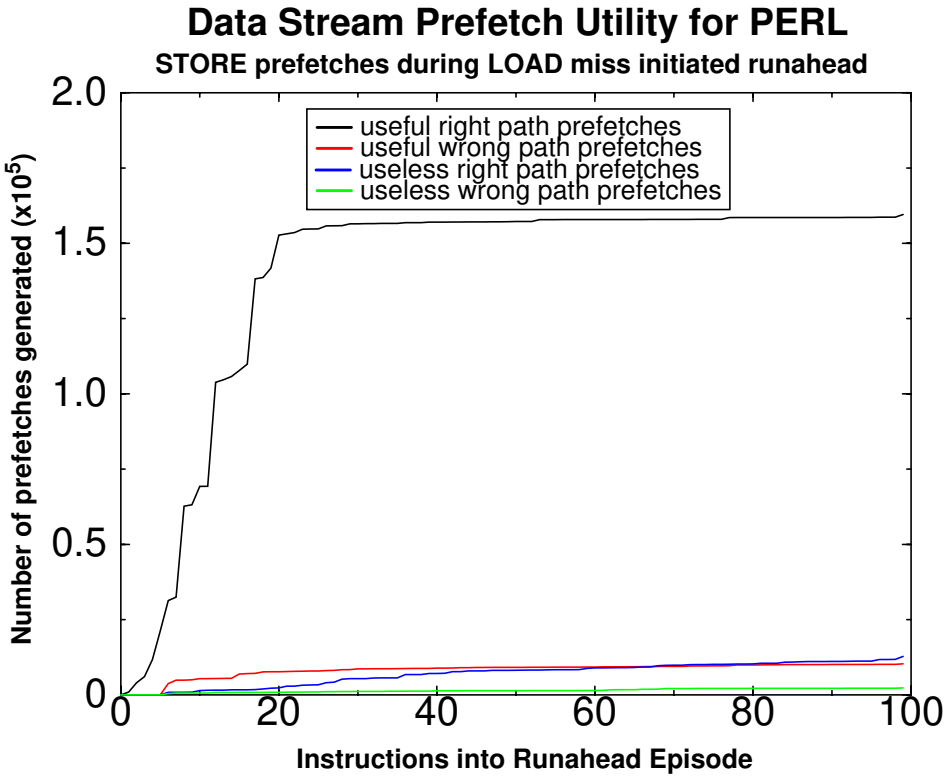


Figure 4.34

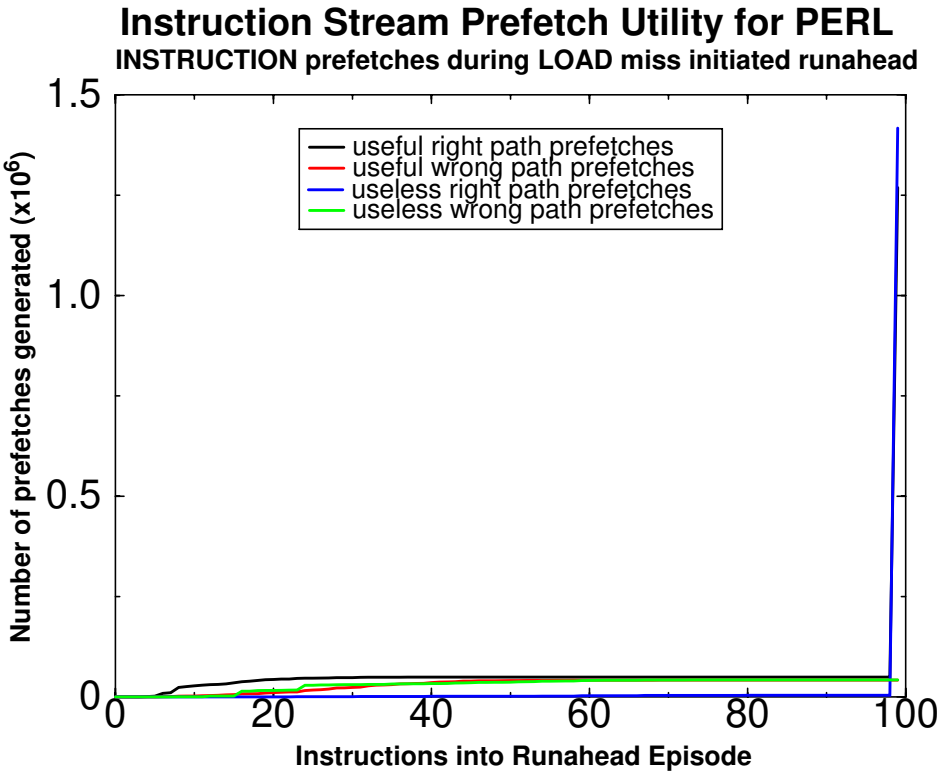


Figure 4.35

Figure 4.36

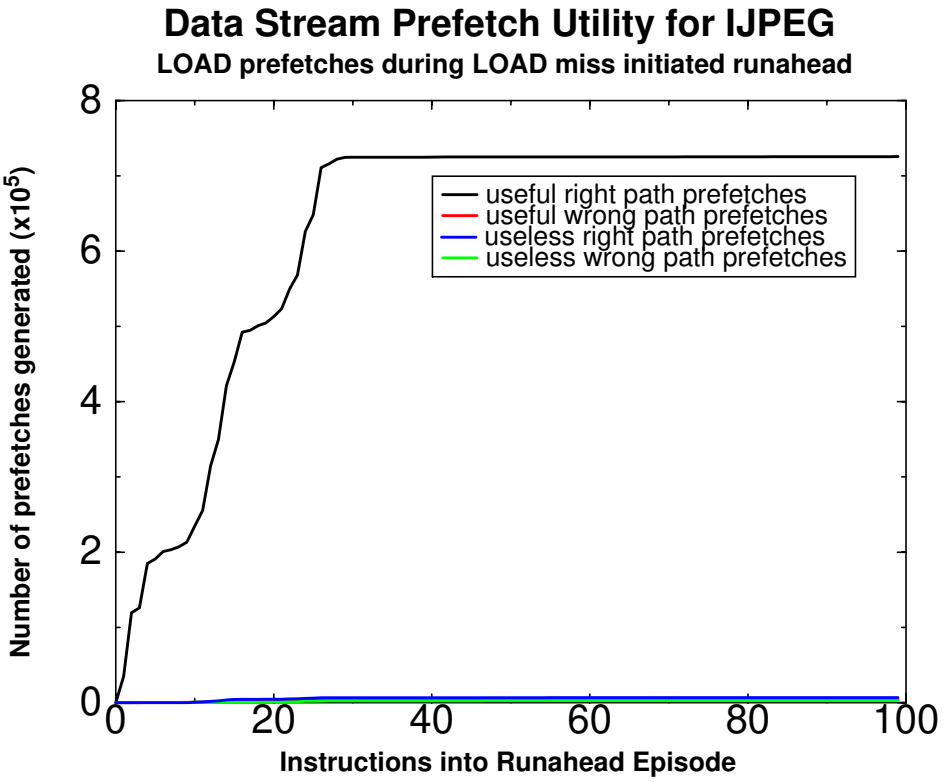
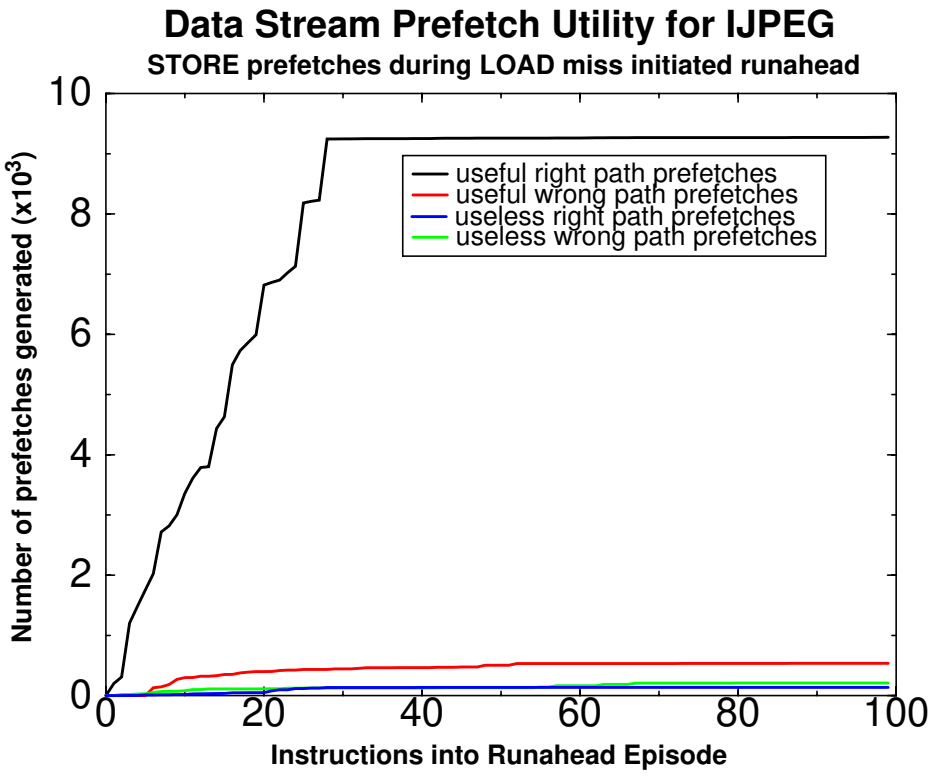


Figure 4.37



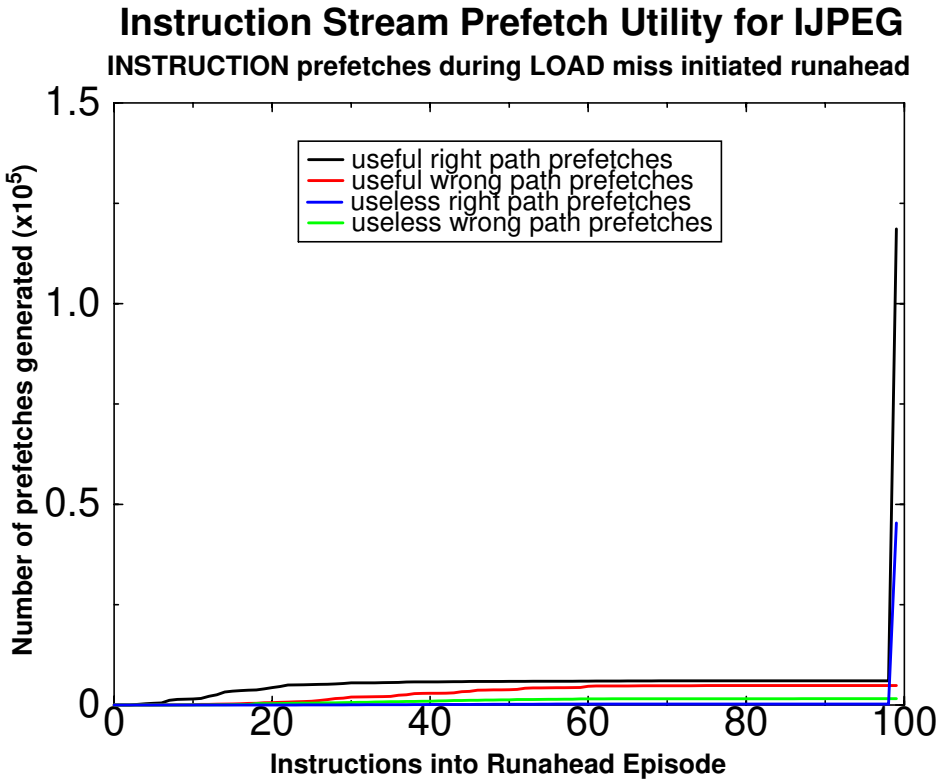


Figure 4.38

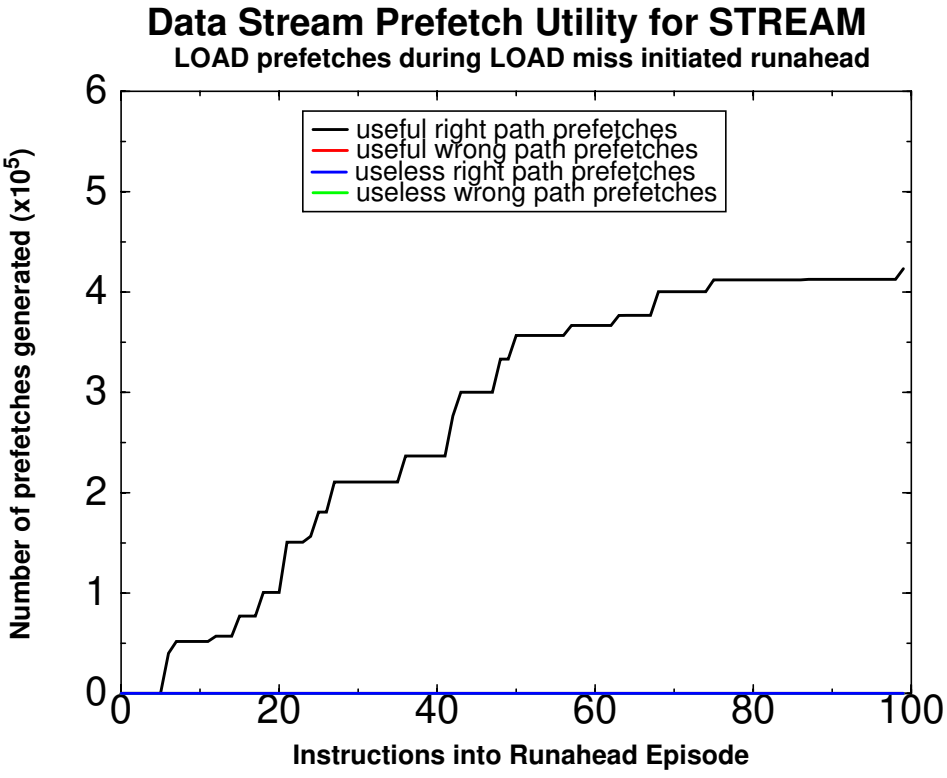
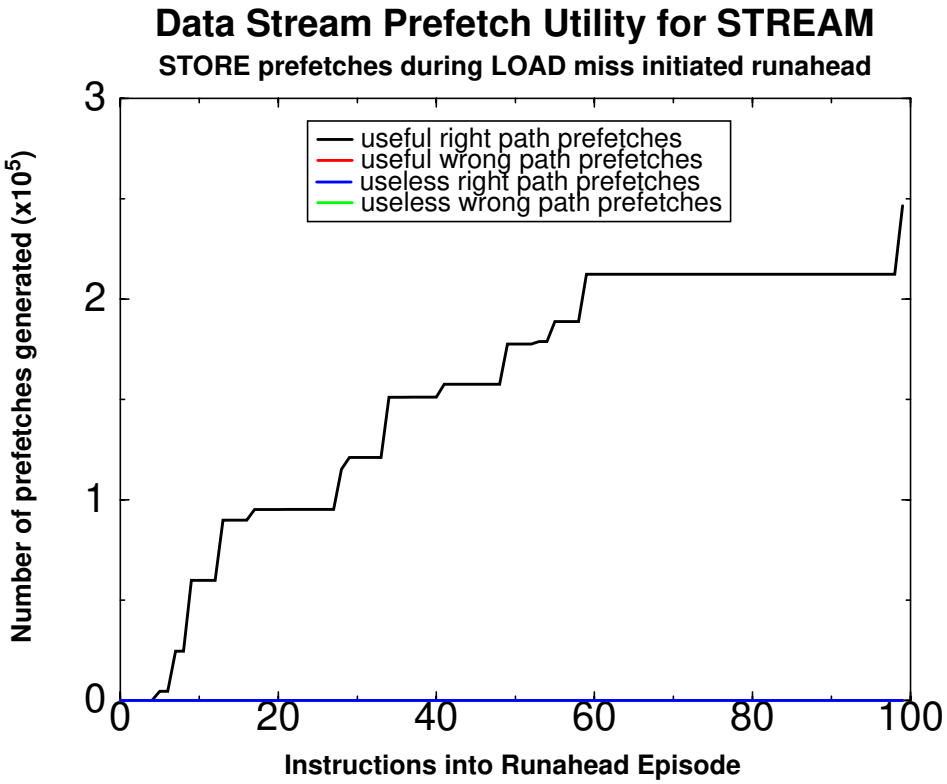


Figure 4.39

Figure 4.40



Store-miss initiated runahead

Store-miss initiated runahead episodes are more promising as they do not add an INV value to the register file at the initiation of runahead. Unfortunately, loads tend to outnumber stores, which will result in more load than store miss initiated runahead episodes.

The first set of plots, shown in Figures 4.41 through 4.43, is for the GO benchmark. The first of these plots, shown in Figure 4.41, is for load prefetches. Note that virtually all of the load prefetches generated are useful/right-path prefetches. Very few useless prefetches are generated, and of these, useless/right-path prefetches predominate. A similar plot for store prefetches is shown in Figure 4.42. The plot for instruction prefetches generated during store-miss initiated episodes, shown in Figure 4.43, is very similar to that for instruction prefetches generated during load-miss initiated runahead episodes, shown in Figure 4.29. The only notable difference is that useful/right-path prefetches predominate over the entire

course of the average runahead episode. This is due to the greater probability of staying on the right path for store miss initiated runahead episodes, as can be seen in Figure 4.21. The plots for the VORTEX, PERL, and IJPEG benchmarks are very similar to those obtained for GO, and can be found in Figures 4.44 through 4.52. Note that the plots for STREAM shown in Figures 4.53 and 4.54 are very similar to their load-miss initiated counterparts in Figures 4.39 and 4.40.

Figure 4.41

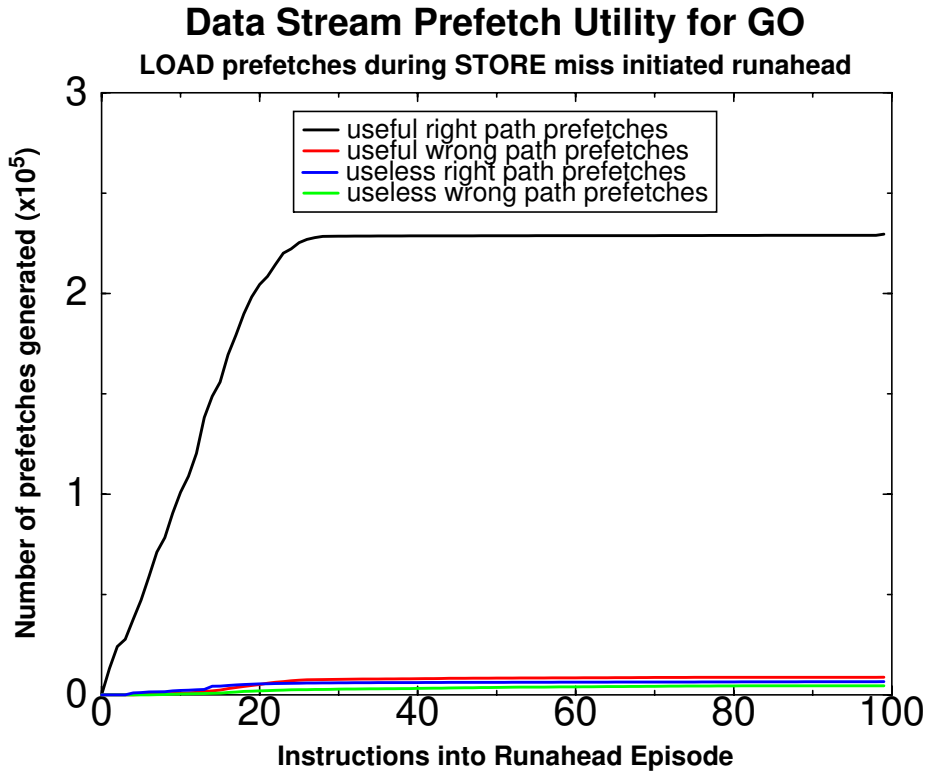
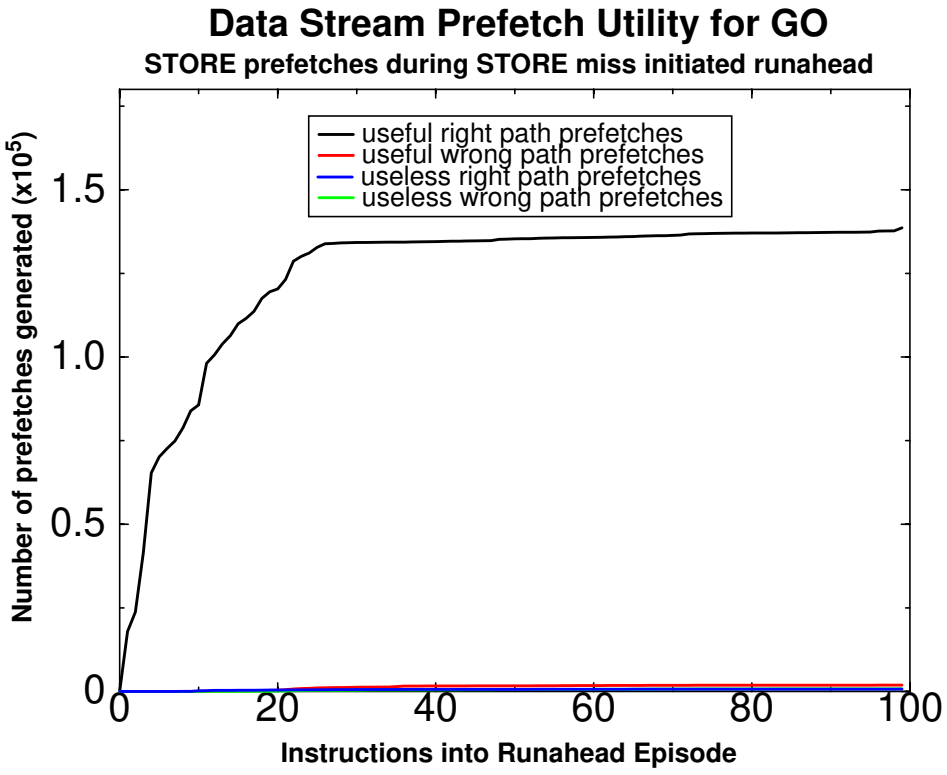


Figure 4.42



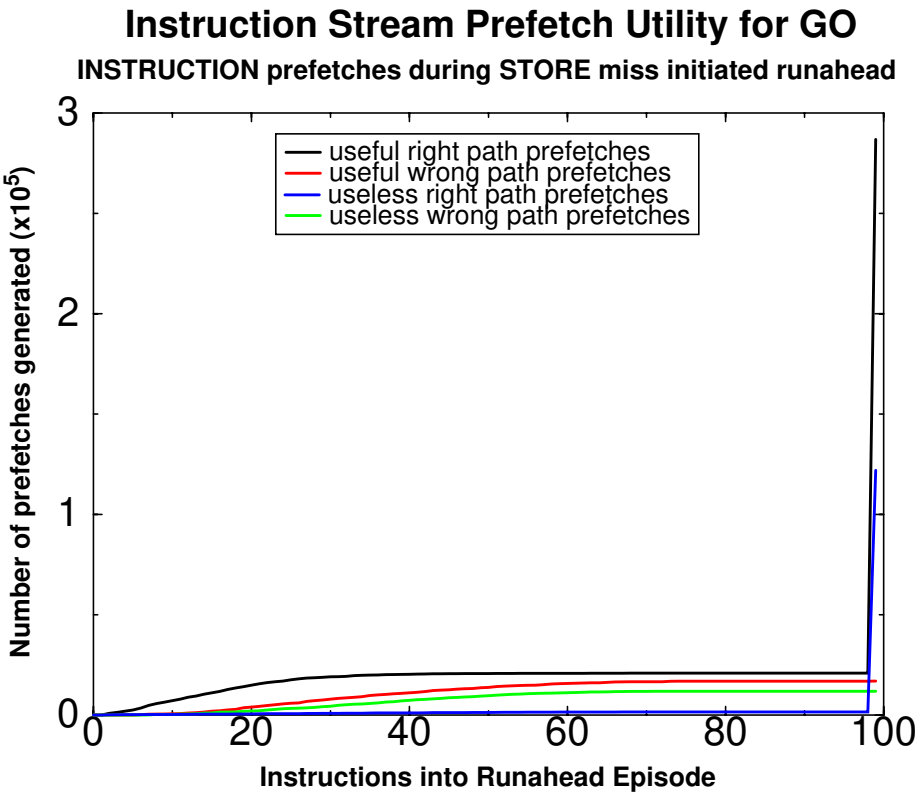


Figure 4.43

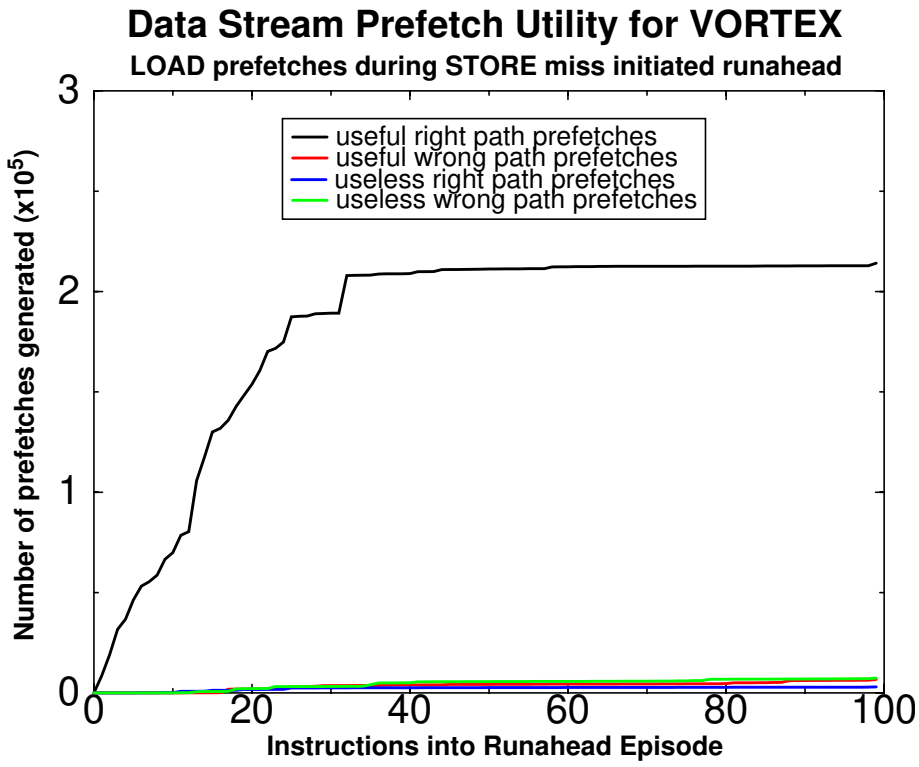


Figure 4.44

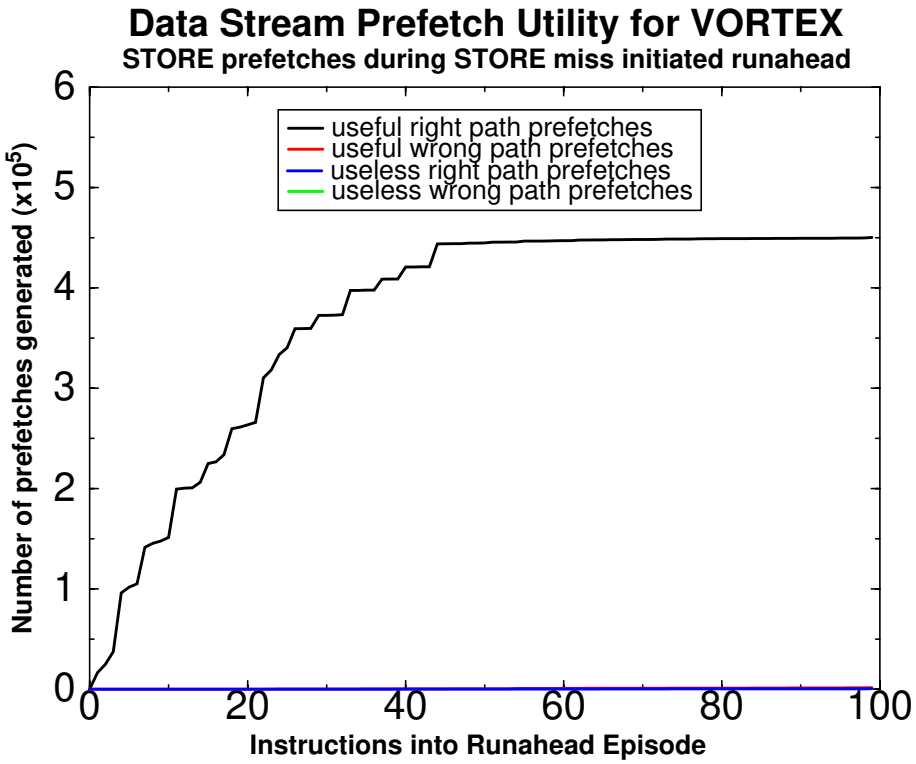


Figure 4.45

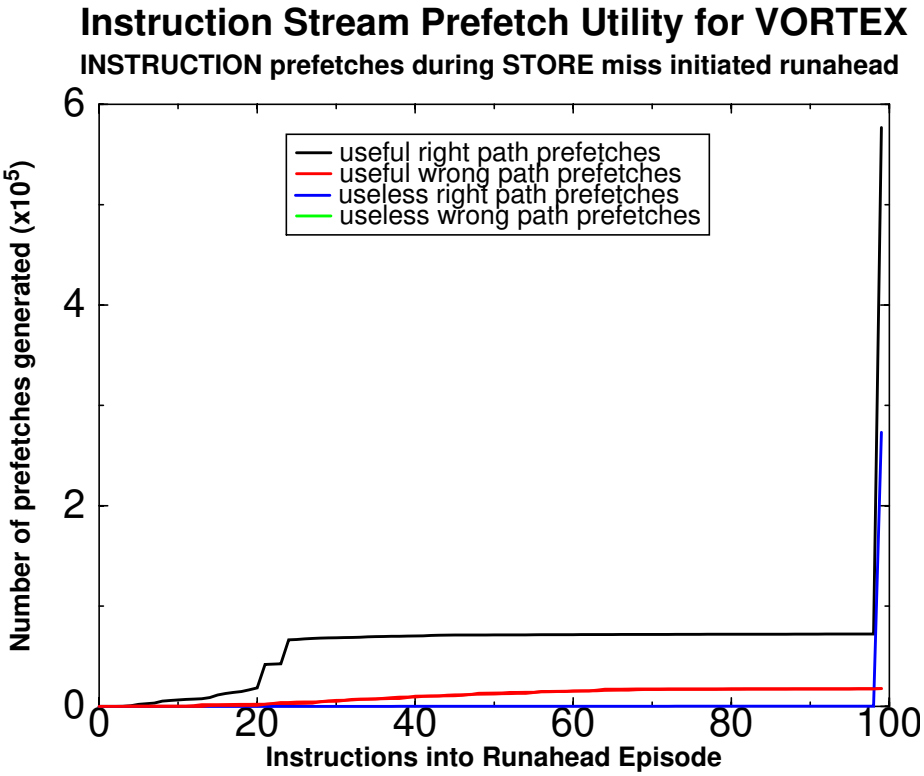


Figure 4.46

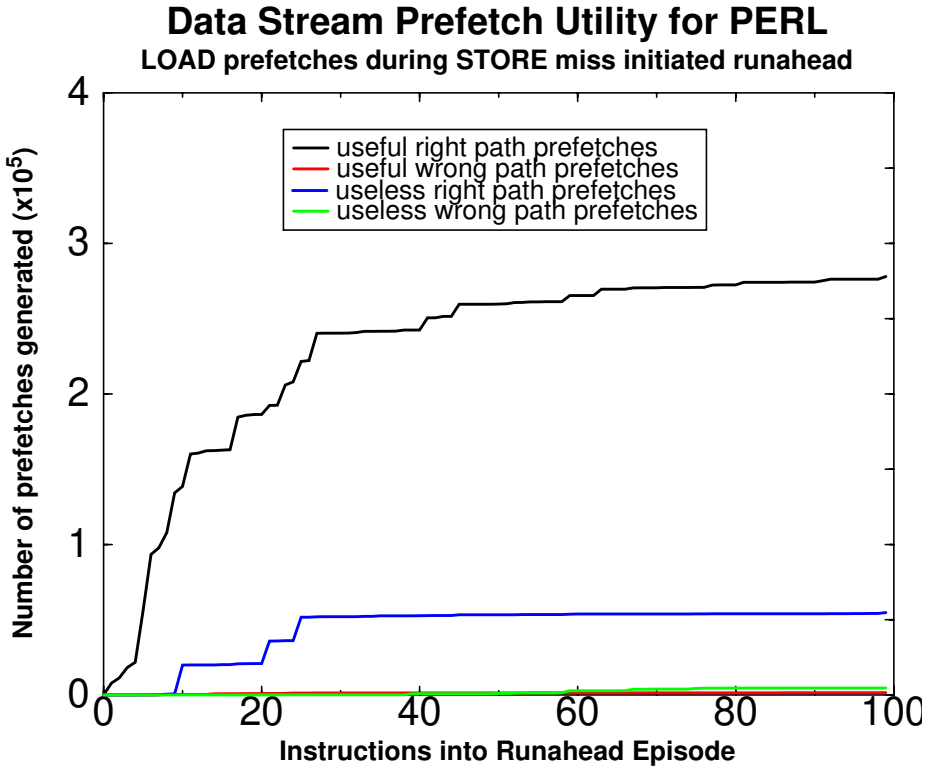


Figure 4.47

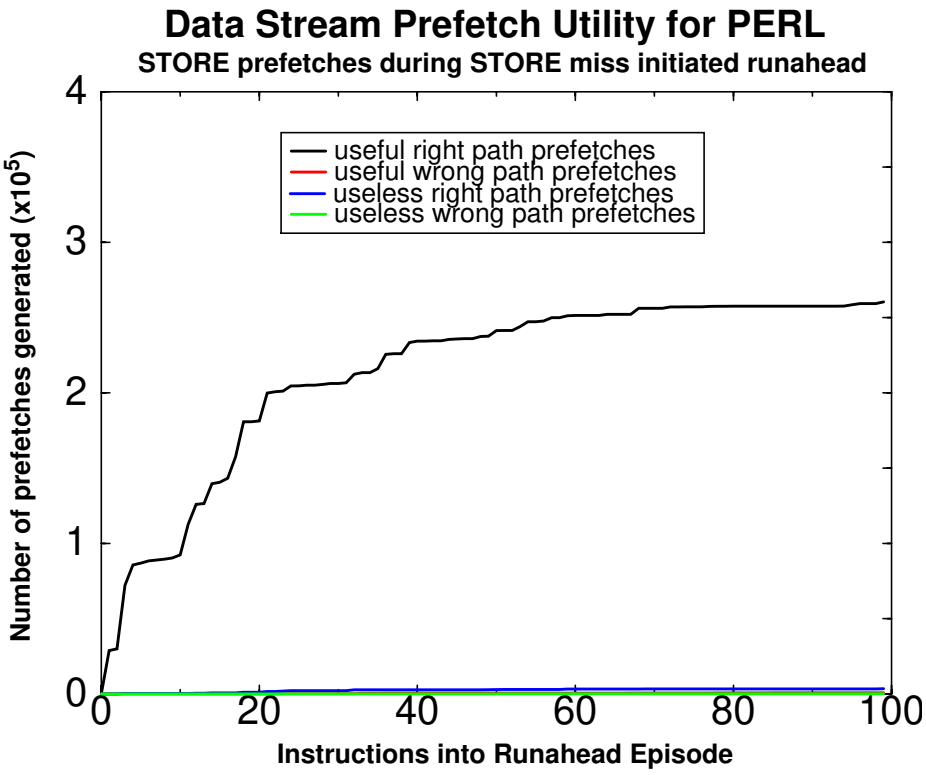


Figure 4.48

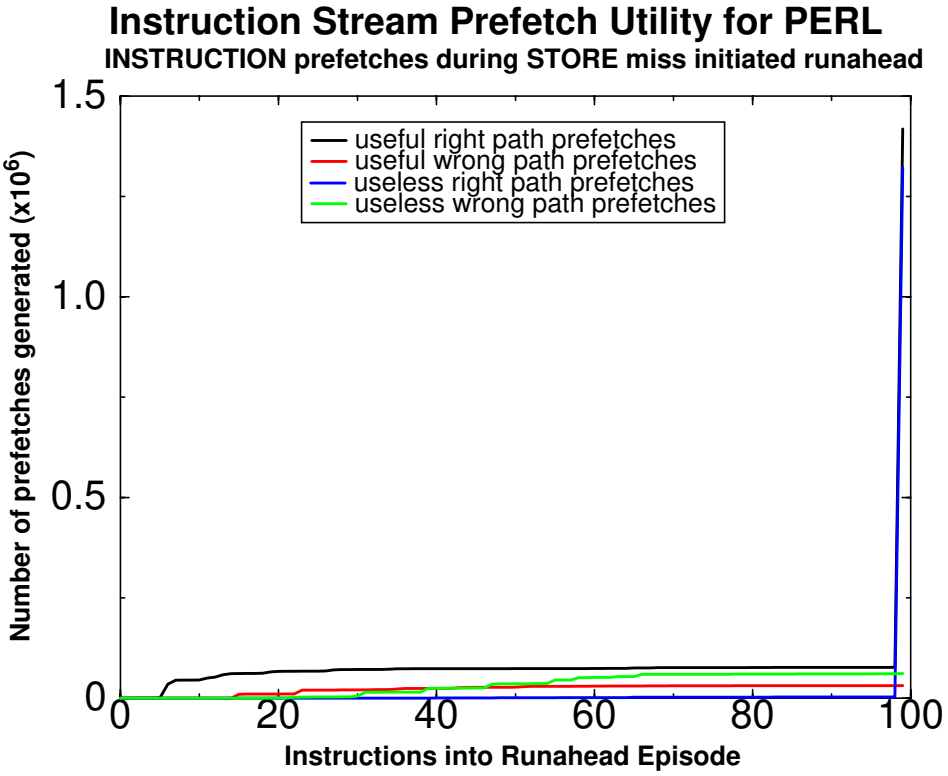


Figure 4.49

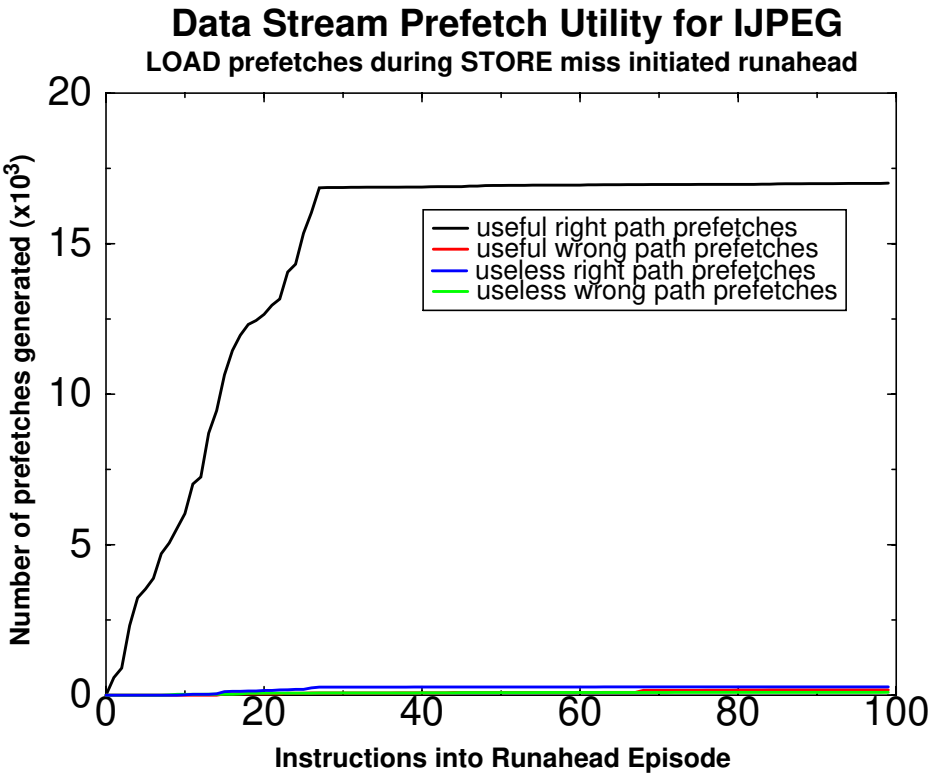


Figure 4.50

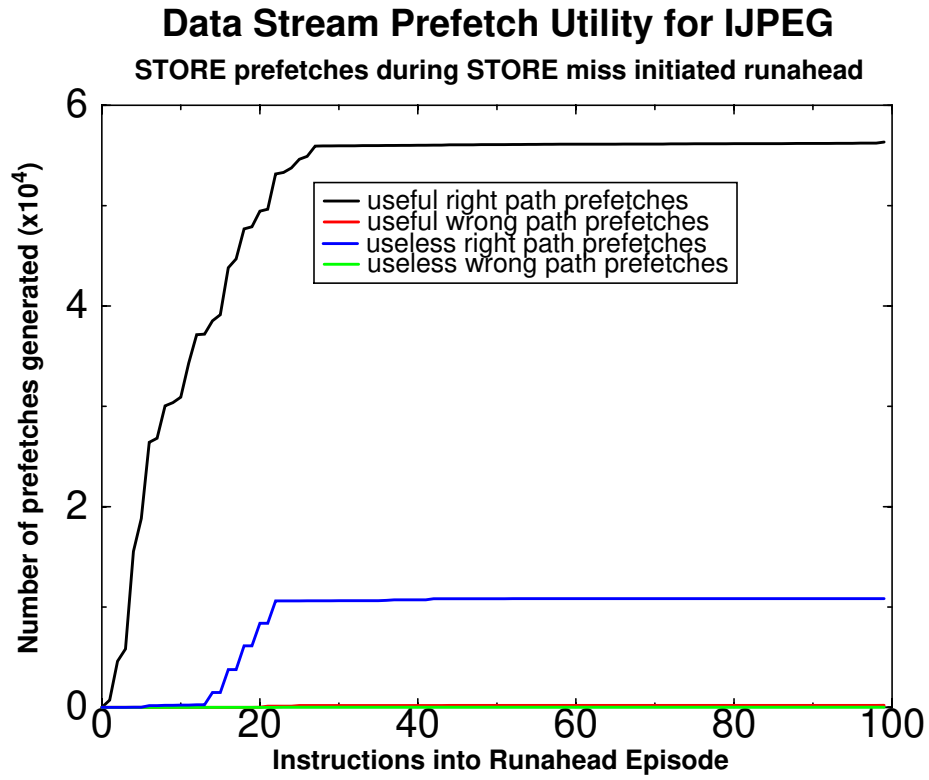


Figure 4.51

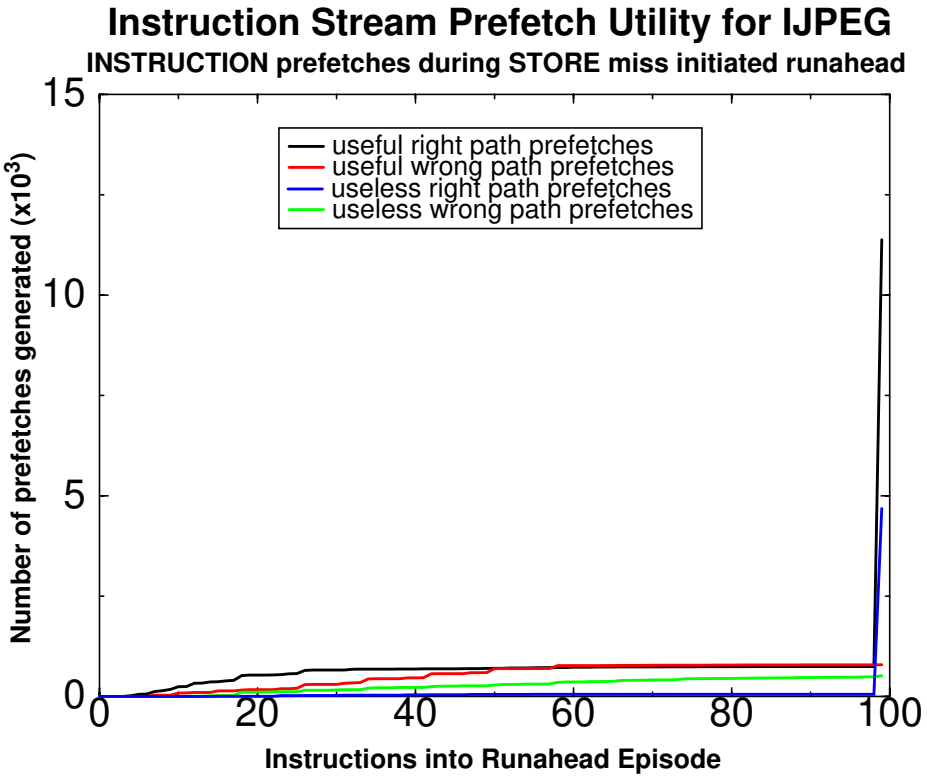


Figure 4.52

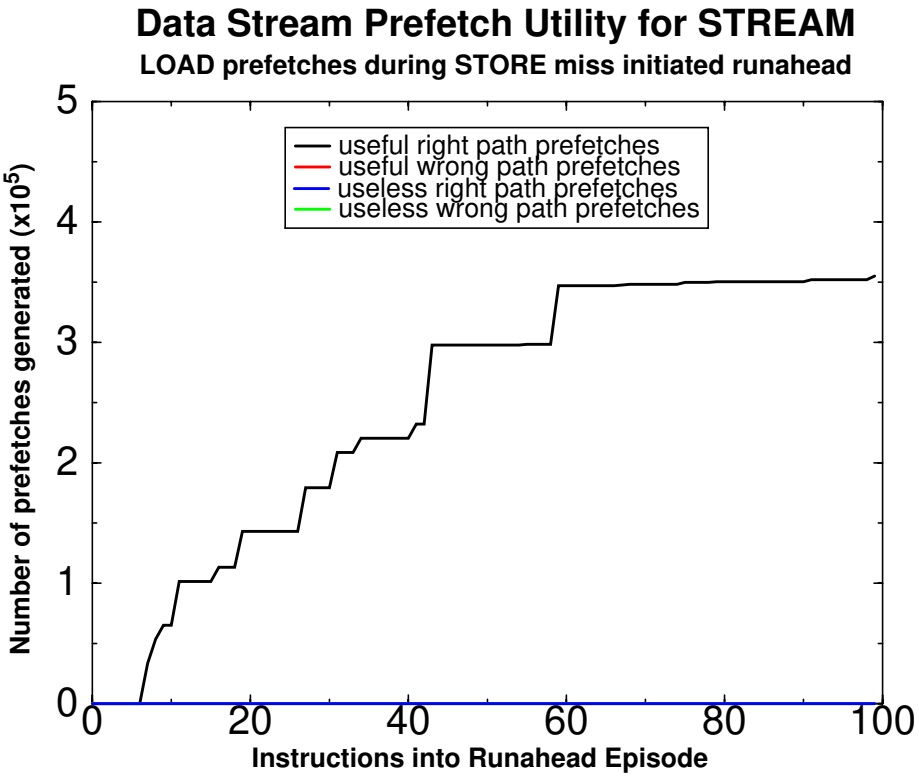


Figure 4.53

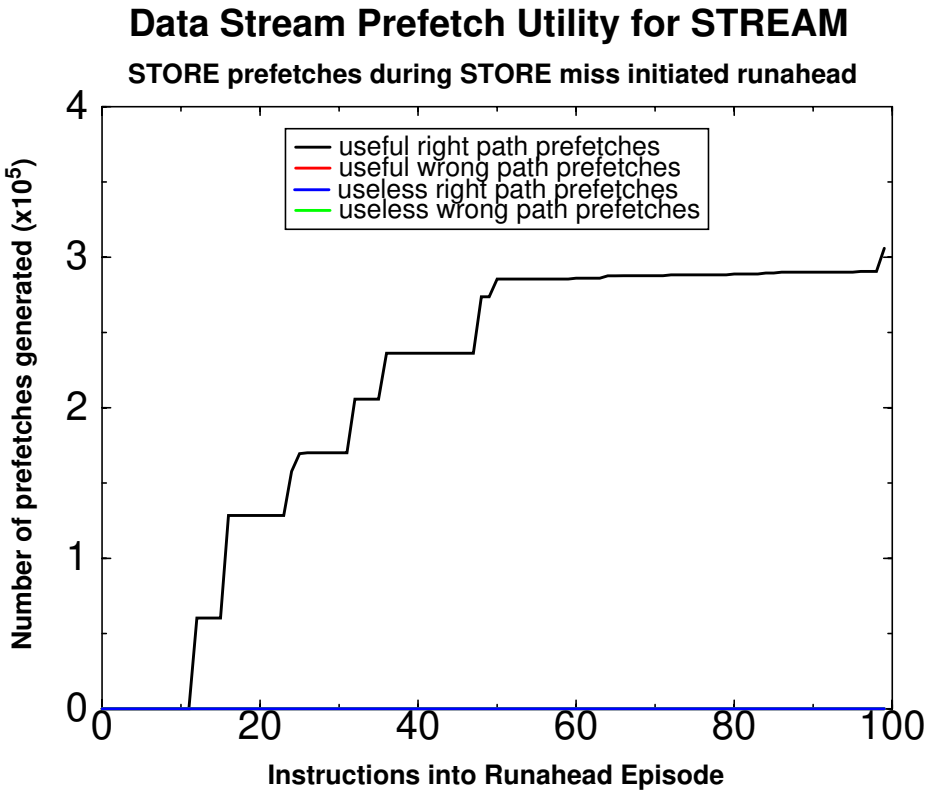


Figure 4.54

Data stream prefetching during instruction cache miss initiated runahead

Instruction miss initiated runahead is inherently more speculative than data stream miss initiated runahead due to the uncertainty arising from the skipping of instructions. Note that we do not present any results here for the STREAM benchmark due to its very low instruction cache miss rate.

The first set of plots, shown in Figures 4.55 and 4.56, are for the GO benchmark. The first of these plots, Figure 4.55, is for load prefetches. Note that useful prefetches outnumber useless prefetches for about the first two dozen instructions into the average episode. After this point the number of useless prefetches is approximately the same as the number of useful prefetches. The situation is even worse for store prefetches, as shown in Figure 4.56. Here useful prefetches are outnumbered by useless prefetches at the start of the average runahead episode.

The second set of plots, shown in Figures 4.58 and 4.59, are for the VORTEX benchmark. Data stream prefetching during instruction cache miss initiated runahead episodes makes no sense at all for VORTEX, as useless prefetches outnumber useful prefetches right from the start.

The third set of plots, shown in Figures 4.61 and 4.62, are for the PERL benchmark. As with GO, some number of useful load prefetches are generated during the first few instructions of the average runahead episode. This situation quickly changes however, with useless prefetches outnumbering useful prefetches. Interestingly, most of the useful load prefetches are on the wrong path. The situation reverses for store prefetches: most store prefetches are useful.

The fourth set of plots, shown in Figures 4.64 and 4.65, are for the IJPEG benchmark. Most of the load and store prefetches are useful, although for store prefetches the ratio approaches unity for runahead episodes that pre-process more than about 50 instructions.

Instruction stream prefetching during instruction cache miss initiated runahead

Data stream prefetches may suffer more from instruction cache miss effects than instruction stream prefetches as the accuracy of data stream prefetches can be affected by both register dependences upon skipped instructions and skipped branches. The validity of instruction stream prefetch addresses is affected primarily by the processor missing branches in skipped instruction cache lines. This implies that the processor may benefit more from instruction than data prefetching during instruction cache miss initiated runahead episodes. As before, we do not present STREAM results here due to the extremely low instruction cache miss rate for this benchmark.

The first plot is for the GO benchmark, and is shown in Figure 4.57. Useful prefetches outnumber useless prefetches by a wide margin for the first two dozen or so instructions into the average episode, after which the number of useless prefetches starts to become significant. This comes about due to a tapering off of the number of useful prefetches that are on the right path, which is largely a consequence of the processor missing taken branches in the instruction cache lines that are skipped. The number of wrong path prefetches, both useful and useless, tend to increase linearly. Note that the wrong path curves tend to have a more or less staircase shape, and that the width of each step is about 8 instructions. This is a result of the fact that most of the wrong path prefetches are sequentially clustered after the runahead initiating miss, and that the instruction cache line width is 8 instructions.

The plots for the VORTEX and PERL benchmarks, shown in Figures 4.60 and 4.63 respectively, show essentially the same behavior as GO.

The final plot is for the IJPEG benchmark, and is shown in Figure 4.66. IJPEG exhibits different behavior, with useful/right path prefetches dominating all other types. Unfortunately IJPEG does not suffer much from instruction cache misses, even without runahead.

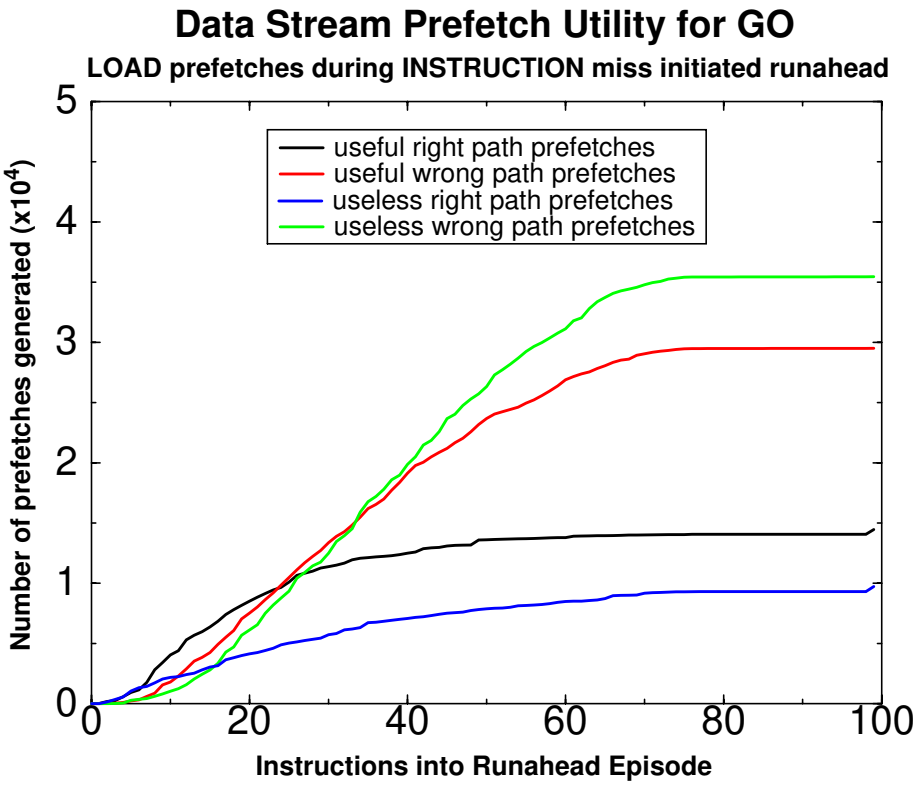


Figure 4.55

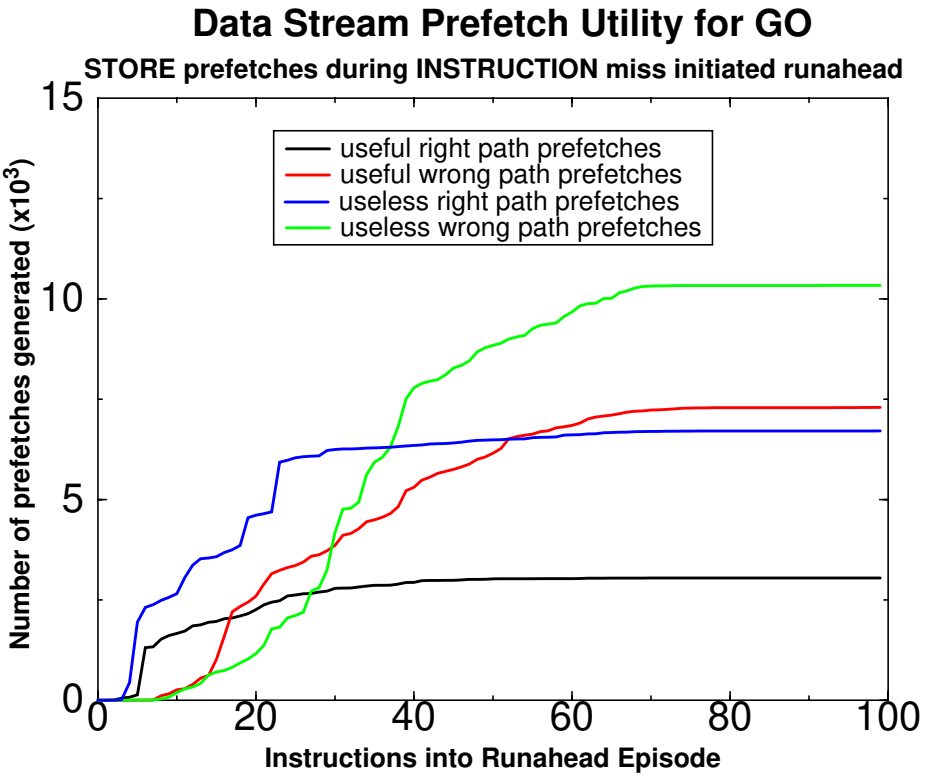


Figure 4.56

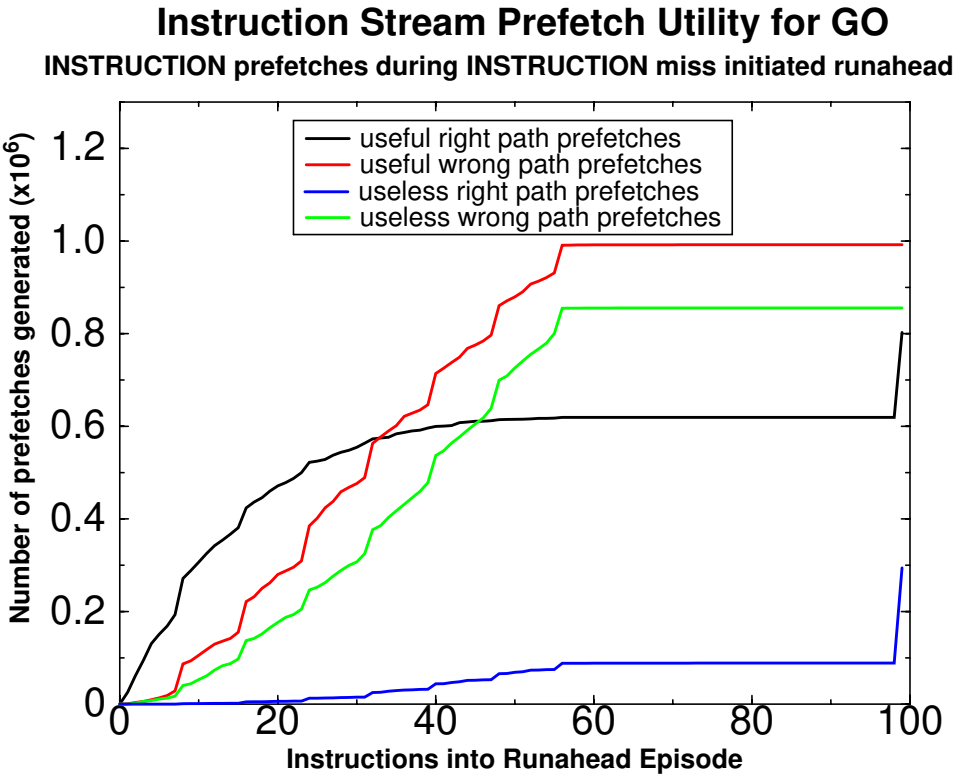


Figure 4.57

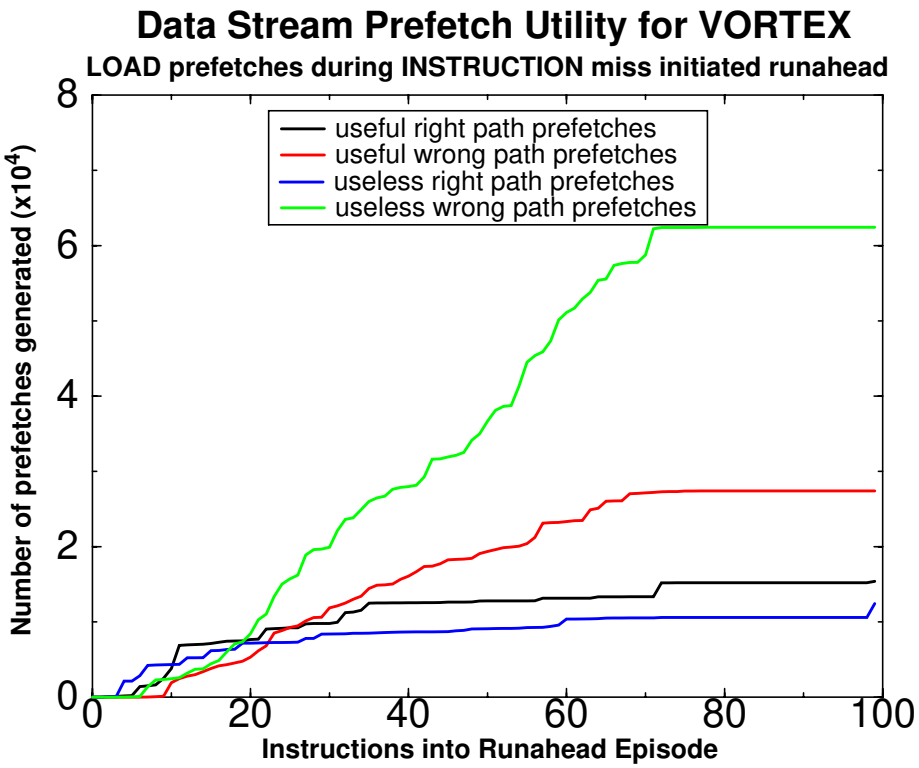


Figure 4.58

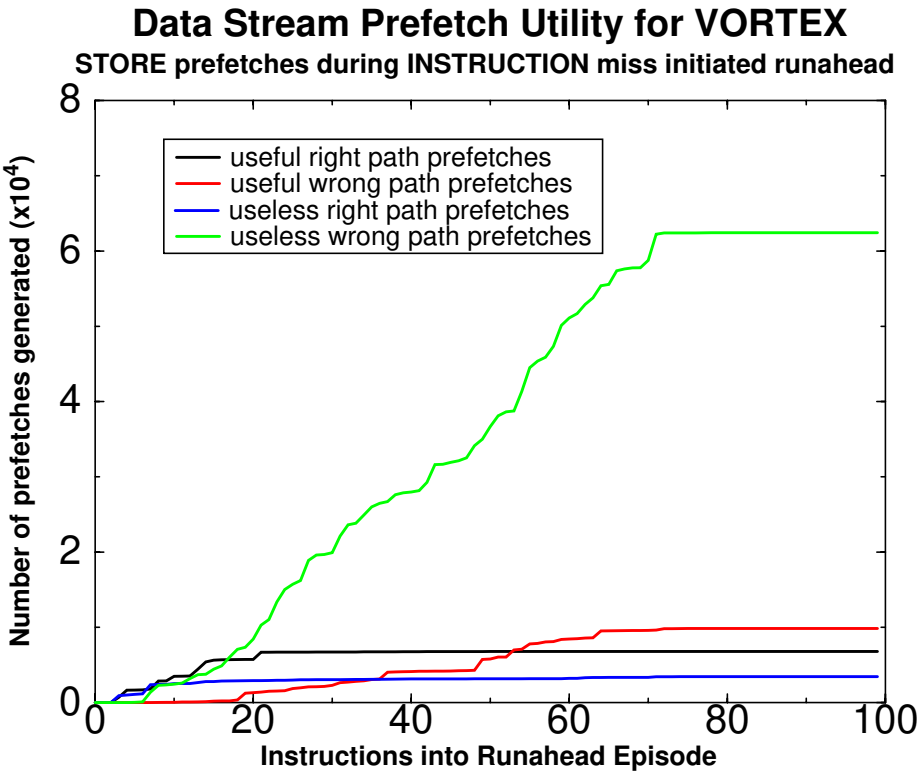


Figure 4.59

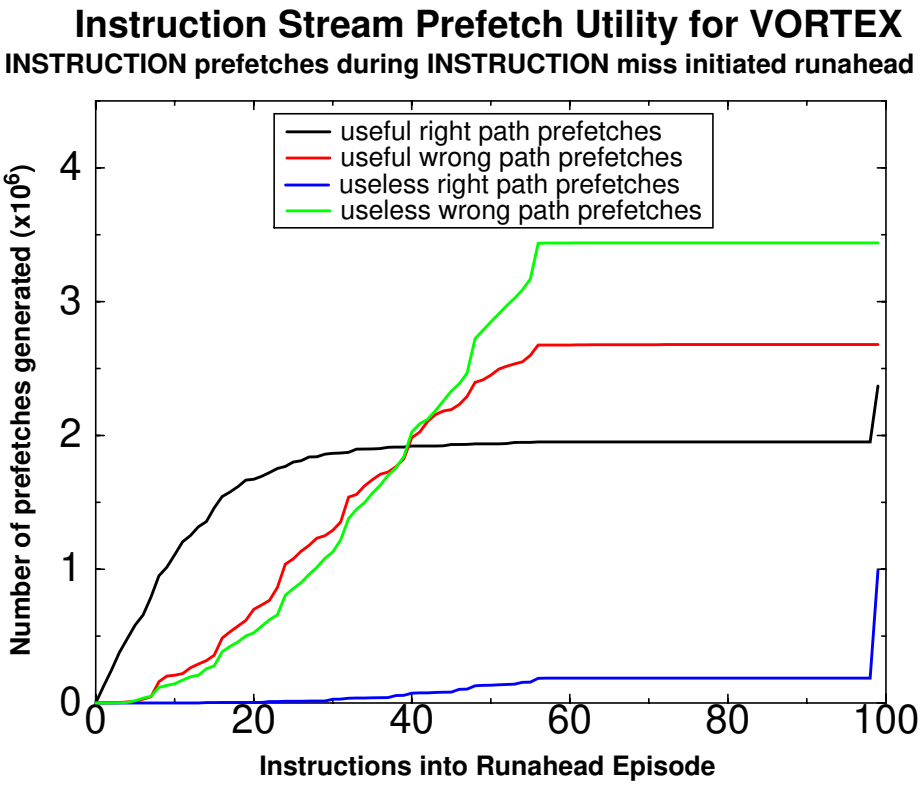


Figure 4.60

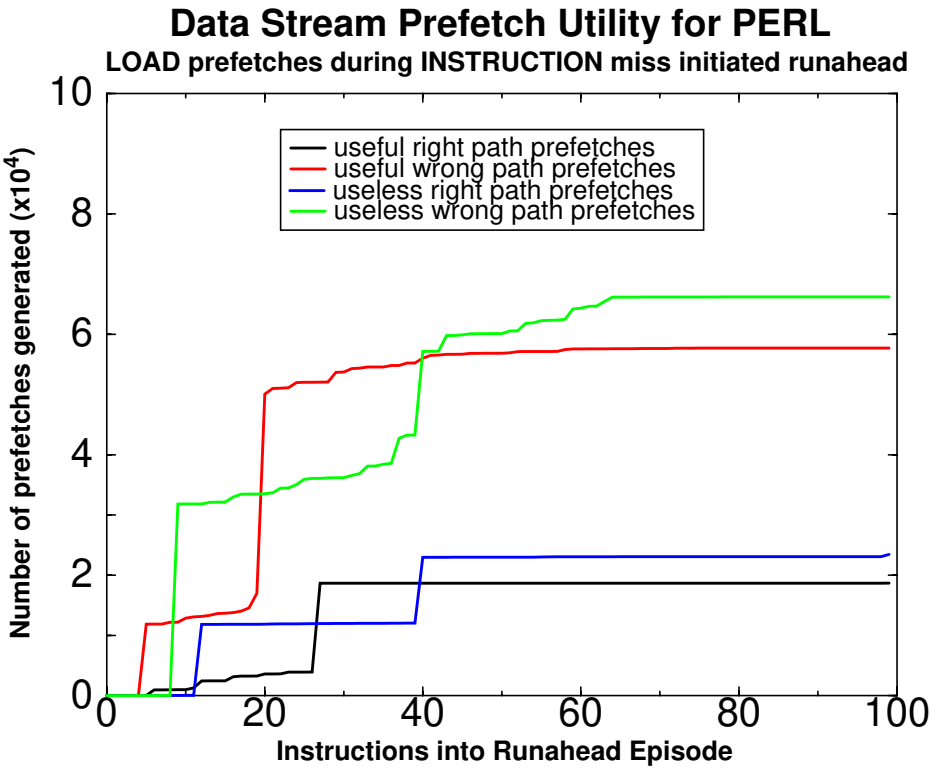


Figure 4.61

Figure 4.62

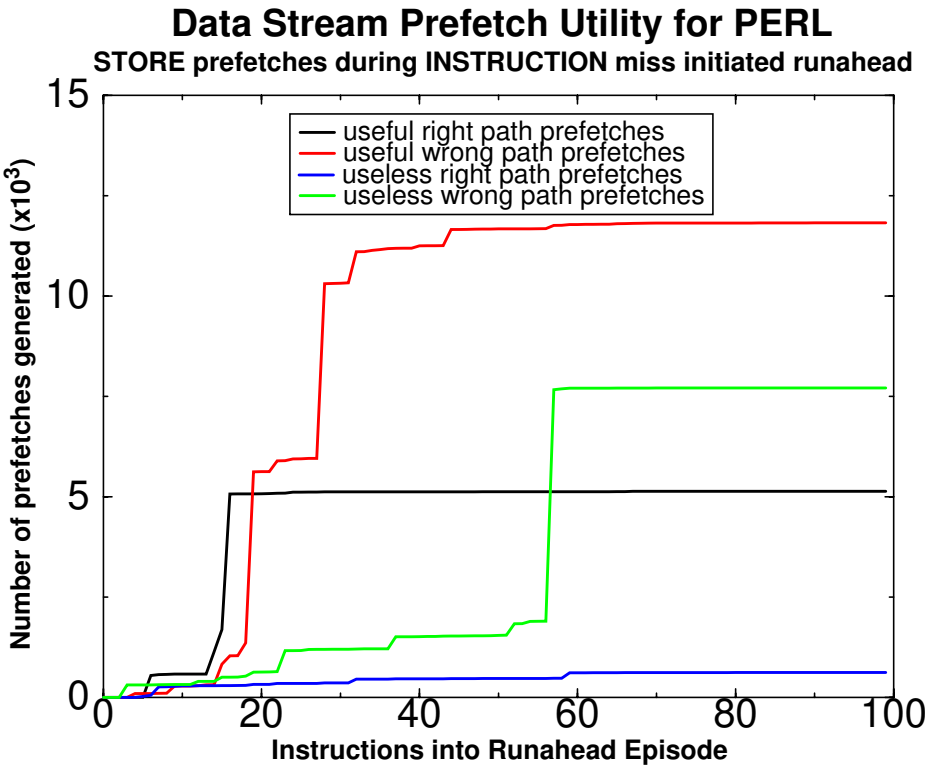
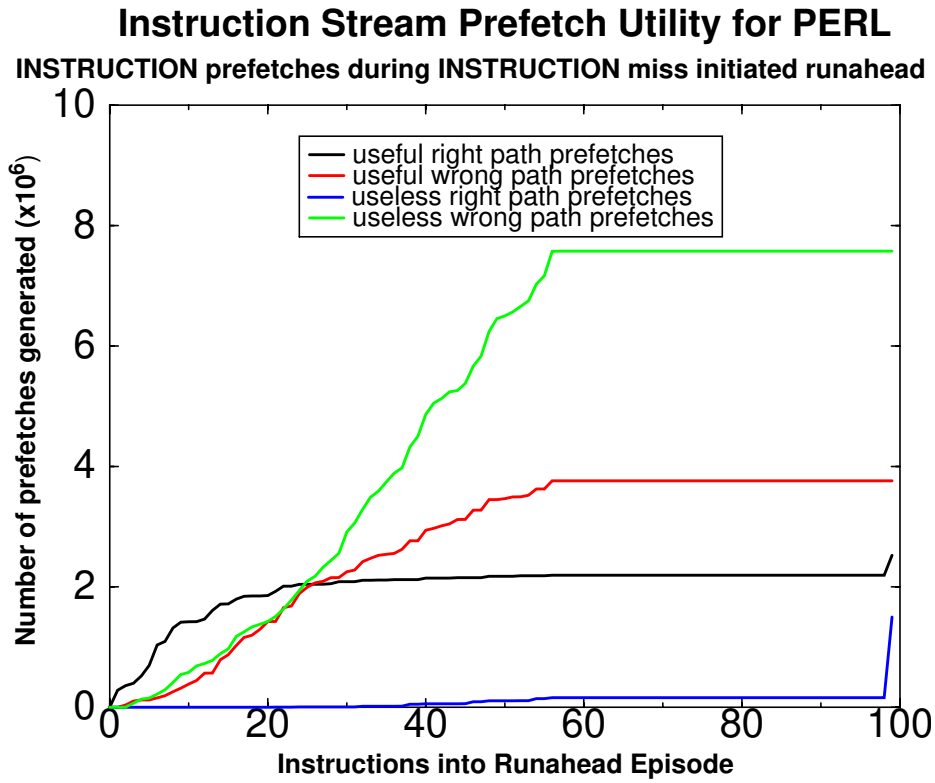


Figure 4.63



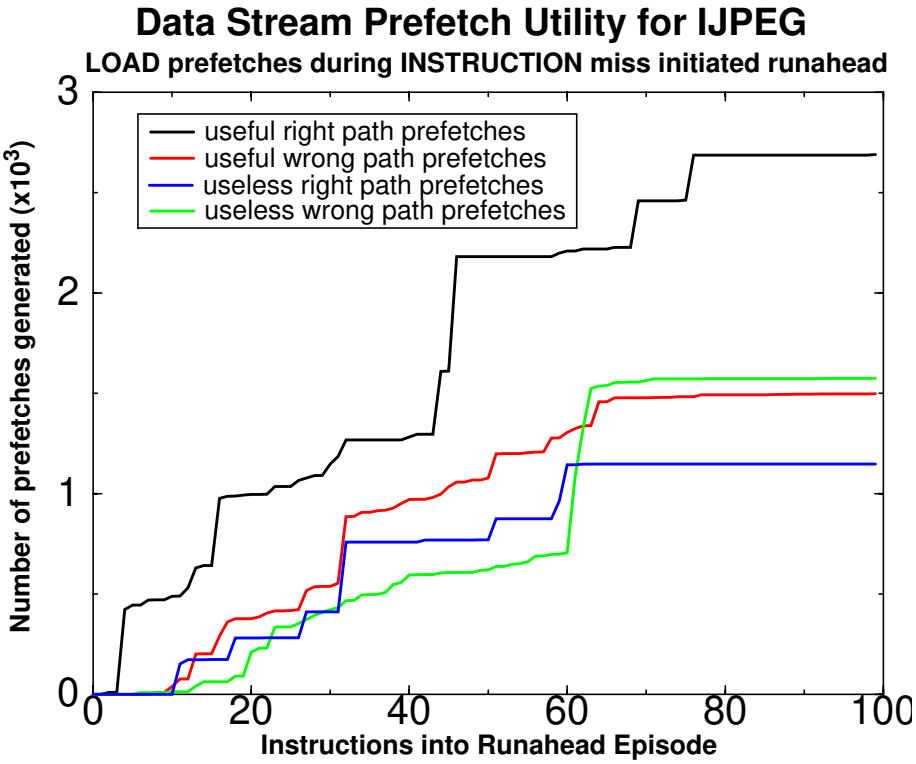


Figure 4.64

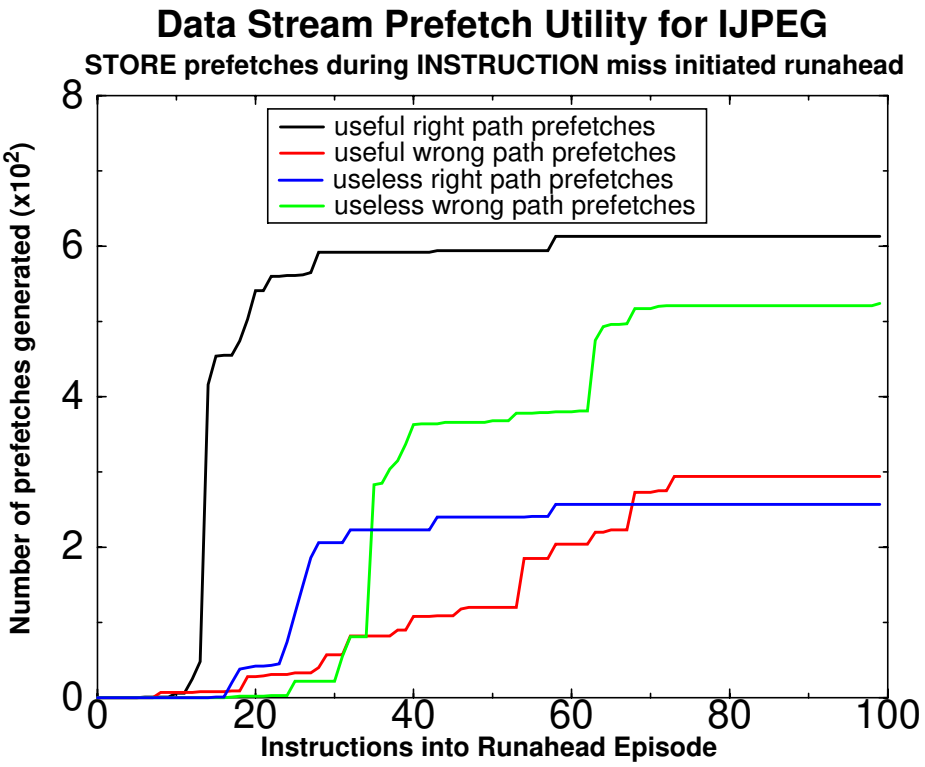
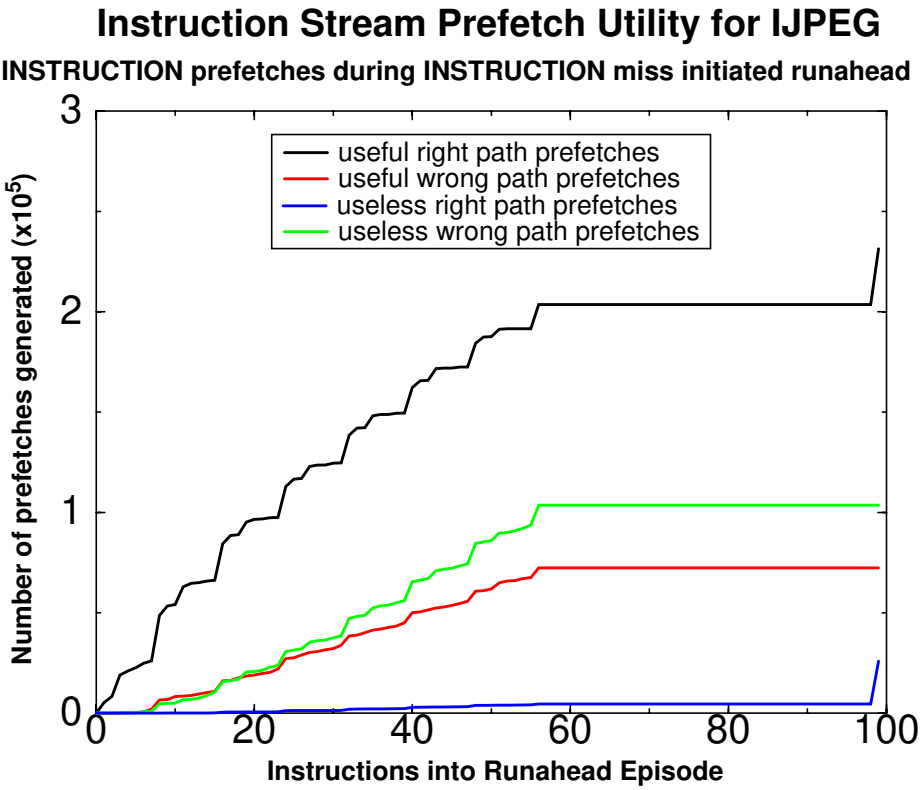


Figure 4.65

Figure 4.66



4.4 Measurements of miss-prefetch spatial locality

Plots that illustrate the spatial locality between runahead-initiating misses and subsequently generated prefetches are provided in Figures 4.67 through 4.75.

The x = 0 point on each plot represents the cache line address of each runahead-initiating L1 data cache miss. The other points on the x-axis represent the distance in cache lines between the address of each runahead-initiating miss and any prefetches that are generated during the corresponding runahead episode. For example, x = 1 represents the address of the next sequential line in the address space after the address that caused the runahead-initiating miss. The y-axis is a cumulative value. Its value at a particular x, say x = 35, represents the cumulative total of all runahead prefetches generated for any of the 35 sequential line addresses after the address of the miss that initiated runahead. The negative portion of the x-

axis represents addresses before the runahead-initiating miss. The extreme data points at x = +/- 100 lines represent the total number of runahead prefetches generated including those whose addresses correspond to lines that are at least 100 lines distant from their corresponding runahead-initiating misses. We include cumulative totals for both useful and useless prefetches on each plot.

4.4.1 Data stream prefetch locality

The fact that our simulated runahead processor can generate data stream prefetches during load-, store- and instruction-miss initiated runahead episodes complicates things somewhat. It does not make sense to talk about locality between data stream prefetches and runahead-initiating instruction stream accesses. For this reason the data stream prefetch locality plots that we present in this section are only for data stream prefetches that are generated during runahead episodes that are initiated on data cache misses. At any rate, relatively few data stream prefetches are generated during instruction stream runahead episodes.

The first data stream locality plot is for the GO benchmark, and is shown in Figure 4.67. As we saw earlier in the prefetch utility plots, virtually all of the prefetches are useful. The spikes at x = +/-100 indicate that most of the prefetches are for lines at addresses far removed from that of their runahead initiating miss. Note that while virtually all of the useless prefetches fall into this category, a small fraction of the useful prefetches show some spatial locality with the runahead initiating miss. The plot for PERL, shown in Figure 4.68, is very similar to that for GO.

The third data stream prefetch locality plot is for the VORTEX benchmark, and is shown in Figure 4.69. Nearly one-half of the prefetches are for lines that are located a small

positive distance from their runahead initiating miss. Nearly all of the remaining prefetches are for lines that show little spatial locality. Again, note that very few prefetches are useless, and that most of the useless prefetches show little spatial locality with their runahead-initiating miss.

The fourth data stream prefetch locality plot is for the IJPEG benchmark, and is shown in Figure 4.70. As with GO and PERL, most of the prefetches are for lines that show little spatial locality, and there are very few useless prefetches. The staggered nature of the useful prefetch curve is due to the way in which IJPEG structures it accesses.

The final data stream prefetch locality plot is for the STREAM benchmark, and is shown Figure 4.71. Many of the useful prefetches are for lines that are a small positive distance from their runahead initiating miss: these represent prefetches for the array that caused the runahead-initiating miss. The spikes at x = +/- 100 represent prefetches generated for arrays other than the one containing the runahead-initiating miss. Prefetches for one array exhibit no locality with respect to accesses in another, as the arrays do not overlap.

Figure 4.67

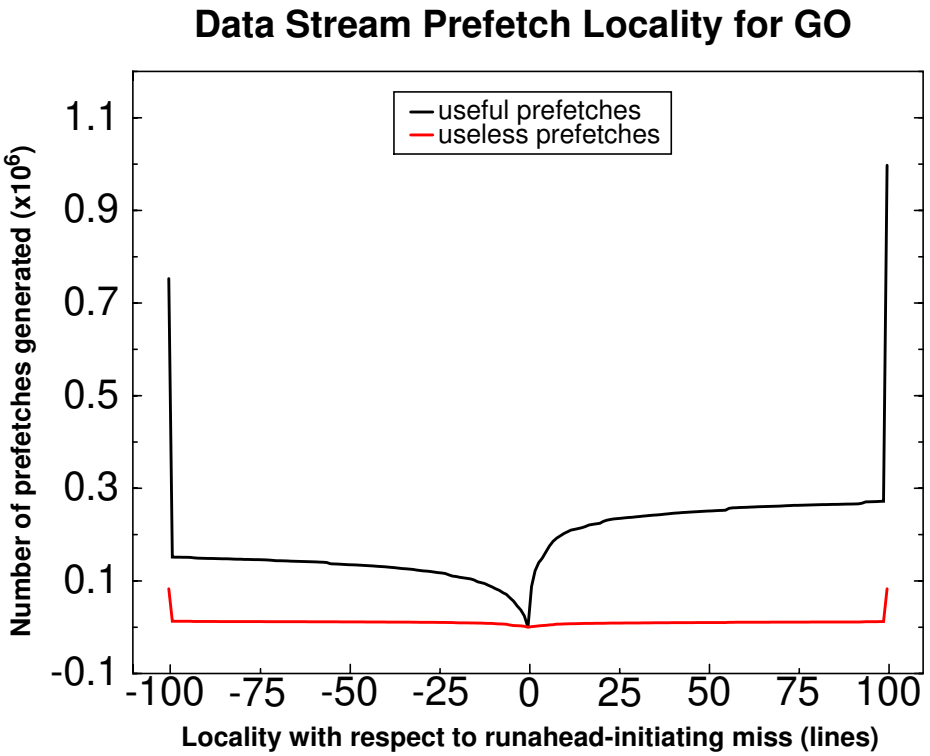
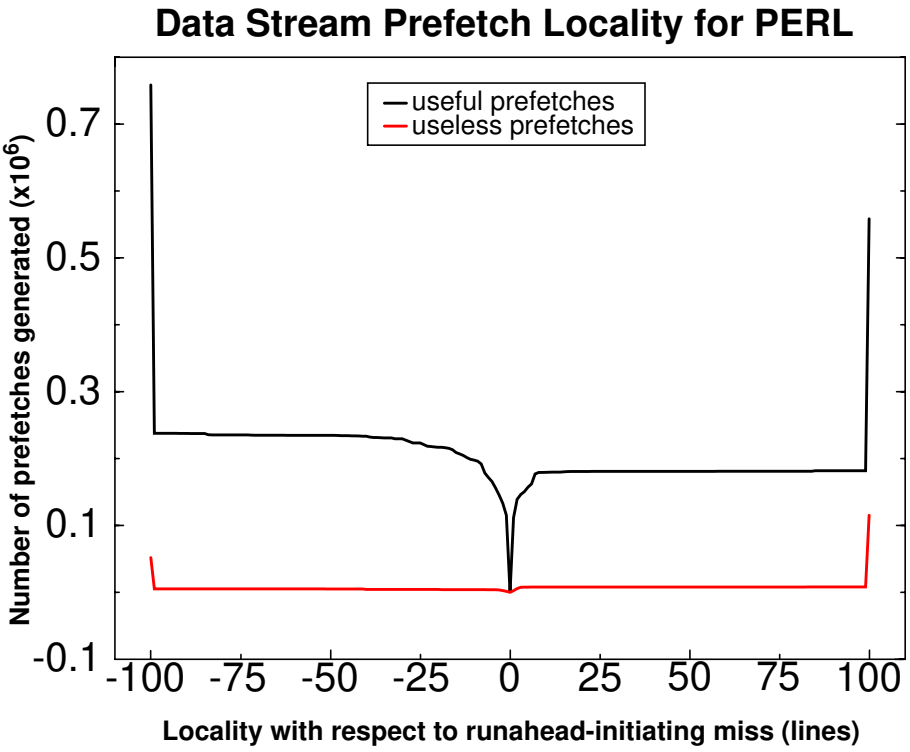


Figure 4.68



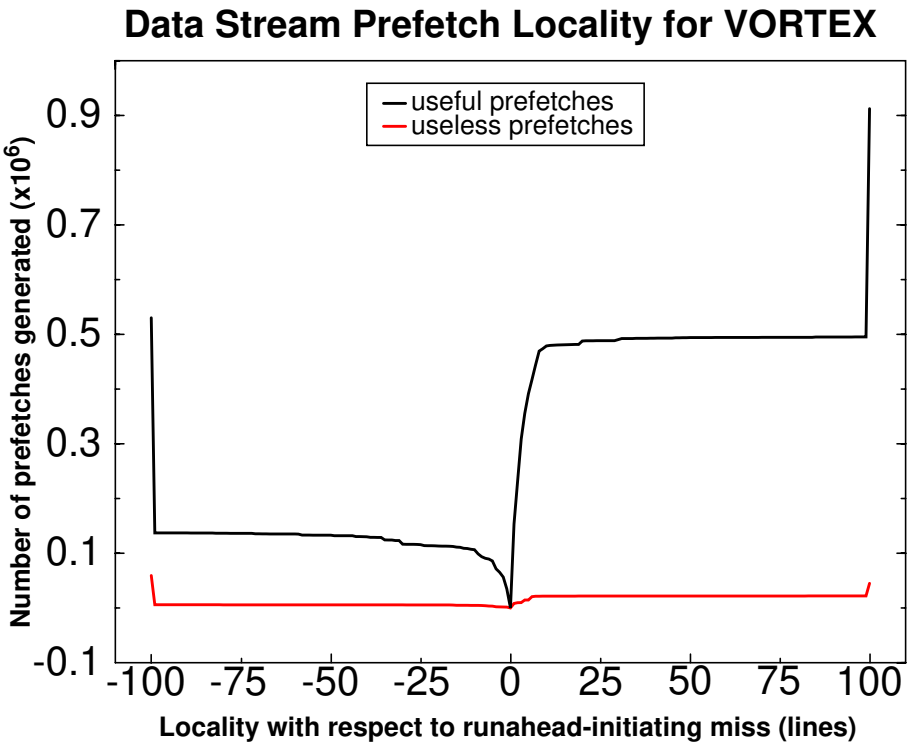


Figure 4.69

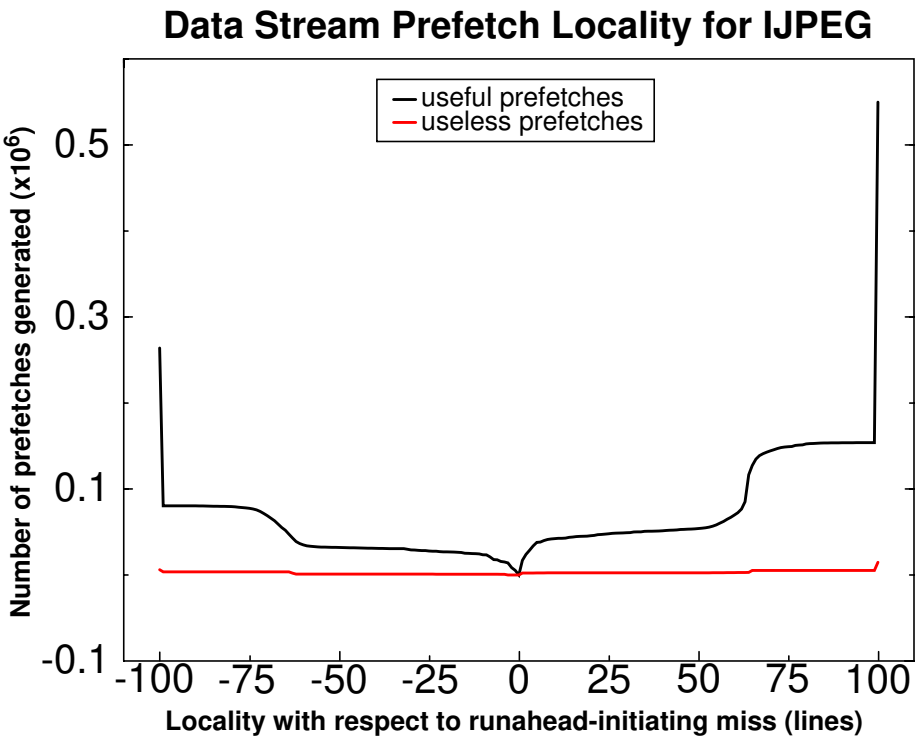
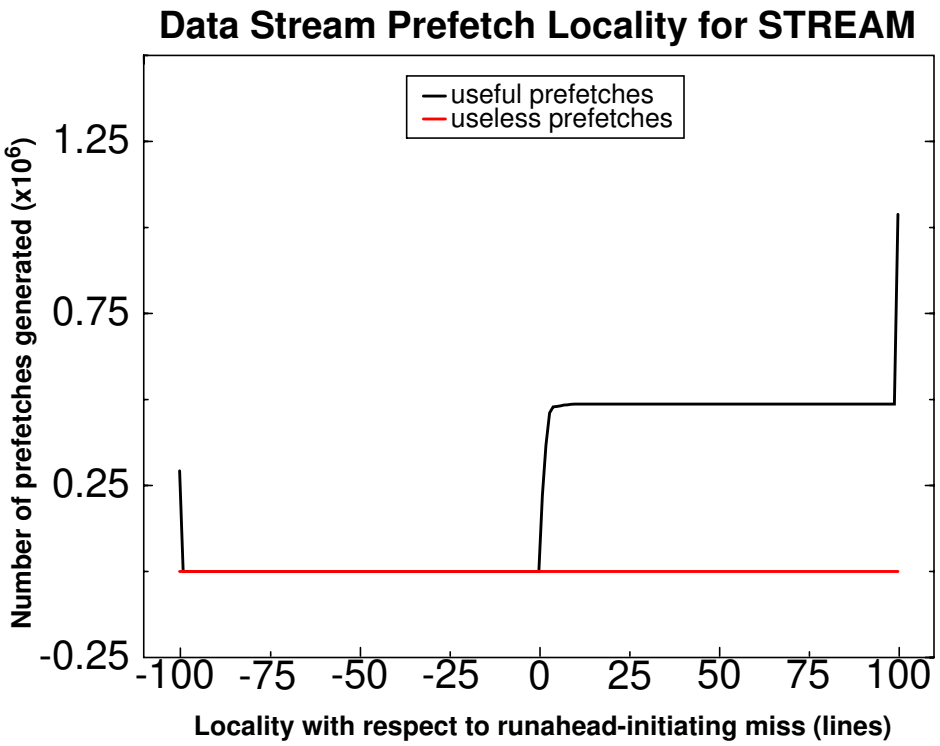


Figure 4.70

Figure 4.71



4.4.2 Instruction stream prefetch locality

The instruction stream prefetch locality plots require some additional discussion before we can present them. The locality between instruction stream prefetches generated during runahead episodes that are initiated by instruction stream accesses is easy to calculate and understand. However, the locality of instruction stream prefetches generated during data stream access initiated runahead is not quite so straightforward. We define the locality of these instruction stream prefetches to be difference between the cache line addresses of the instruction prefetches and the cache line address of the PC of the load or store access that initiated the runahead episode. As was the case with the prefetch utility plots, we do not provide a plot here for the STREAM benchmark due to its extremely low instruction cache miss rate.

The first instruction stream prefetch locality plot is for the GO benchmark, and is shown in Figure 4.72. This plot looks nothing like its data stream counterpart in Figure 4.67. The overwhelming majority of the prefetches are for lines that are a small positive distance from their runahead initiating miss. This was expected, as instruction stream accesses typically exhibit a significant amount of spatial locality, which is why sequential prefetching techniques work well with the instruction stream of most applications. Most of the prefetches that are generated for lines very close to their runahead initiating miss are useful; the ratio is lower the farther away they are generated. This was also expected. The plots for the VORTEX and IJPEG benchmarks, shown in Figures 4.73 and 4.74 respectively, are essentially similar to that for GO.

The fourth instruction stream prefetch locality plot is for the PERL benchmark, and is shown in Figure 4.75. This figure is essentially similar to that for GO, with the exception that the majority of the prefetches that are generated are useless. Even so, prefetches that are generated for lines very close to that of the runahead initiating miss are likely to be useful.

Figure 4.72

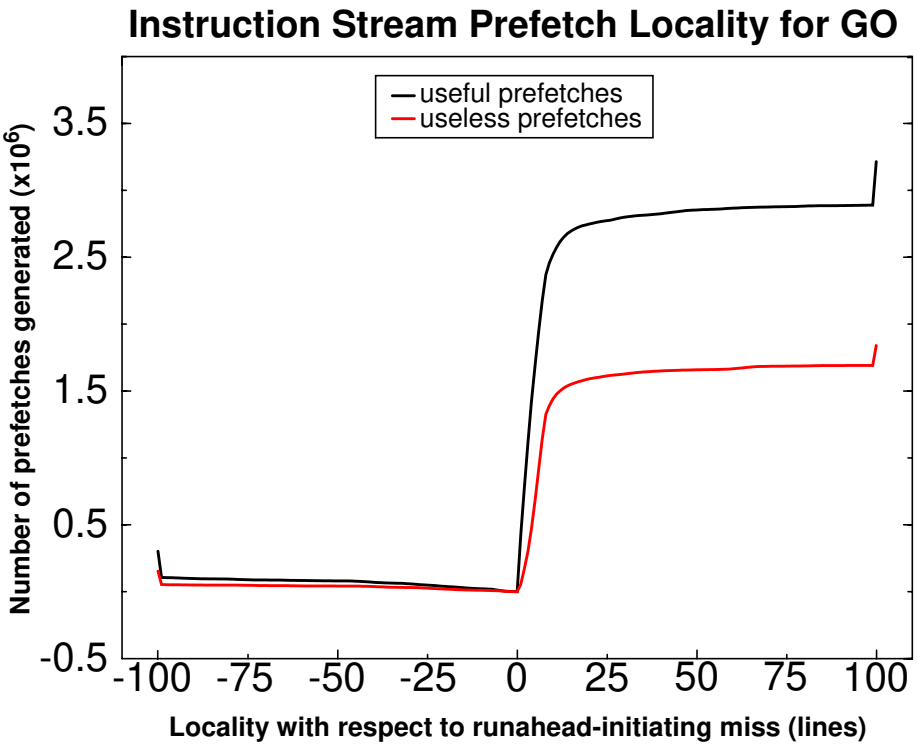
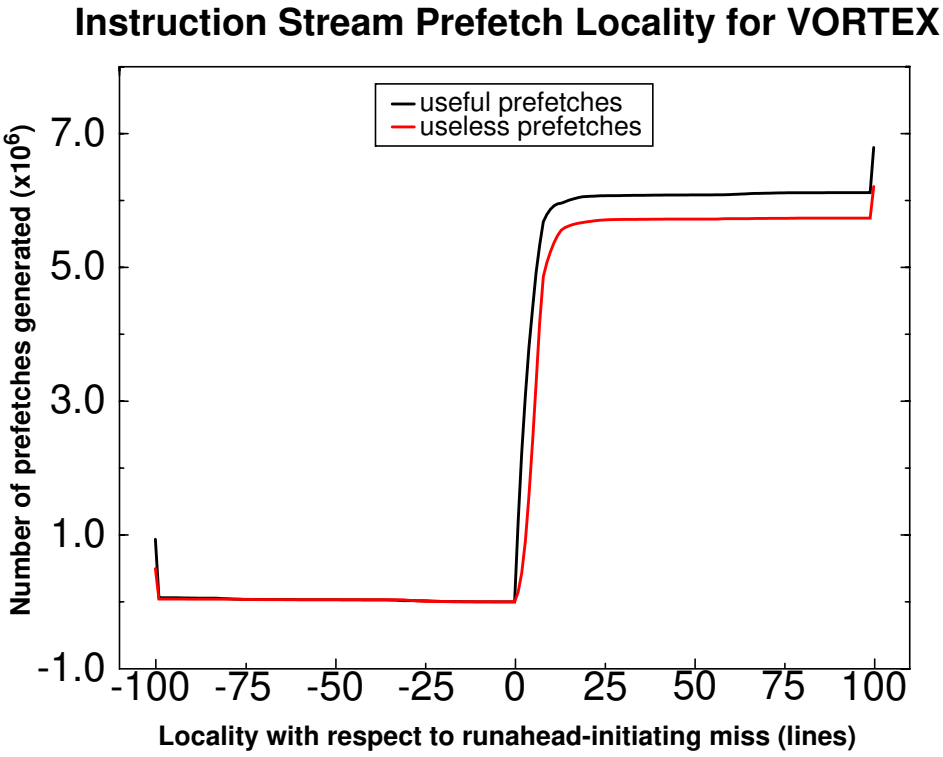


Figure 4.73



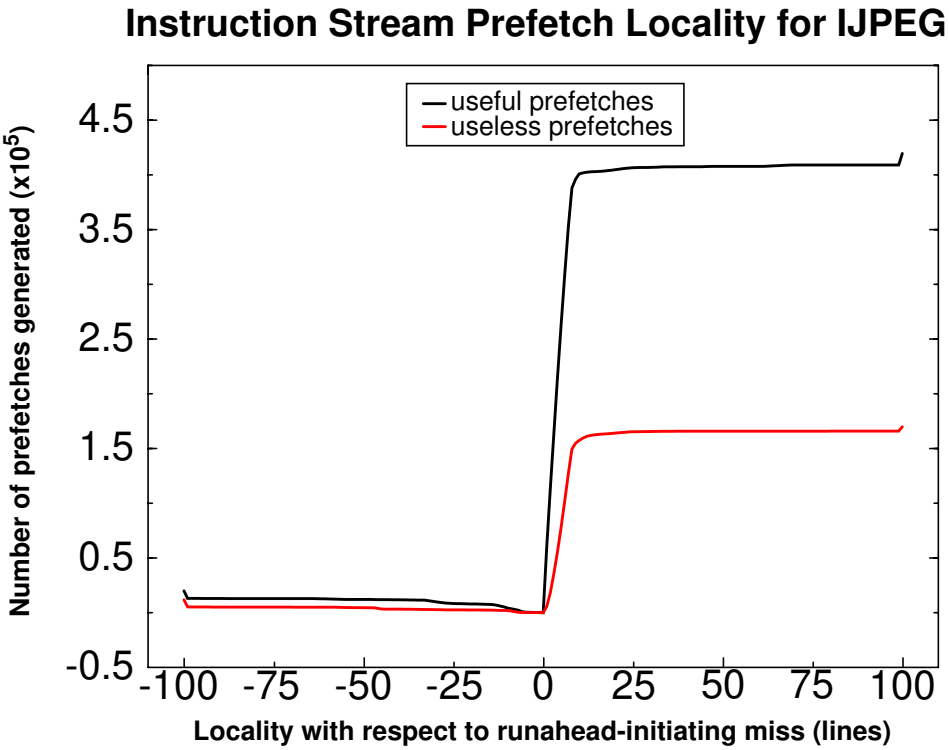


Figure 4.74

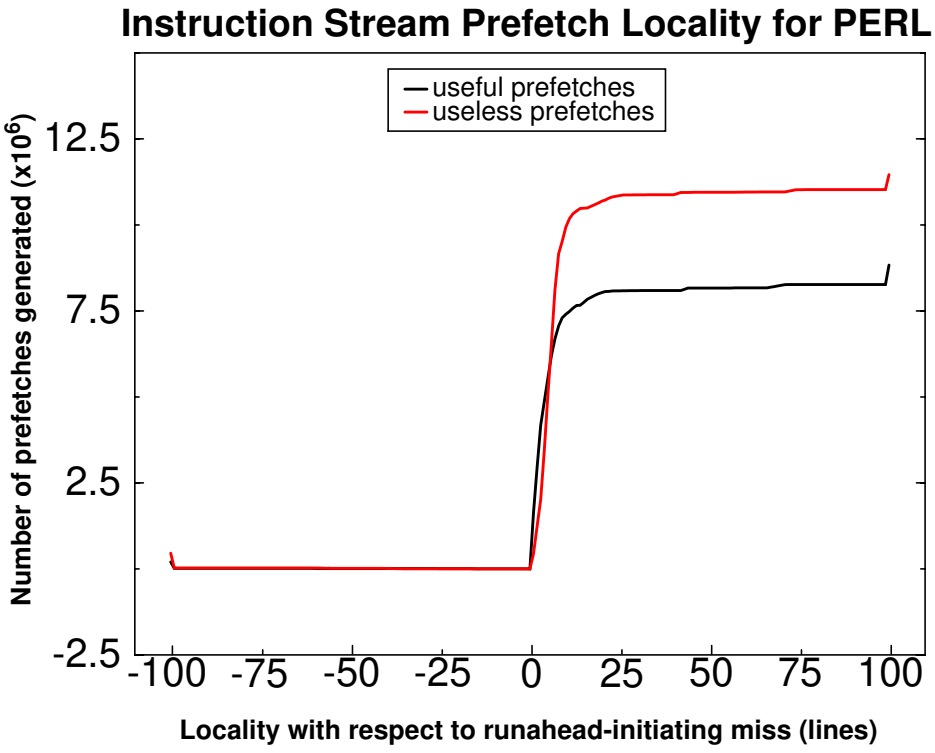


Figure 4.75

4.5 What happens to potential data stream prefetches during runahead

Many of the pre-processed load and store instructions are unable to generate prefetches for a variety of reasons:

- the instruction could not generate its target address with a VALID register, or
- the instruction hit in the L1 data cache, obviating the need to prefetch, or
- a demand fetch or prefetch for the missing line was already in progress, or
- the prefetch queue was full, forcing the processor to drop the prefetch, or
- the prefetch missed in the L2 data cache but its virtual page had not yet been the target of a demand fetch (page fault).

If a potential prefetch does not fall into any of the above categories, it will eventually be serviced by the memory hierarchy. Once this occurs it is still possible for the prefetch to fall into the useless prefetch category if its corresponding line in the L1 data cache is not accessed during normal operation before it is cast out of the cache, or the simulation ends.

We have divided up all of the potential load and store prefetches encountered during each simulation into the above categories, and present them in Figure 4.76. There are two stacked bars for each of the benchmarks: one each for load and store instructions. The height of each bar represents the total number of loads and stores that are preprocessed during the course of each simulation. Note that it is possible for this number to be quite high: the number of loads and stores pre-processed during the STREAM simulation exceeds the number of instructions of all types that are actually executed during the course of the simulation. This is a consequence of the processor spending more time in runahead than in normal operation.

From Figure 4.76 it can be seen that a small fraction of the pre-processed loads and stores generate prefetches. For the SPEC benchmarks most of the potential prefetches are lost as they either hit in the L1 data cache or their address register is INV. The first category

is misleading, as the fact that their accesses hit in the L1 data cache means that a prefetch is unnecessary. For this reason we have reproduced the plot in Figure 4.76 in Figure 4.77, with the L1 data cache hit category removed.

For the SPEC benchmarks most of the potential prefetches that can compute their target addresses, and that do not correspond to an in-flight fetch or prefetch, actually generate prefetches that are useful. Very few are dropped due to page faults or a full prefetch queue. The only SPEC benchmark that has to drop a significant number of prefetches due to a full queue is VORTEX, which it does for store prefetches.

The STREAM benchmark exhibits different behavior. A very large fraction of the potential prefetches correspond to lines for which there is already an outstanding prefetch. The nature of the benchmark leads to a great deal of overlap between consecutive runahead episodes. Very few are dropped due to an outstanding demand fetch; again the nature of the benchmark ensures that most accesses that go out to off-chip memory are prefetches and not demand fetches. A large fraction of the potential prefetches are dropped due to a full data stream prefetch queue. Increasing the size of the prefetch queue cannot improve performance as the main memory interface is already saturated with fetch, prefetch, and store-through requests. STREAM has an average main memory bandwidth of 1.39 GB/s, corresponding to an 87% utilization of peak bandwidth (Figure 4.9).

Figure 4.76

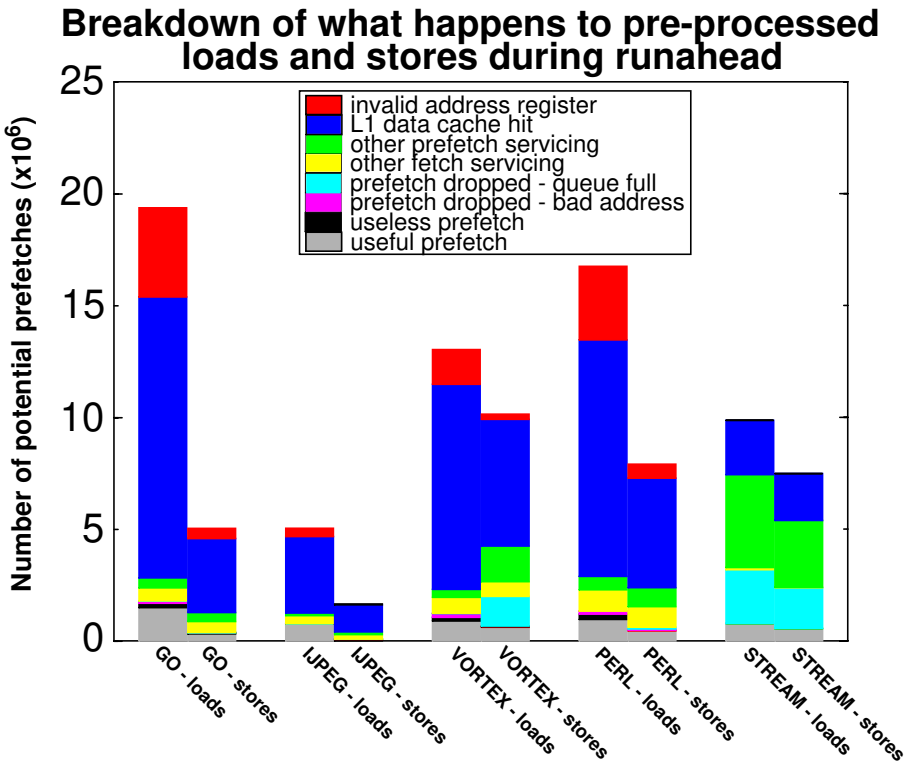
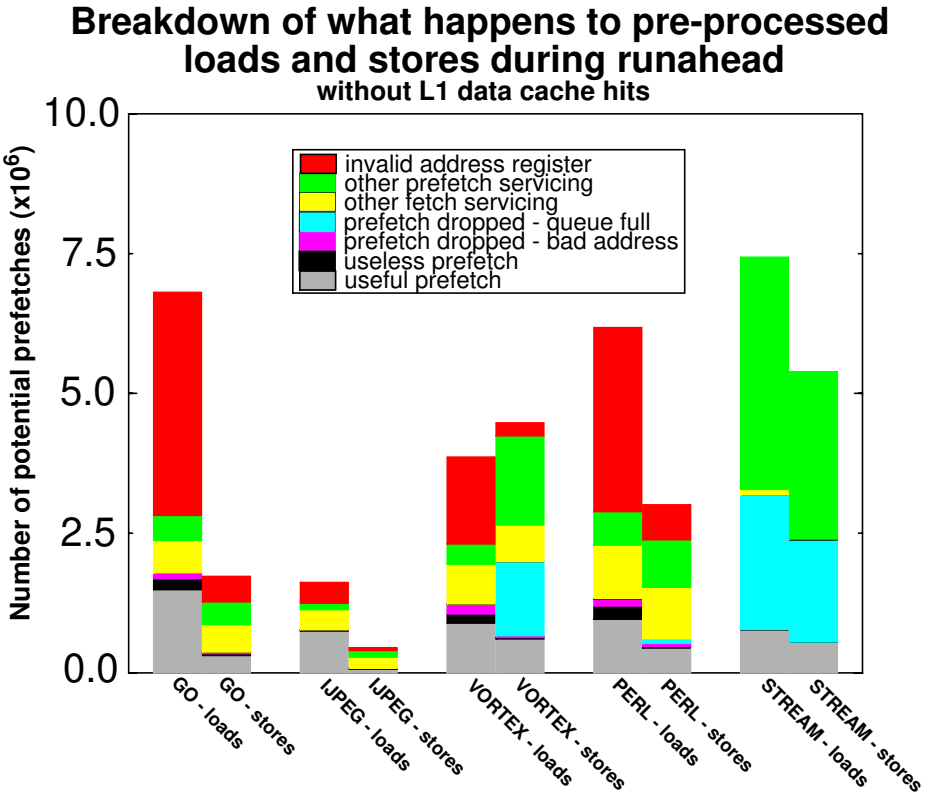


Figure 4.77



4.6 Branch Prediction Effects

Branch mispredictions can cause a runahead processor to pre-process instructions down wrong paths, which can reduce performance by generating useless prefetches.

A runahead processor may be able to reduce the number of branch mispredictions during both runahead and normal operation by pre-processing branch instructions before they are executed during normal operation. Conditional branch outcomes are recorded in the Runahead Branch Register (RBR) during runahead episodes for use during normal operation as branch predictors. The RBR contents can also be used during runahead episodes that overlap with previous episodes. In this way the RBR functions as a small window into the likely future outcome of conditional branches. Refer to Chapter 3 for more details on RBR operation.

The number of outcomes that are added to the RBR is proportional to the length of the average runahead episode, which is roughly equal to the average latency to off-chip memory. This leads to an interesting trade-off: a runahead processor with a high latency to off chip memory may have a better branch misprediction ratio than a runahead processor with a low latency off-chip memory. The potentially improved branch misprediction rate, combined with the longer runahead episode length, may allow a runahead processor with a high latency (or inexpensive) off chip memory hierarchy to have performance competitive, or perhaps even superior to, a runahead processor with a low latency (or expensive) off-chip memory hierarchy.

One factor that can affect the accuracy of predictors in the RBR is L1 instruction cache misses. Our baseline runahead processor can enter runahead mode on instruction cache misses, and once in runahead mode continues to pre-process instructions even if subsequent

instruction cache misses occur. The processor does this by assuming that the instructions in the missing instruction cache line are not of critical importance during runahead and simply skips over them sequentially until either normal operation resumes, or it finds a sequential instruction cache line to pre-process. This skipping of instructions can cause the RBR to yield incorrect predictors if the skipped instructions would have modified a branch condition or contain a taken branch.

In order to address these concerns we present several sets of branch misprediction data for each of our benchmarks. The baseline for comparison is a non-runahead processor using 1024 2-bit counters to predict conditional branches. Several runahead processor configurations are compared with this baseline. All of them have a 1024 entry RBR in addition to the 1024 2-bit counters. These models are as follows:

1. Runahead with real L2 instruction and data caches
2. Runahead with perfect L1 instruction cache and real L2 data cache
3. Runahead with perfect L1 instruction cache and no L2 data cache

The first model is the standard runahead processor model that we have been using all along. This model can incur instruction cache misses which can interfere with the operation of the RBR. The second model has a perfect L1 instruction cache, and is intended to illustrate more clearly the performance enhancing capabilities of the RBR approach. The third model is the same as the second, only it has no L2 data cache. The lack of an L2 data cache dramatically increases the length of the average runahead episode, which provides the processor with more opportunities to add branch outcomes to the RBR.

4.6.1 Branch misprediction rates

We present our branch misprediction statistics in Figures 4.78 through 4.82 with a series of four stacked bars for each benchmark: one bar for the non-runahead processor and one each for the three different runahead processor configurations. The height of each bar represents the total number of conditional branches executed in the benchmark in question. As a result all of the bars for each benchmark are the same height. However, each bar is divided into two sections representing the number of branches predicted with the RBR and the number of branches predicted with the 2-bit counters. The overall branch misprediction rate (percentage) for each processor configuration is provided at the top of the each bar. The branch misprediction rates for branches predicted with the RBR and 2-bit counters are provided to the right of their portion of each bar.

The results for the STREAM benchmark are shown in Figure 4.78. Note that the branch misprediction rate for all of the processor configurations is zero. The non-runahead processor naturally has to use the 2-bit counters to predict all of its benchmarks, while the runahead processors use the RBR to predict all of their branches. The RBR is able to provide predictors for all of the conditional branches as a result of the regular behavior of the STREAM benchmark. The small size of the STREAM executable also helps in that there are virtually no instruction cache misses. The fact that the branches in STREAM are independent of the streaming data also helps. The RBR works very well for this benchmark, unfortunately the simple nature of this benchmark does not require sophisticated branch prediction: static branch prediction would suffice.

The results for the IJPEG benchmark are shown in Figure 4.79. This benchmark exhibits markedly different behavior than STREAM: the RBR is effectively utilized during

runahead episodes, but the realistic nature of IJPEG as compared to STREAM keeps things under control. For the baseline runahead processor the RBR misprediction rate is a fairly low 8.98%. Unfortunately only 13% of the conditional branches are serviced by the RBR, preventing it from contributing much to the overall branch misprediction rate of 10.12%. The 2-bit counters have a misprediction rate of 10.26%, which is only slightly higher than the 2-bit counter rate for the non-runahead processor. This is a result of instruction cache misses during runahead, which can cause the processor to go down wrong paths and/or miss conditional branches during runahead. Both can result in the 2-bit counters getting improperly updated during runahead, and can also cause outcomes to be added in the wrong sequence to the RBR. This results in mispredictions that would otherwise not happen. This can be seen in the third stacked bar, which is for the runahead processor with a perfect L1 instruction cache. Note that both the 2-bit counter (10.24%) and RBR (8.56%) branch misprediction rates are reduced, resulting in an overall branch misprediction rate of 10.05%. Unfortunately, this overall branch misprediction rate is still close to that of the non-runahead processor without the RBR. The fourth stacked bar represents a runahead processor identical to that of the third, only without an L2 data cache. The lack of an L2 data cache significantly increases the length of the average runahead episode, providing the processor with more opportunities to add outcomes to the RBR. Removing the L2 data cache allows the RBR to service 39% of the conditional branches, with a misprediction rate of 5.23%. The longer runahead episodes also provide more opportunities to train the 2-bit counters, resulting in an improved misprediction rate of 9.84%. The overall misprediction rate for this runahead processor is a significantly improved 8.56%.

The results for the VORTEX benchmark are shown in Figure 4.80. Unfortunately runahead does not help the branch misprediction rate for this benchmark at all. The non-runahead processor is able to get a respectably low misprediction rate of 6.65% with only the 2-bit counters. The baseline runahead processor (second stacked bar) increases this somewhat to an overall rate of 7.95%. Nearly half of the conditional branches are serviced by the RBR, at a 7.66% misprediction rate, while the remainder are serviced by the 2-bit counters, at a 8.16% misprediction rate. The situation is improved somewhat when a perfect L1 instruction cache is used (third bar). Here the overall misprediction rate of 6.67% is nearly identical to that of the non-runahead processor, with a significantly lower rate of 4.81% provided by the RBR. This improved RBR performance is offset by the 2-bit counter misprediction rate of 8.16%. Eliminating the L2 data cache makes things worse with an overall misprediction rate of 8.34%, an RBR rate of 7.12%, and a 2-bit counter rate of 10.19%.

The results for the GO benchmark are shown in Figure 4.81. Runahead is able to consistently improve the branch misprediction rate for this benchmark. The non-runahead processor (first bar) has a rather high misprediction rate of 21.95%. This is improved upon somewhat with the baseline runahead processor (second bar) which achieves an overall rate of 21.09%, with an RBR rate of 20.41% and 2-bit counter rate of 21.49%. The misprediction rate improves even more when a perfect L1 instruction cache is added (third bar) providing an overall rate of 20.75%. Eliminating the L2 data cache provides a large misprediction rate improvement (fourth bar), with an overall rate of 16.73%. The RBR misprediction rate is reduced to 13.31%, and the 2-bit counter rate to 20.88%.

The results for the PERL benchmark are shown in Figure 4.82. The non-runahead processor utilizing only the 2-bit counters (first bar) provides a misprediction rate of 10.86%. The baseline runahead processor (second bar) has a somewhat worse overall misprediction rate of 13.19%. This is largely the result of the rather large RBR misprediction rate of 16.66%, and is caused by instruction cache misses during runahead which allow the RBR to get out of sync with the instruction stream. When a perfect L1 instruction cache is used (third bar), the situation improves. While the overall misprediction rate is not significantly improved (10.34%), the RBR rate does come down to 9.45%. The best results are obtained when the L2 data cache is deleted (fourth bar). When this is done the overall branch misprediction rate drops to 9.84%. This is entirely due to the RBR rate dropping to 8.65%, as the 2-bit counter rate increases slightly to 11.63%

Note that increasing the length of the average runahead episode does not necessarily translate into a fourfold increase in the proportion of branches serviced by the RBR. This is a consequence of our using aggressive runahead, which relies upon branch prediction to resolve conditional branches during runahead that cannot be resolved with VALID registers, as well as an increase in overlap between successive runahead episodes, which prevents the addition of branches to the RBR. If a branch prediction is incorrect, then the processor will go down the wrong path during the runahead episode. This can add many wrong path branch outcomes to the RBR, which are likely to be incorrect. The RBR detects these corrupted entries and flushes them when the processor discovers that a conditional branch has been mispredicted with the RBR. While it is possible to flush additional incorrect predictors from the RBR before they cause mispredictions during normal operation, it is not possible to flush the 2-bit counters. This inability to unwind the 2-bit counter state leads us to con-

clude that it may be possible to improve performance if 2-bit counter updates are not allowed during runahead.

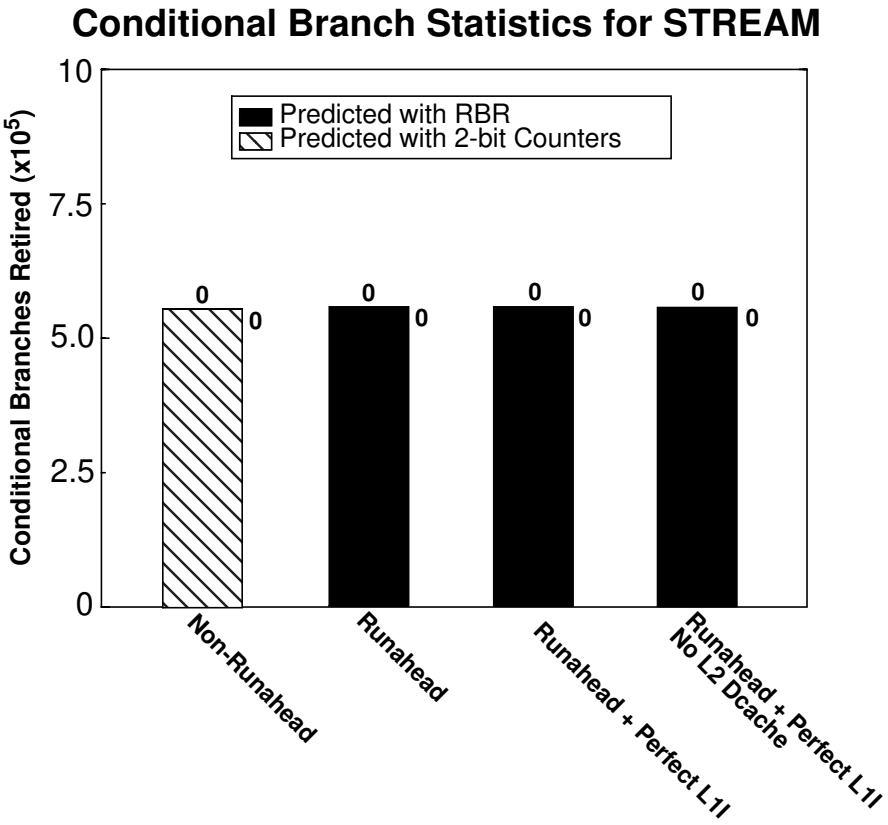


Figure 4.78

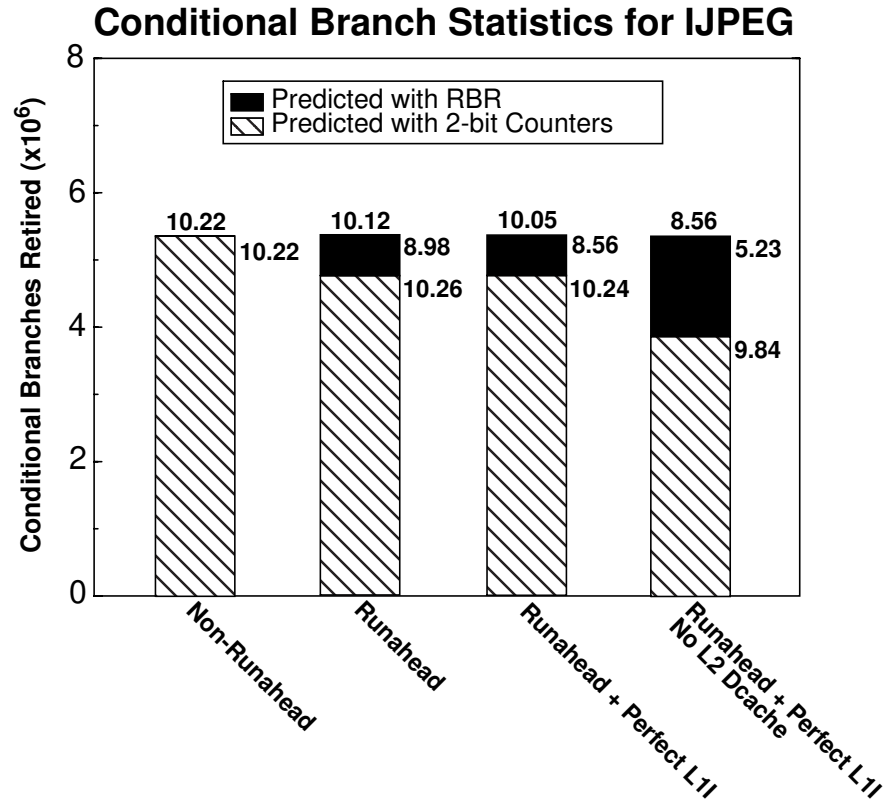


Figure 4.79

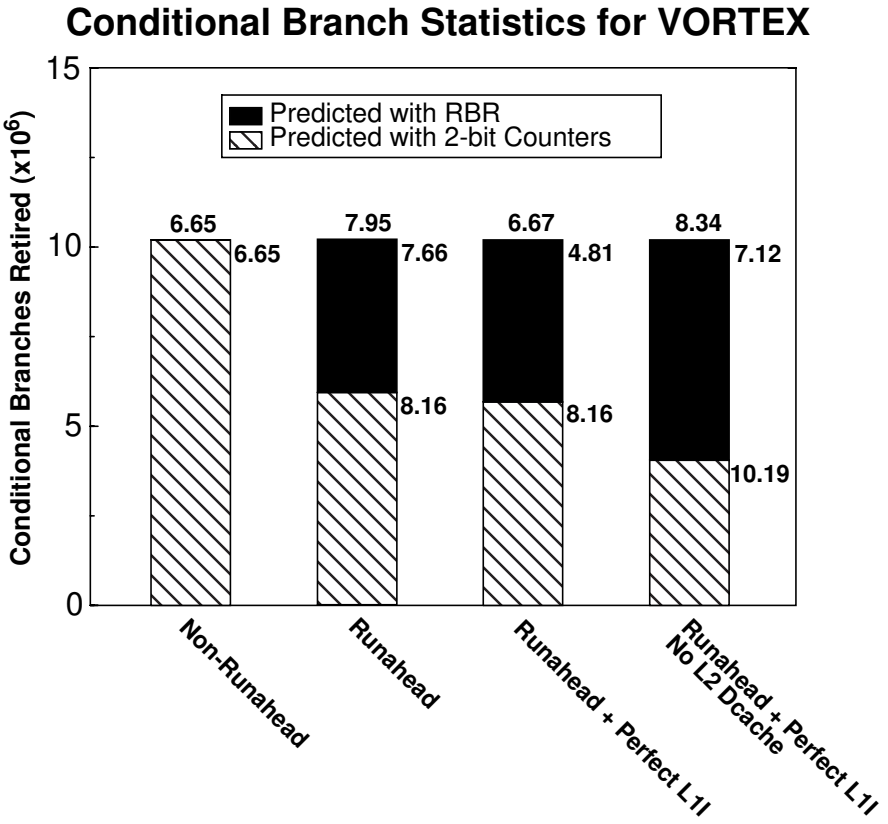


Figure 4.80

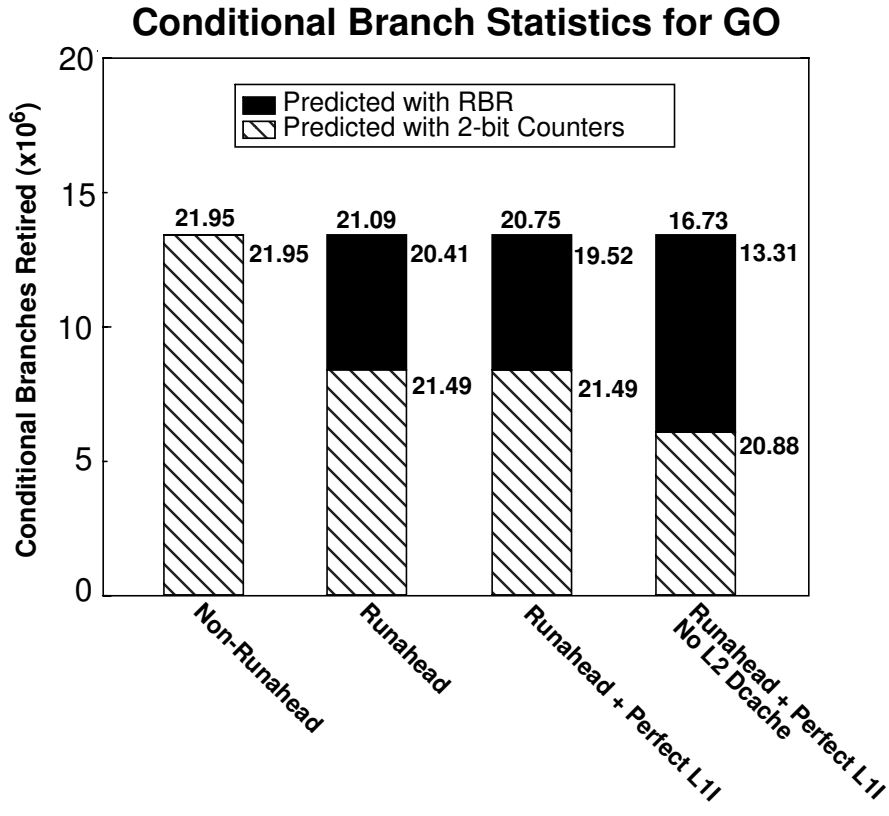
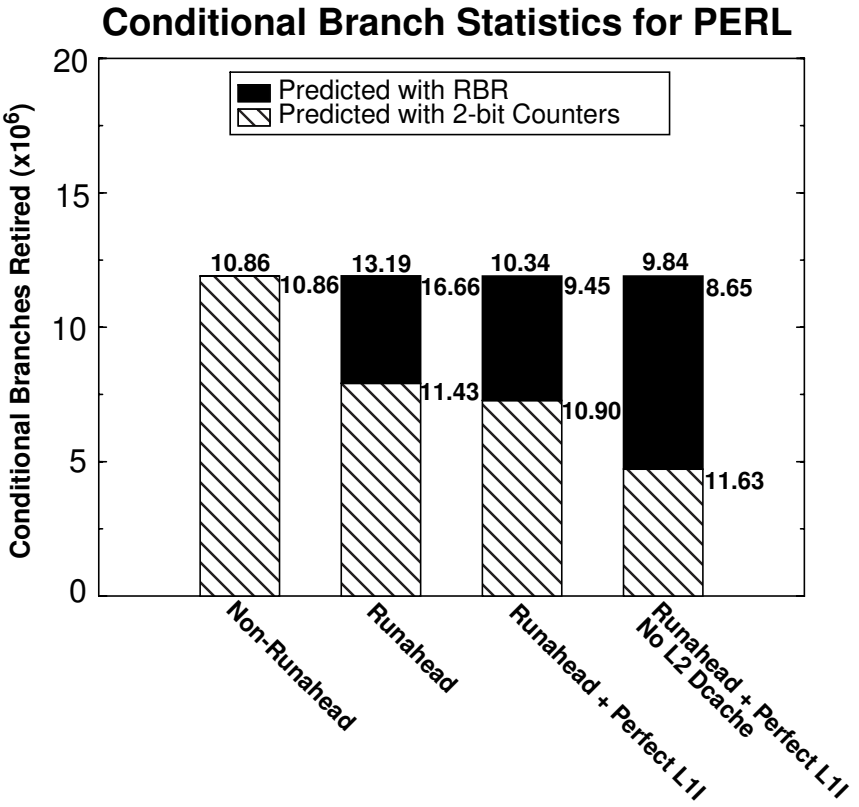


Figure 4.81

Figure 4.82



4.6.2 Runahead Branch Register Utilization

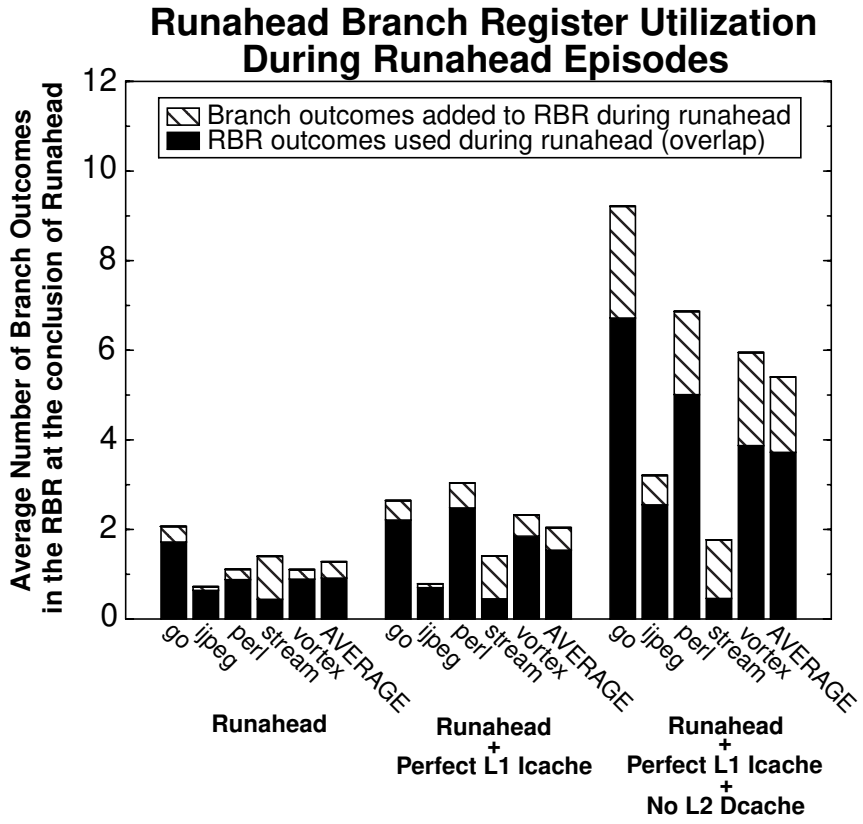
These simulations assume that the RBR can hold up to 1024 predictors in an attempt to ensure that RBR size is not a performance limiting factor. The average number of outcomes added to the RBR during runahead is a particularly useful statistic. Unfortunately this is not the entire picture, as runahead episodes can overlap, during which time outcomes cannot be added to the RBR. Instead, outcomes are read out of the RBR (non-destructively) for use as predictors during runahead. This results in an interesting trade-off: less overlap allows the processor to potentially add more outcomes to the RBR for use during normal operation, but the presence of overlap allows the processor to use the RBR as a predictor during runahead, which may increase the likelihood that runahead will stay on the proper path and generate more useful prefetches than would otherwise be the case.

We have measured the above statistics for each of the three runahead processor configurations previously described, and present the results for each benchmark in Figure 4.83. The height of each stacked bar in Figure 4.83 represents the number of outcomes in the RBR at the conclusion of the average runahead episode. These outcomes are divided into two categories: those corresponding to branches that overlap with prior runahead episodes, and those that were added to the RBR during the current runahead episode. As was mentioned earlier, the overlap outcomes were used as predictors during runahead, while the non-overlap outcomes were added to the RBR during runahead. There is also an AVERAGE case for each runahead processor configuration, which is simply the mean of the results for all of the benchmarks.

Note that for almost all of the benchmarks there are more RBR outcomes used as predictors during runahead then there are outcomes added to the RBR. The only exception to this rule is STREAM. The average number of outcomes in the RBR increases significantly when a perfect L1 instruction cache is used (the center group). The average number of outcomes in the RBR is directly related to the average number of outcomes added to the RBR, which is proportional to the number of instructions actually preprocessed on average. Hence reducing the instruction cache miss rate (or simulating a perfect instruction cache) is one way of increasing RBR utilization.

Eliminating the L2 data cache (right-most group) significantly increases the length of the average runahead episode, which increases the number of branches pre-processed. This can be readily seen in Figure 4.83. The only exception is the STREAM benchmark, which does not benefit from the L2 data cache due to its lack of data stream locality.

Figure 4.83



4.7 Summary and Conclusions

The highlights of our experimental results are as follows:

- Runahead leads to a significant reduction in CPI for all of the benchmarks that were examined. The reduction in CPI attributable to runahead ranges from 16% for IJPEG to 77% for STREAM.
- Data stream prefetches generated during load and store miss initiated runahead episodes are highly likely to be useful: typically at least 90% of prefetched lines are accessed at least once during normal operation. Data stream prefetches generated after an instruction cache miss is detected are less likely to be useful, however very few of these are generated.
- Instruction stream prefetches generated during runahead episodes are likely to be useful: typically about two-thirds of prefetched lines are accessed at least once during normal operation.

- Runahead can improve conditional branch prediction accuracy to some extent for some benchmarks. Our results indicate that attempting to save branch outcomes during runahead for use during normal operation will not noticeably improve performance in most cases.
- The price of these improvements is an increase in the bandwidth that the L2 caches and main memory buses must supply. Main memory bus utilization can become a bottleneck for some benchmarks, such as STREAM. Fortunately most of the bus bandwidth dedicated to prefetching is useful.

The most important result of these experiments is that it is possible to achieve good performance with even a simple pipeline when runahead is employed. Other than the potentially troublesome register file save and restore operation, the hardware required to implement runahead is rather modest, and should not impact the cycle time of an aggressive processor.

Allowing the processor to generate data stream prefetches during instruction cache miss initiated runahead episodes appears to be of dubious utility. Any data stream prefetches that are generated are about as likely to be useless as they are to be useful. However, the relatively few data stream prefetches that were generated in this fashion did not have a noticeable impact on our simulations.

Allowing the processor to generate instruction stream prefetches will improve performance. Instruction prefetches generated during load- and store-miss runahead episodes are more likely to be useful than their instruction-miss initiated runahead counterparts, although the majority of the latter are still useful. Reducing the number of instruction prefetches that are generated, particularly those generated late in the average runahead episode, may improve performance.

Chapter 5

Runahead at the Instruction and Function level

Thus far we have confined ourselves to looking at the effects that runahead has on overall performance statistics: CPI, number of prefetches generated, etc. Unfortunately, these statistics do not give us any information about what is happening during runahead for individual functions or instructions. This chapter addresses these concerns.

We modified our simulator such that it could record information about runahead on a per instruction basis. This was done by associating counters with every instruction in the benchmark. These counters record the following statistics:

- The PC, IPC, mnemonic, and registers used by the instruction
- number of useless prefetches generated by the instruction
- number of useful prefetches generated by the instruction
- number of times a line prefetched by an instruction is accessed during normal operation
- number of demand fetches generated by the instruction during normal operation
- number of times a line demand fetched by an instruction is accessed during normal operation
- number of times the instruction is executed during normal operation
- number of times the instruction is pre-processed during runahead

We also wanted to be able to determine exactly where in the benchmark source code each of the prefetch-generating instructions are. This was done by having ATOM record the function, source file name, and line number for every instruction in each benchmark.

5.1 Processor Model

All of the simulation results reported in this chapter are for processors that have perfect L1 instruction cache. This was done to remove second order effects from our processor model, as our intent is to examine what happens when instructions are actually pre-processed. In all other respects the runahead and non-runahead processor models are the same as those described in Chapter 3.

5.1.1 The distribution of prefetches on a per-function basis

In this section we present a breakdown of prefetch activity in each function. Tables 5.1 through 5.4 contain a listing of the top 40 prefetch producing functions in each benchmark. STREAM results are not provided, as this benchmark does not perform any function calls. Each entry provides the function name, the number of prefetches generated by instructions within the function, the fraction of the prefetches that were useful, and the fraction of all prefetches that were generated by the function. This last entry is used to sort the table entries in descending order.

Table 5.1 Prefetches Generated by Most Prolific Functions in GO

Function Name	Prefetches Generated by Function	Fraction of Prefetches that are Useful	Fraction of all Prefetches	Comment
blockuc	179100	0.921173	0.088261	
addlist	150427	0.786355	0.074131	
mrglist	76359	0.947786	0.037630	
dellist	68390	0.935195	0.033703	
chckside	65733	0.938844	0.032393	
lupdate	52084	0.968052	0.025667	
getterr	51463	0.849620	0.025361	
getefflibs	46609	0.938124	0.022969	
bestpot	42462	0.928148	0.020925	
cpylist	41615	0.983011	0.020508	
check_ex	38542	0.903741	0.018994	
undercut	36995	0.910285	0.018231	
ldndate	36097	0.945923	0.017789	
radiatepiece	35322	0.944850	0.017407	
adpot	33612	0.961353	0.016564	
upltr	32795	0.940845	0.016161	
iscaptured	31468	0.952205	0.015507	
getarmyuc_pot	30681	0.880023	0.015120	
findshapes	28236	0.989092	0.013915	
fixcnprot	26140	0.891048	0.012882	
newalive	24763	0.942333	0.012203	
rmpot	23162	0.955919	0.011414	
getarmyex_pot	22247	0.940037	0.010963	
getarmynbp	21797	0.811625	0.010742	
markspot	21764	0.974959	0.010725	
extendforeyes	21295	0.933412	0.010494	
startalive	20871	0.877150	0.010285	
pointeyes	19663	0.865789	0.009690	
getarmyth_pot	19606	0.905437	0.009662	
getarmycn_pot	19418	0.875322	0.009569	
evallibsterr	19245	0.924136	0.009484	
getarmylibs	17697	0.946319	0.008721	
canrunhere	17379	0.885494	0.008564	
getarmytv_pot	15599	0.891211	0.007687	
lcombine	15307	0.969360	0.007543	
findblock	15212	0.911320	0.007496	
cntterr	15212	0.936563	0.007496	
genrestatk	14330	0.942917	0.007062	
initarmyalive	13245	0.843488	0.006527	
livesordies	13119	0.926443	0.006465	

Table 5.2 Prefetches Generated by Most Prolific Functions in IJPEG

Function Name	Prefetches Generated by Function	Fraction of Prefetches that are Useful	Fraction of all Prefetches	Comment
rgb_ycc_convert	644767	0.989852	0.750714	
rgb_ycc_start	43508	0.780730	0.050657	
decode_mcu	26810	0.942186	0.031215	
jpeg_idct_islow	19038	0.998949	0.022166	
h2v2_merged_upsample	15500	0.989097	0.018047	
memset	12792	1.000000	0.014894	libc/memset.s
fullsize_smooth_downsample	9404	0.870693	0.010949	
h2v2_smooth_downsample	8227	1.000000	0.009579	
encode_one_block	7081	0.797486	0.008245	
sep_downsample	6168	1.000000	0.007182	
spec_difference_images	5085	0.984267	0.005921	
forward_DCT	4624	0.994810	0.005384	
expand_right_edge	4207	1.000000	0.004898	
pre_process_context	3689	0.995392	0.004295	
jpeg_fdct_islow	3090	1.000000	0.003598	
encode_mcu_huff	3047	1.000000	0.003548	
fix_huff_tbl	2813	0.991824	0.003275	
decompress_data	2547	1.000000	0.002966	
merged_2v_upsample	2488	1.000000	0.002897	
start_pass_merged_upsample	2427	1.000000	0.002826	
process_data_simple_main	2261	1.000000	0.002633	
compress_output	2017	0.731780	0.002348	
__divl	1830	1.000000	0.002131	libc/divrem.s
jpeg_read_scanlines	1790	1.000000	0.002084	
compress_data	1711	0.906487	0.001992	
fill_bit_buffer	1688	0.999408	0.001965	
compress_first_pass	1598	0.948686	0.001861	
memcpy	947	1.000000	0.001103	libc/memcpy.s
encode_mcu_gather	836	0.997608	0.000973	
access_virt_barray	788	0.623096	0.000917	
htest_one_block	787	0.872935	0.000916	
decompress	731	0.998632	0.000851	
spec_define_subimage_int	661	0.998487	0.000770	
_doprnt	620	1.000000	0.000722	libc/doprnt.c
free	599	0.994992	0.000697	libc/malloc.c
gen_huff_coding	550	1.000000	0.000640	
malloc	500	0.994000	0.000582	libc/malloc.c
cartesian_alloc	494	0.961538	0.000575	libc/malloc.c
start_input_pass	489	0.883436	0.000569	
read_markers	441	0.968254	0.000513	

Table 5.3 Prefetches Generated by Most Prolific Functions in PERL

Function Name	Prefetches Generated by Function	Fraction of Prefetches that are Useful	Fraction of all Prefetches	Comment
malloc	570019	0.884549	0.300116	libc/malloc.c
eval	253016	0.999186	0.133213	
cmd_exec	237005	0.999439	0.124783	
regexec	150807	1.000000	0.079400	
str_gets	133191	0.998851	0.070125	
str_sset	129381	0.991629	0.068119	
hsplit	124980	0.606377	0.065802	
str_new	54913	0.992916	0.028912	
memmove	47028	0.998830	0.024760	
safemalloc	34438	0.987369	0.018132	
str_nset	33081	0.979535	0.017417	
hfetch	27634	0.980097	0.014549	
cartesian_alloc	18238	0.797291	0.009602	libc/malloc.c
do_match	16457	0.992830	0.008665	
memcpy	15958	0.915904	0.008402	libc/memcpy.s
str_grow	10661	0.936591	0.005613	
hash_insert	10439	0.985631	0.005496	libc/malloc.c
memset	8958	0.998326	0.004716	libc/memset.s
hstore	7888	0.901242	0.004153	
cartesian_insert	5562	0.922869	0.002928	libc/malloc.c
nsavestr	2675	0.875888	0.001408	
yyparse	2048	0.878906	0.001078	
str_magic	816	0.976716	0.000430	
yylex	776	0.905928	0.000409	
stabent	374	0.903743	0.000197	
free	359	0.774373	0.000189	libc/malloc.c
cartesian_free	260	0.934615	0.000137	libc/malloc.c
make_op	143	0.958042	0.000075	
cartesian_merge	138	0.949275	0.000073	libc/malloc.c
op_new	131	0.984733	0.000069	
cartesian_growheap2	125	1.000000	0.000066	libc/malloc.c
scanident	123	0.959350	0.000065	
realloc	124	0.951613	0.000065	libc/malloc.c
yyerror	111	0.333333	0.000058	
scanstr	100	0.960000	0.000053	
__filbuf	94	1.000000	0.000049	libc/filbuf.c
str_free	91	0.890110	0.000048	
make_acmd	91	0.978022	0.000048	
block_head	79	1.000000	0.000042	
main	60	0.900000	0.000032	

Table 5.4 Prefetches Generated by Most Prolific Functions in VORTEX

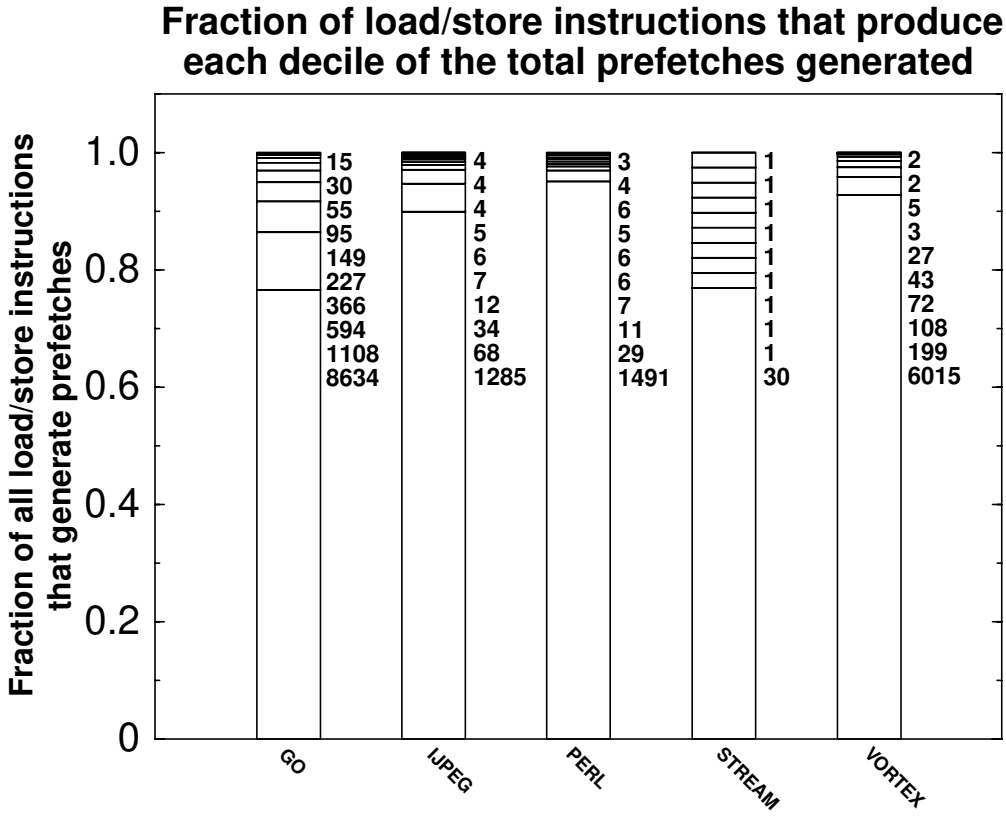
Function Name	Prefetches Generated by Function	Fraction of Prefetches that are Useful	Fraction of all Prefetches	Comment
memset	306155	0.999824	0.176391	libc/memset.s
Chunk_ChkGetChunk	235250	0.964973	0.135539	
TmFetchCoreDb	95140	0.979241	0.054815	
memcpy	72160	0.962486	0.041575	libc/memcpy.s
Mem_GetWord	62077	0.982200	0.035766	
Grp_GetRegion	46910	0.939139	0.027027	
TransGetMap	46152	0.993326	0.026590	
OaGet	44366	0.969346	0.025561	
Ut_StackTrack	43600	0.668991	0.025120	
Tree_CompareKey	42506	0.899308	0.024490	
C_ReFaxToDb	38364	0.975263	0.022103	
Mem_GetAddr	31738	0.997416	0.018286	
malloc	24902	0.924745	0.014347	libc/malloc.c
SpclAddIntoTree	24739	0.996605	0.014253	
Ut_MoveBytes	21900	0.993607	0.012618	
Person_Import	18599	0.984462	0.010716	
Grp_GetFrozenEntry	17318	0.965412	0.009978	
SaFindIn	17101	0.945442	0.009853	
OmGetObjHdr	16641	0.886545	0.009588	
Grp_GetEntry	15760	0.999873	0.009080	
KernelGetAttrInfo	14922	1.000000	0.008597	
__divlu	13990	0.992995	0.008060	libc/divrem.s
OaGetObject	13866	0.984494	0.007989	
Tree_AddInto	12999	0.833141	0.007489	
OaUpdateObject	12830	0.995012	0.007392	
Void_ExtendCore	12824	0.988459	0.007389	
Tree_RecurseSearch	11986	0.930502	0.006906	
sprintf	11949	0.302536	0.006884	libc/sprintf.c
Trans_FetchAttrOffset	11217	0.999554	0.006463	
Chunk_ChkPutChunk	10887	0.982824	0.006273	
OmGetObject	10736	0.792288	0.006186	
printf	10621	0.147350	0.006119	
Ut_PrintErr	10480	0.955057	0.006038	
EnvFetchAttrOffset	10376	0.912298	0.005978	
TmIsValid	10356	0.968231	0.005967	
EnvFetchObjSize	9851	0.998985	0.005676	
C_GetObjectImage	9836	0.999593	0.005667	
Tree_GetRecursePos	9729	0.998664	0.005605	
Ut_CompareString	9500	0.941579	0.005473	
Mem_NewChunkChunk	9450	0.994074	0.005445	

5.1.2 The distribution of prefetches on a per-instruction basis

We were particularly interested in finding out which instructions were the most effective at generating prefetches during runahead. It turns out that a small fraction of the load and store instructions tend to generate most of the prefetches. This is illustrated in Figure 5.1. The load and store instructions in each benchmark are sorted based upon the number of prefetches that they generate. The instructions in this sorted list are then divided into 10 groups, or deciles, each of which generate approximately 10% of the total prefetches. The first decile in each stacked bar (depicted at the top of each bar) corresponds to a very small fraction of the total number of load and store instructions that generate prefetches. The size of the deciles increase as you move down each bar, corresponding to the lower number of prefetches generated by each load and store instruction in the deciles. In other words, the load and store instructions in the higher deciles are more effective at generating prefetches. Note that we provide the number of instructions in each decile to the right of each stacked bar.

For example, only 15 instructions generate 10% of the prefetches for the GO benchmark. The next 10% of the prefetches are generated by 30 instructions. The number in each decile increases swiftly until we reach the tenth decile, which consists of 8634 instructions. The decile breakdowns for the rest of the SPEC benchmarks are similar to that for GO. The very small number of instructions in the STREAM benchmark result in 90% of the prefetches being generated by only 9 instructions.

Figure5.1



It is possible of course that the utility of the prefetches generated in each decile may be quite different. For example, the fact that a large number of prefetches are generated by the relatively few instructions in the first decile is of little importance if a majority of the prefetches generated by them are useless. Similarly, the large number of instructions in the 10th decile may be more important if most of the useful prefetches that are generated are in this decile. To address these concerns we computed the fraction of the total prefetches in each decile that are useful for each benchmark. These fractions are provided in Figure 5.2.

The fraction of the prefetches in each decile that are useful is consistently high for all of the benchmarks. As we have seen in earlier chapters, all of the prefetches generated by STREAM are useful, which manifests itself as a straight-line across the top of Figure 5.2. The rest of the benchmarks are not quite as well behaved, but they are fairly consistent

across the deciles nonetheless, indicating that we can safely assume that the proportion of useful to useless prefetches is uniform across the deciles for all of the benchmarks.

If this was not the case, then we could easily use the compiler to improve runahead performance even more. Suppose, for example, that first few deciles of instructions for a benchmark produce nearly all of the useful prefetches. This implies that the overwhelming bulk of the large number of load and store instructions in the remaining deciles are producing useless prefetches. Using this information we could have the compiler mark the load and store instructions in the useless deciles such that the processor will not generate runahead prefetches for them on a miss during runahead. This could dramatically cut down on the number of useless prefetches that are generated, and lead to a significantly better utilization of the memory hierarchy as well as better performance overall.

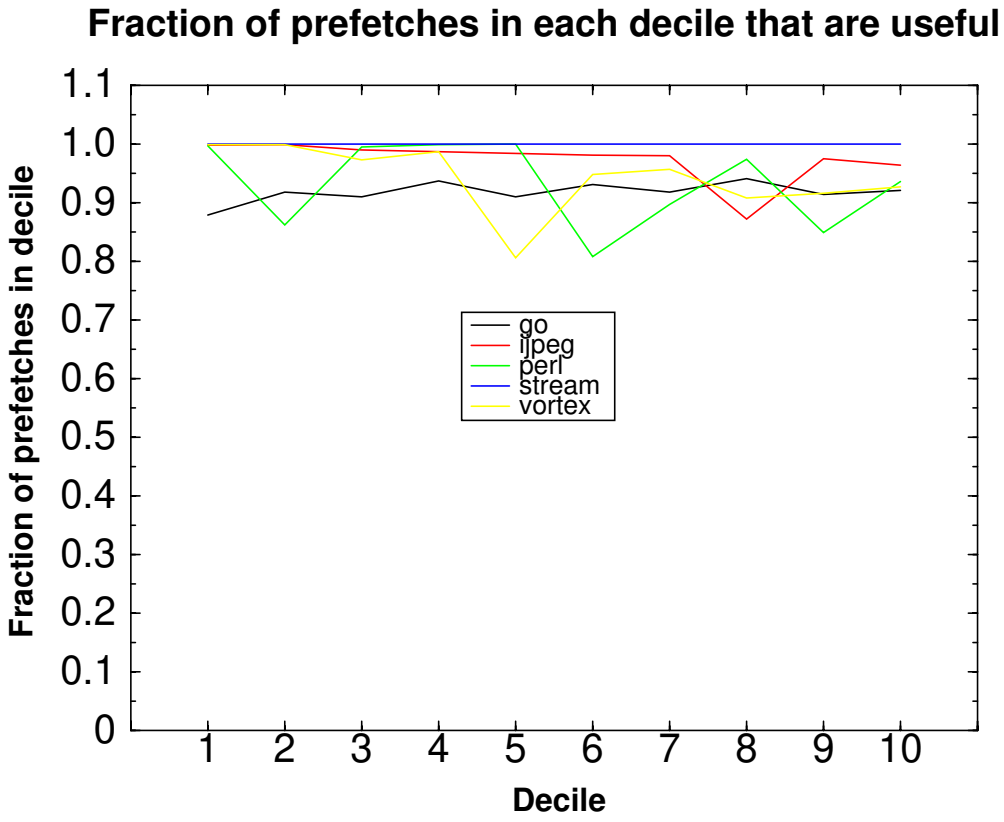


Figure 5.2

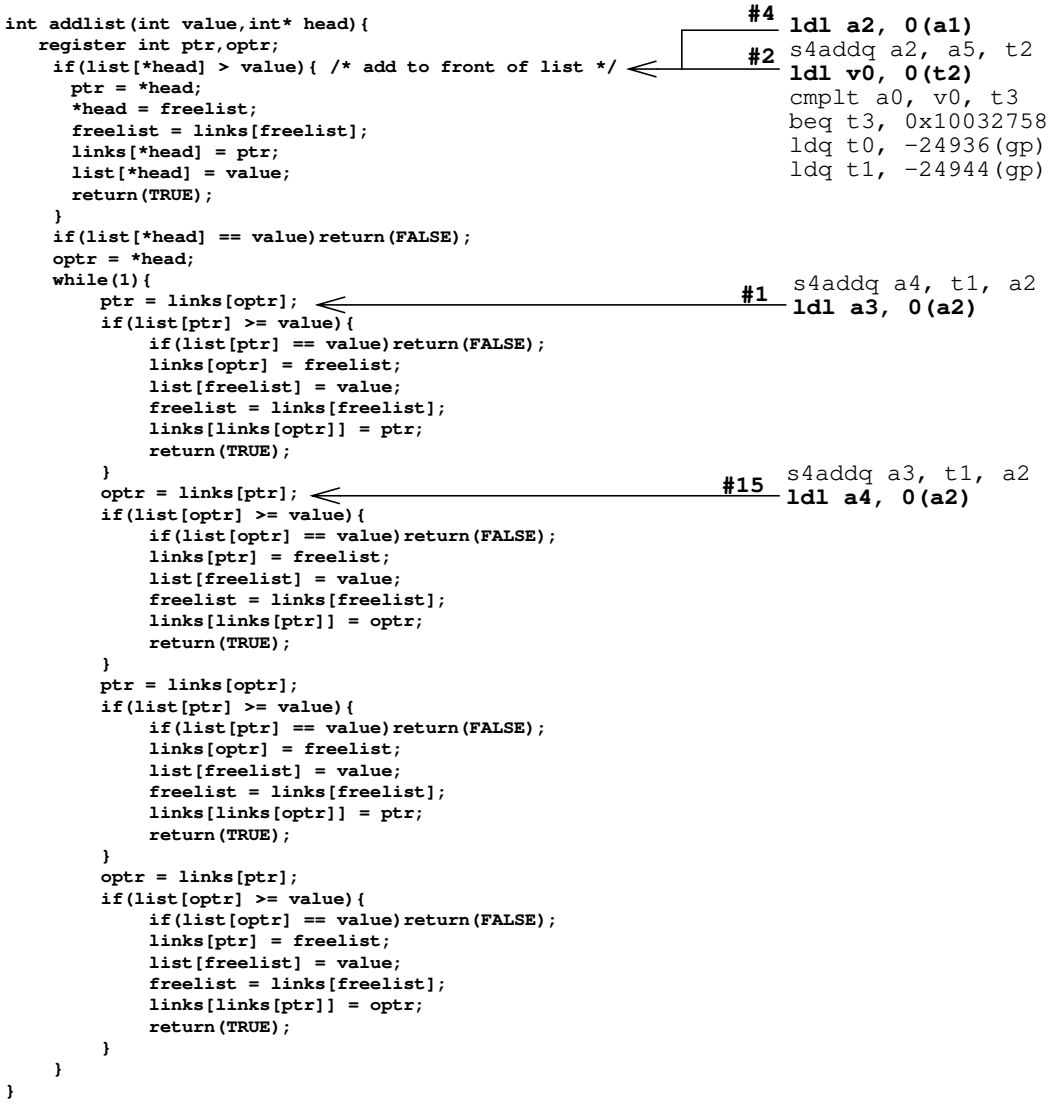
5.1.3 Where are the most useful loads and stores in the source code?

We now present an analysis of the load and store instructions in the top decile of each benchmark. Compiler register usage conventions are provided in Table 5.5, "DEC OSF/1 Alpha Register Usage," on page 166.

The GO benchmark

In the top decile for GO, four of the fifteen instructions in the decile are in the `addlist()` function. This function is shown in Figure 5.3.

Figure 5.3 addlist() function from the GO benchmark




```
int addlist(int value,int* head){
   register int ptr,optr;
    if(list[*head] > value){ /* add to front of list */
      ptr = *head;
      *head = freelist;
      freelist = links[freelist];
      links[*head] = ptr;
      list[*head] = value;
      return(TRUE);
    }
    if(list[*head] == value)return(FALSE);
    optr = *head;
    while(1){
        ptr = links[optr];
        if(list[ptr] >= value){
            if(list[ptr] == value)return(FALSE);
            links[optr] = freelist;
            list[freelist] = value;
            freelist = links[freelist];
            links[links[optr]] = ptr;
            return(TRUE);
        }
        optr = links[ptr];
        if(list[optr] >= value){
            if(list[optr] == value)return(FALSE);
            links[ptr] = freelist;
            list[freelist] = value;
            freelist = links[freelist];
            links[links[ptr]] = optr;
            return(TRUE);
        }
        ptr = links[optr];
        if(list[ptr] >= value){
            if(list[ptr] == value)return(FALSE);
            links[optr] = freelist;
            list[freelist] = value;
            freelist = links[freelist];
            links[links[optr]] = ptr;
            return(TRUE);
        }
        optr = links[ptr];
        if(list[optr] >= value){
            if(list[optr] == value)return(FALSE);
            links[ptr] = freelist;
            list[freelist] = value;
            freelist = links[freelist];
            links[links[ptr]] = optr;
            return(TRUE);
        }
    }
}
```

The third most productive prefetch instruction is in the function `undercut()`, shown in Figure 5.4.

Figure 5.4 undercut() function from the GO benchmark

```
int undercut(int s,int dir,int dir2,int c,int udir,int udir2){
    int lcount = 0,rcount=0,sn,g,uc,cval,dist;
    sn = s-dir;                                                    addq s2, s2, t12
    do {                                                           ldq t5, -28352(gp)
        sn += dir;                                                 s8addq t12, t5, s6
        g = board[sqrbrd[sn][udir]];                               addq s6, t8, t6
        dist = dstbrd[sn][udir];  <-------------------------- #3   ldl s4, 0(t6)
        uc = grcolor[g];
        if((gralive[g] < SMOTHERED || gralive[g] > DEAD) &&
            uc == 1-c && canundercut(sqrbrd[sn][udir],dir2,sn,dist)){
            lcount++;
            }
        if((gralive[g] == SMOTHERED || grthreatened[g]) && uc == c &&
            dist < 2)
            lcount++;
        g = board[sqrbrd[sn][udir2]];
        dist = dstbrd[sn][udir2];
        uc = grcolor[g];
        if((gralive[g] < SMOTHERED || gralive[g] > DEAD) &&
            uc == 1-c && canundercut(sqrbrd[sn][udir2],-dir2,sn,dist)){
            rcount++;
            }
        if((gralive[g] == SMOTHERED || grthreatened[g]) && uc == c &&
            dist < 2)
            rcount++;
        }while(edge[sn] != 0 && (edge[sn] != 1 || edge[sn-dir] != 2));
    cval = 0;
    if(rcount)cval += 4;
    if(lcount)cval += 4;
    return(cval);
}
```

The 5th and 13th most productive prefetch instructions are in the function `del-list()`, shown in Figure 5.5.

Figure 5.5 dellist() function from the GO benchmark

The 9th, 11th, 12th, and 14th most productive prefetches are located in the function `blockuc()`, which is shown in Figure 5.6.

Figure 5.6 blockuc() function from the GO benchmark

```
int blockuc(int army,int s){
    int ptr,ptr2,rlist = EOL,c,crawl,so,sd,i,ldtmp,sn;
    c = grcolor[list[armygroups[army]]];
    if(ltrgd[s] == 4 || ltrgd[s] == 5)
        for(ptr = nblbp[s]; ptr != EOL; ptr = links[ptr]){
            if(edge[list[ptr]] == edge[s] && edge2[s] < 4 &&
               edge2[list[ptr]] < edge2[s])
                findblock(s,list[ptr],&rlist,c);
            if(edge[list[ptr]] >= edge[s])continue;
            findblock(s,list[ptr],&rlist,c);
            }
    else if(ltrgd[s] == 3){
        i = fdir[s];
        for(ldtmp = ldir[i]; i < ldtmp; ++i){
            sn = s + nbr[i];
            if(board[sn] == NOGROUP)continue;
            if(grlibs[board[sn]] != 2)continue;
            for(ptr = grlbp[board[sn]]; ptr != EOL; ptr = links[ptr])
                if(lnbn[list[ptr]] > 1)addlist(list[ptr],&rlist);
            }
        if(lnbf[s][c] == 1 && grlibs[lgr[s]] == 2)
            addlist(s,&rlist);
        for(ptr = nblbp[s]; ptr != EOL; ptr = links[ptr]){
            if(edge[list[ptr]] < edge[s])continue;
            if(lnbf[list[ptr]][1-c] != 0){
                addlist(list[ptr],&rlist);
                if(lnbn[list[ptr]] < 3)
                    addlist(s,&rlist);
                }
            if(lnbn[list[ptr]] == 4)addlist(list[ptr],&rlist);
            }
        }
    if(rlist == EOL){
        if(ld[s] == NEUTRALLD){
            crawl = FALSE;
            so = sd = NOSQUARE;
            for(ptr = nblbp[s]; ptr != EOL; ptr = links[ptr]){
                if(edge[list[ptr]] == edge[s]){
                    so = list[ptr];
                    crawl = TRUE;
                    }
                if(edge[list[ptr]] < edge[s]){
                    sd = list[ptr];
                    }
                }
            if(lnbf[s][c] > 1)addlist(s,&rlist);
            else if(crawl){
                if(lnbn[so] == 3 && lnbf[so][c] == 1 && so != NOSQUARE)
                    addlist(so,&rlist);
                else addlist(s,&rlist);
                }
            else if(sd != NOSQUARE && lnbf[s][c] > 1)addlist(sd,&rlist);
            else addlist(s,&rlist);
            }
        if(grthreatened[lgr[s]])
            addlist(s,&rlist);
        for(ptr = nblbp[s]; ptr != EOL; ptr = links[ptr]){
            if(edge[list[ptr]] > edge[s] &&
               (edge2[s] > 3 || edge2[list[ptr]] != edge2[s]))continue;
            if(lnbf[list[ptr]][1-c] != 0){
                addlist(list[ptr],&rlist);
                if(lnbn[list[ptr]] < 3)
                    addlist(s,&rlist);
                }
            else if(board[list[ptr]+list[ptr]-s] == NOGROUP){
                if(lnbf[list[ptr]+list[ptr]-s][1-c] != 0)
                    addlist(list[ptr],&rlist);
                }
            else for(ptr2 = nblbp[list[ptr]]; ptr2 != EOL; ptr2 = links[ptr2])
                if(edge[list[ptr2]] > edge[s] && lnbn[list[ptr2]] > 2)
                    addlist(list[ptr2],&rlist);
            }
        }
    return(rlist);
}
```

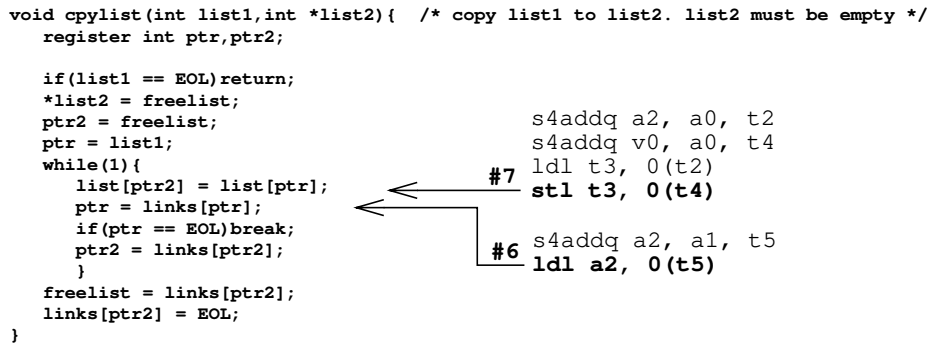
```
lda  sp, -256(sp)
ldah gp, 6(t12)
stq  ra, 0(sp)
lda  gp, -21552(gp)
stq  s0, 8(sp)
stq  s1, 16(sp)
#14 stq  s6, 56(sp)
stq  s2, 24(sp)
stq  s3, 32(sp)
stq  s4, 40(sp)
stq  s5, 48(sp)
addl a0, 0, a0
stq  a1, 216(sp)
ldq  s5, -29448(gp)
```

```
ldq  t5, -29032(gp)
#9 s4addq a0, t5, t6
ldq  t10, -29184(gp)
ldl  t7, 0(t6)
s4addq t7, s5, t8
ldl  t9, 0(t8)
s4addq t9, t10, t11
ldl  t12, 0(t11)
stl  t12, 176(sp)
```

```
ldq  ra, 0(sp)
ldq  s0, 8(sp)
#12 ldq  s1, 16(sp)
ldq  s2, 24(sp)
ldl  v0, 184(sp)
ldq  s3, 32(sp)
#11 ldq  s6, 56(sp)
ldq  s4, 40(sp)
ldq  s5, 48(sp)
lda  sp, 256(sp)
ret  zero, (ra), 1
```

The 6th and 7th most productive prefetches are located in the function `cpylist()`, shown in Figure 5.7.

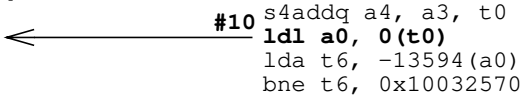
Figure 5.7 cpylist() function from the GO benchmark



The 10th most productive instruction is located in the function `mrglist()`, shown in Figure 5.8.

Figure 5.8 mrglist() function from the GO benchmark

```
/* merge list1 into list2, leaving list1 unchanged.  return
 * number of elements added to list2
 */


int mrglist(int list1, int* list2){
   register int ptr1,ptr2, count,temp,temp2;
   count = 0;
   if(list1 == EOL)return(0);
   if(*list2 == EOL){
      cpylist(list1,list2);
      ptr1 = *list2;
      while(ptr1 != EOL){
        ++count;
        ptr1 = links[ptr1];
        }
      return(count);
      }

   if (list[list1] < list[*list2]) {
      temp = *list2;
      *list2 = freelist;
      freelist = links[freelist];
      links[*list2] = temp;
      list[*list2] = list[list1];
      ptr1 = links[list1];
      ++count;
      }
   else if(list[list1] == list[*list2])
      ptr1 = links[list1];
   else ptr1 = list1;

   ptr2 = *list2;

   while(ptr1 != EOL){

/* guaranteed that list[ptr1] > list[ptr2] */

      if(links[ptr2] == EOL){
         links[ptr2] = freelist;
         while(ptr1 != EOL){
            list[freelist] = list[ptr1];
            ptr2 = freelist;
            freelist = links[freelist];
            ptr1 = links[ptr1];
            ++count;
            }
         links[ptr2] = EOL;
         return(count);
         }

      temp2 = list[links[ptr2]];
      if(list[ptr1] < temp2){
         temp = links[ptr2];
         links[ptr2] = freelist;
         ptr2 = freelist;
         freelist = links[freelist];
         links[ptr2] = temp;
         list[ptr2] = list[ptr1];
         ++count;
         ptr1 = links[ptr1];
         }
      else if(list[ptr1] == temp2)
         ptr1 = links[ptr1];
      else ptr2 = links[ptr2];
      }

   return(count);
   }
```



The 8th most productive instruction is located in the function `findshapes()`, shown in Figure 5.9.

Figure 5.9 findshapes() function from the GO benchmark

```
void findshapes(int fsqr,int lsqr){
    int s,sh;
    int top; /* start at this point */
    int bot; /* stop when get past this point */
    int width; /* width of rectangle */
    int right; /* right edge of rectangle */
    int color, point,t,b,l,r,left,up;
    t = b = l = r = TRUE;  /* do all patterns at top, bottom, etc */
    for(sh = 0; sh < numshapes; ++sh){
       if(shapes[sh].xsize > boardsize || shapes[sh].ysize > boardsize)
           continue;
       bot = lsqr;
       left = xval[fsqr] - shapes[sh].xsize + 1;
       if(left < 0)left = 0;
       up = yval[fsqr] - shapes[sh].ysize + 1;
       if(up < 0)up = 0;
       top = up * boardsize + left;

       if(xval[bot] + shapes[sh].xsize > boardsize)
           bot -= xval[bot] + shapes[sh].xsize - boardsize;
       if(yval[bot] + shapes[sh].ysize > boardsize)
           bot -= boardsize * (yval[bot] + shapes[sh].ysize - boardsize);
       right = xval[bot];
       width = right - xval[top] + 1;
       t = yval[top] == 0;
       b = yval[bot] == boardsize-shapes[sh].ysize;
       l = xval[top] == 0;
       r = xval[bot] == boardsize-shapes[sh].xsize;            ldq t9, -22840(gp)
       color = vclr[values[shapes[sh].startpoint]];        #8  s4addq a0, t9, t12
       point = points[shapes[sh].startpoint];  <-----------  ldl t11, 0(t12)
       switch(shapes[sh].where){                               stl t11, 192(sp)
        case ANYWHERE:
        for(s = top; s <= bot; ++s){
            if(xval[s] > right)s += boardsize - width;
            if(grcolor[board[s+point]] == color && match(sh,s))
                addlist(sh,&shapebrd[s]);
            else if(grcolor[board[s+point]] == 1-color &&
                match2(sh,s))
                addlist(sh,&shapebrd[s]);
            else if(shapebrd[s] != EOL)
                dellist(sh,&shapebrd[s]);
            }
        break;
        case TOP:
        if(t)for(s = top; s < top + width; ++s)
            if(grcolor[board[s+point]] == color && match(sh,s))
                addlist(sh,&shapebrd[s]);
            else if(grcolor[board[s+point]] == 1-color &&
                match2(sh,s))
                addlist(sh,&shapebrd[s]);
            else if(shapebrd[s] != EOL)
                dellist(sh,&shapebrd[s]);
        break;

                (DELETIA)

        case RIGHT:
        if(r)for(s = bot; s >= top; s -= boardsize){
            if(grcolor[board[s+point]] == color && match(sh,s))
                addlist(sh,&shapebrd[s]);
            else if(grcolor[board[s+point]] == 1-color &&
                match2(sh,s))
                addlist(sh,&shapebrd[s]);
            else if(shapebrd[s] != EOL)
                dellist(sh,&shapebrd[s]);
            }
        break;
        }
    }
}
```

The IJPEG Benchmark

The IJPEG benchmark is rather different from most of the benchmarks that we have examined in that prefetch generation is largely confined to a very small section of the code. Virtually all of the instructions in the top 7 deciles are confined to the bracketed lines of source code in the function `rgb_ycc_convert()`, shown in Figure 5.10.

Figure 5.10 rgb_ycc_convert() function from the IJPEG benchmark

```
METHODDEF void
rgb_ycc_convert (j_compress_ptr cinfo,
          JSAMPARRAY input_buf, JSAMPIMAGE output_buf,
          JDIMENSION output_row, int num_rows)
{
  my_cconvert_ptr cconvert = (my_cconvert_ptr) cinfo->cconvert;
  register int r, g, b;
  register INT32 * ctab = cconvert->rgb_ycc_tab;
  register JSAMPROW inptr;
  register JSAMPROW outptr0, outptr1, outptr2;
  register JDIMENSION col;
  JDIMENSION num_cols = cinfo->image_width;

  while (--num_rows >= 0) {
    inptr = *input_buf++;
    outptr0 = output_buf[0][output_row];
    outptr1 = output_buf[1][output_row];
    outptr2 = output_buf[2][output_row];
    output_row++;
    for (col = 0; col < num_cols; col++) {
      r = GETJSAMPLE(inptr[RGB_RED]);
      g = GETJSAMPLE(inptr[RGB_GREEN]);
      b = GETJSAMPLE(inptr[RGB_BLUE]);
      inptr += RGB_PIXELSIZE;
      /* If the inputs are 0..MAXJSAMPLE, the outputs of these equations
       * must be too; we do not need an explicit range-limiting operation.
       * Hence the value being shifted is never negative, and we don't
       * need the general RIGHT_SHIFT macro.
       */
      /* Y */
      outptr0[col] = (JSAMPLE)
         ((ctab[r+R_Y_OFF] + ctab[g+G_Y_OFF] + ctab[b+B_Y_OFF])
          >> SCALEBITS);
      /* Cb */
      outptr1[col] = (JSAMPLE)
         ((ctab[r+R_CB_OFF] + ctab[g+G_CB_OFF] + ctab[b+B_CB_OFF])
          >> SCALEBITS);
      /* Cr */
      outptr2[col] = (JSAMPLE)
         ((ctab[r+R_CR_OFF] + ctab[g+G_CR_OFF] + ctab[b+B_CR_OFF])
          >> SCALEBITS);
    }
  }
}
```

Top 39 most productive instructions

(~70% of prefetches)

The PERL Benchmark

Most of the instructions in the top deciles of the PERL benchmark are in the libc code. Unfortunately, we did not have access to the libc source code. Note that 30% of the prefetches are generated within the `malloc()` function (Table 5.3).

The VORTEX Benchmark

The first decile of prefetches for VORTEX are generated by two instructions that are in the libc `memset()` function. Unfortunately, we did not have access to the libc source code. However, the second decile of prefetches are generated by two instructions that are in the `Chunk_ChkGetChunk()` function, which is shown in Figure 5.11.

Figure 5.11 Chunk_ChkGetChunk() function from the VORTEX benchmark

```
boolean    ChkGetChunk (numtype     ChunkNum,    indextype     Index,
                        size_t      SizeOfUnit,  ft F,lt Z,zz *Status)
{
indextype  StackPtr = 0;

  Mem_ChunkExpanded               = 0;
  if (ChunkExists (ChunkNum,McStat))
  {
     if (Chunk_IsData        (ChunkNum)
    // macro = ((Theory->Flags[ChunkNum] & Is_Data))

     &&  Chunk_IsNumeric     (ChunkNum)
     && (size_t)Unit_Size    (ChunkNum)   == SizeOfUnit)
     {
       StackPtr = Stack_Ptr  (ChunkNum);
       if (Index   >= StackPtr)
       {
          if (SetGetSwi)
            *Status    = Set_EndOfSet;

          else {
             DumpChunkChunk (0, ChunkNum);
            *Status    = Err_IndexOutOfRange;
          }
       }

     } else {
        if ((size_t)Unit_Size     (ChunkNum) != SizeOfUnit)
        {
           *Status = Err_BadUnitType;

        } else {
           *Status = Err_BadChunk;
        }
        DumpChunkChunk (0, ChunkNum);
     }
  }

TRACK(TrackBak,"ChkGetChunk\n");
return(STAT);
}
```

```
0x100441e4:ldq    v0, -19216(gp)
0x100441c8:zap    t0, 0xf0, t2
0x100441d0:ldq    v0, 0(v0)
0x100441d4:ldq    t3, 16(v0)
0x100441d8:s4addq t2, t3, t4
0x100441e0:ldl    t1, 0(t4)
0x100441e4:and    t1, 0x8, t5
0x100441e8:beq    t5, 0x10044260
0x100441ec:and    t1, 0x40, t6
0x100441f0:beq    t6, 0x10044260
0x100441f4:ldq    t8, 0(v0)
0x100441f8:zap    t0, 0xf0, t7
0x100441fc:s8addq t7, t8, t9
#4 0x10044200:ldq    a0, 0(t9)
#3 0x10044204:ldl    t10, -28(a0)
0x10044208:cmpeq  a2, t10, t11
0x1004420c:beq    t11, 0x10044260
```

The STREAM benchmark

As we mentioned earlier the STREAM benchmark is quite small, and nearly all of the prefetches are generated by only nine instructions. Because of this we have reproduced here the disassembled code that makes up the `main()` function of STREAM. The size of the code requires us to break it up into two sections in Figures 5.12 and 5.13. Each of the instructions in the figures has its source code line number, PC, and mnemonic. They also have their ranking in terms of the number of prefetches that they generate, with #1 indicating the instruction that generates the most prefetches, #2 the next most prolific instruction, etc. Note that this is only done for the top 20 prefetch-generating instructions. It should also be noted that the top 9 instructions each generate approximately 10% of the total number of prefetches.

Figure 5.12 First portion of the main () function from the STREAM benchmark

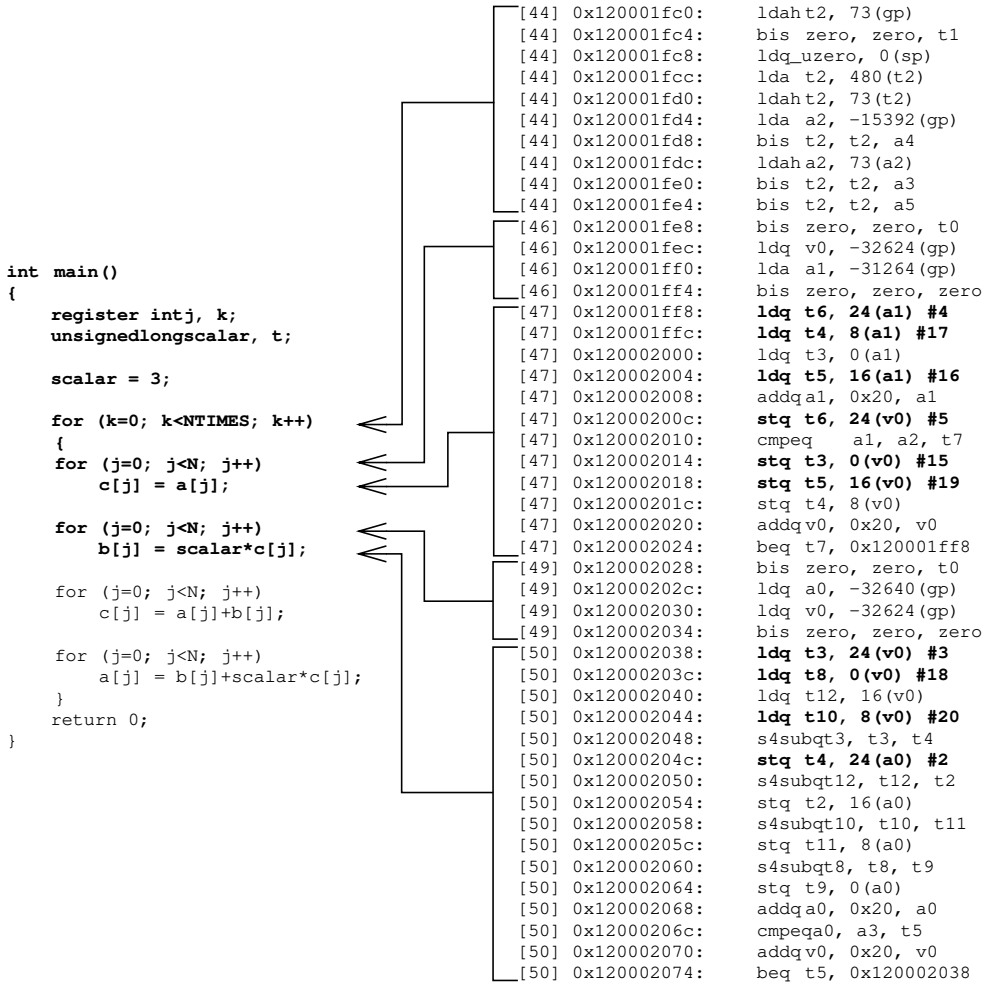
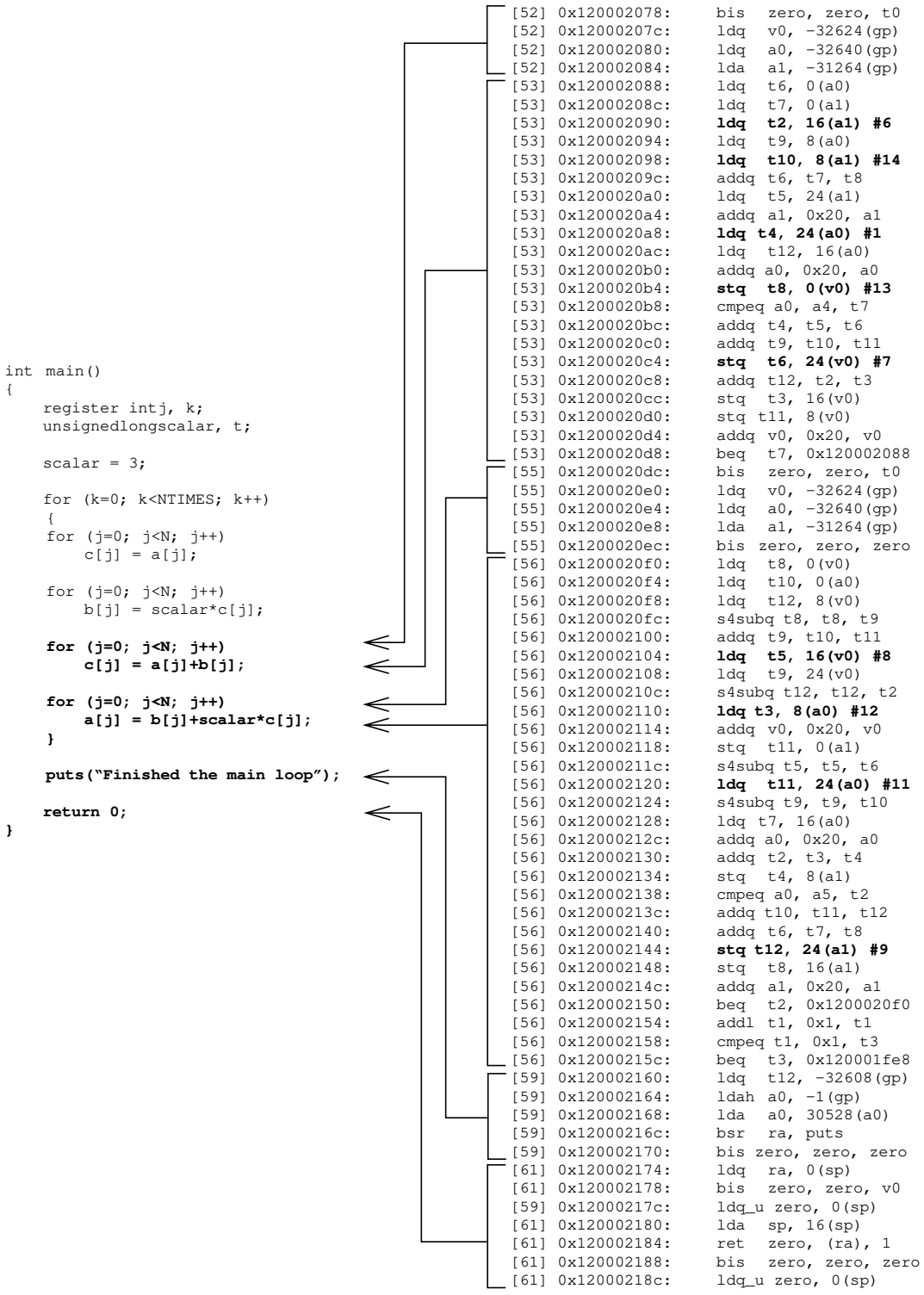


Figure 5.13 Second portion of the main() function from the STREAM benchmark



5.2 Register file effects

We were also interested in finding out which registers and instructions were used during runahead episodes to generate prefetches and to keep the processor on the right path. If the processor typically only uses a small subset of the instruction set and register file during runahead, then it may be practical to design a very simple runahead co-processor that can pre-process the instruction stream in parallel with a more advanced processor that actually executes applications.

In our preliminary runahead studies described in [35] we presented some early register file statistics. Unfortunately, we only measured the average number of VALID registers that were in the register file over the course of the average runahead episode. While this data was interesting, it left quite a bit to be desired as one would expect that some registers would be more useful during runahead than others.

We added code to the simulator that examines all of the instructions in a given runahead episode every time that a register value is used to perform a task of interest. Every instruction in a given runahead episode is saved in a linked list. When a register is used to generate a prefetch address, for example, the simulator scans backwards through the linked list to determine which instructions and registers contributed to computing the that register's value. When we find an instruction that computes a value that is needed to compute this final result, we record the instruction type and destination register number in an array:

```
prefetch_regs[INSTR_TYPE][INSTR_DEST_REG]++;
```

Similar counters are used to record the instruction and register information used to resolve conditional and indirect branches. In order to keep the statistics meaningful, we divide the ALPHA instructions that can update registers into several classes:

- load
- load-address (these compute an address and place it in the destination register)
- unconditional_branch (the return address is written to a register)
- indirect_branch (the return address is written to a register)
- add_sub (add, sub, etc.)
- compare (cmpeq, cmple, etc.)
- logical (and, or, eqv, etc.)
- conditional_move (cmoveq, etc.)
- byte_manipulation (shift, mask, insert, extract, zap, etc.)
- multiply

This approach captures register values that are computed during runahead episodes, however many register values used during runahead are computed during normal operation. These values are covered by adding a non_runahead instruction category.

Suppose, for example, that we have a runahead episode as depicted in Figure 5.14. In this example runahead is initiated with a load miss (instruction a). After preprocessing a number of instructions, the processor generates a data stream prefetch using register r3 after detecting a load miss (instruction h). When the prefetch is generated the simulator scans backwards through the linked list containing the instructions in the runahead episode up to that point to determine how the value in register r3 was computed. The immediately preceding instruction (g) is a load-address instruction that computes r3 using r4 as an operand. The simulator records this information by incrementing prefetch_regs[load_address][3]. At this point the simulator needs to find the source of the register r4, which as it turns out is the destination of instruction f, an add instruction. The simulator then increments prefetch_regs[add_sub][4]. Instruction f needs r1 and r2 as sources, which are provided by instructions e and d respectively. This process continues until the simulator finds the runahead initiating instruction, which is the first instruction in the linked list containing the

runahead instructions. At this point the register sources r5, r6, and r7 are still unaccounted for. As these register values were not computed during the runahead episode, they must have been computed during normal operation. The simulator records this by incrementing prefetch_regs[non_runahead][5], prefetch_regs[non_runahead][6], and prefetch_regs[non_runahead][7]. Note that r31 is hard-wired to the value zero. We do not trace uses of r31 for this reason. Compiler register usage conventions are provided in Table 5.5.

Figure 5.14

Normal Operation	Instr		Comments	Counter action
		. . .		
		. . .	still need r5 r6 r7	prefetch_regs[non_runahead][5, 6, 7]++
Runahead Initiated	**a**	**load r0, 0(r2)**	load miss	
	b	**load r9, 0(r1)**	(not an r3 source)	
	c	**and r4, r6, r7**	found r4, need r5 r6 r7	prefetch_regs[logical][4]++
	d	**mult r2, r6, r7**	found r2, need r4 r5 r6 r7	prefetch_regs[multiply][2]++
	e	**sll r1, r4, r5**	found r1, need r2 r4 r5	prefetch_regs[byte_manip][1]++
	f	**add r4, r1, r2**	found r4, need r1 r2	prefetch_regs[add_sub][4]++
	g	**lda r3, 32(r4)**	found r3, need r4	prefetch_regs[load_address][3]++
	h	**load r21, 32(r3)**	need to find r3 sources	
Prefetch Generated				

Table 5.5 DEC OSF/1 Alpha Register Usage

Register Name	Software Name	Use and Linkage
r0	v0	Used for expression evaluations and to hold integer function results.
r1 .. r8	t0 .. t7	Temporary registers; not preserved across function calls.
r9 .. r14	s0 .. s5	Saved registers; their values must be preserved across function calls.
r15	FP or s6	Frame pointer or a saved register.
r16 .. r21	a0 .. a5	Argument registers; used to pass the first 6 integer type arguments; their values are not preserved across procedure calls.
r22 .. r25	t8 .. t11	Temporary registers; not preserved across procedure calls.
r26	ra	Contains the return address; used for expression evaluation.
r27	pv or t12	Procedure value or a temporary register.
r28	at	Assembler temporary register; not preserved across procedure calls.
r29	GP	Global pointer.
r30	SP	Stack pointer.
r31	zero	Always has the value 0.

From "*Alpha Architecture Reference Manual*," Part III p 1-1,
Digital Equipment Corporation 1992.

5.2.1 Data stream prefetch address computation

The technique described in the previous section was used to record the sources of data stream prefetch addresses. We have chosen to present these statistics using 3D bar charts, which are shown in figures 5.15 through 5.19. The height of each bar in the figures represents the number of times that an instruction from a particular class (x-axis) wrote a value to its destination register (y-axis) that was used to compute a load or store address that generated a data stream prefetch.

The prefetch register statistics for the GO benchmark are shown in Figure 5.15. Note that for the GO benchmark most of the register values are computed by either load-address instructions during runahead or by non-runahead instructions. These load-address and non-runahead instructions tend to write to either r29 (the stack pointer) or r30 (the global

pointer). Virtually all of the remaining register values are provided by add_sub, load, or non_runahead instructions.

The VORTEX benchmark register usage, shown in Figure 5.16, is similar to that of the GO benchmark. Many of the register values used are produced by non_runahead instructions, of which VORTEX tends to use the stack pointer (r30) by far the most. Many of the values that are computed during runahead are load-address instruction updates of either the global pointer (r29) or the stack pointer (r30).

The PERL benchmark prefetch register usage, shown in Figure 5.17 is similar to that for GO and VORTEX. Many of the register values used are produced by non-runahead instructions, with the stack (r30) and global (r29) pointers being by far the most updated registers.

The register usage for the IJPEG benchmark, shown in Figure 5.18, is rather different than that for GO and VORTEX. By far the bulk of the prefetch addresses are computed with register values that are not modified during runahead. Of these registers, the one that is employed the most is r16, an argument register. Other non-runahead updated registers that see frequent use are the integer function result register (r0), saved registers (r9-r13), argument registers (r17-r21), return address register (r26), and the stack pointer (r30).

The register usage for the STREAM benchmark, shown in Figure 5.19, uses far fewer instruction and register resources than the other benchmarks. About two-thirds of the prefetch calculations are performed with add-subtract instructions that update either a temporary (r16 and r17), or r0, which in this case is used to hold expression results instead of function results. The remainder of the register values are obtained from non-runahead instructions that update the same registers. Note that it is easy to verify the information in Figure 5.19 by examining the STREAM instructions in Figures 5.12 and 5.13.

Figure 5.15 Source of Data Stream Prefetch Addresses for GO

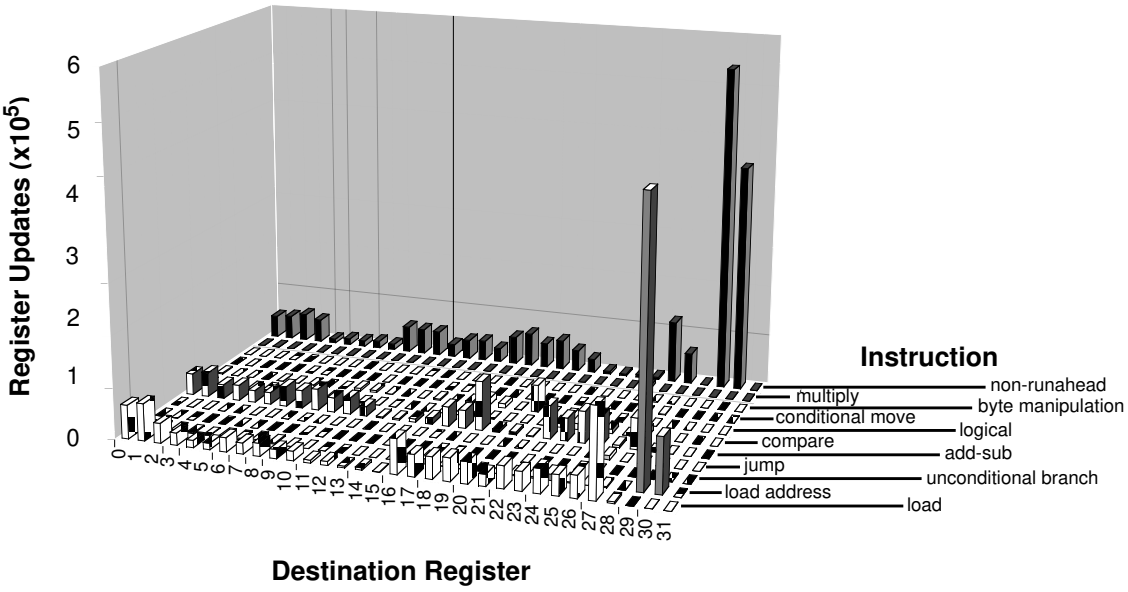


Figure 5.16 Source of Data Stream Prefetch Addresses for VORTEX

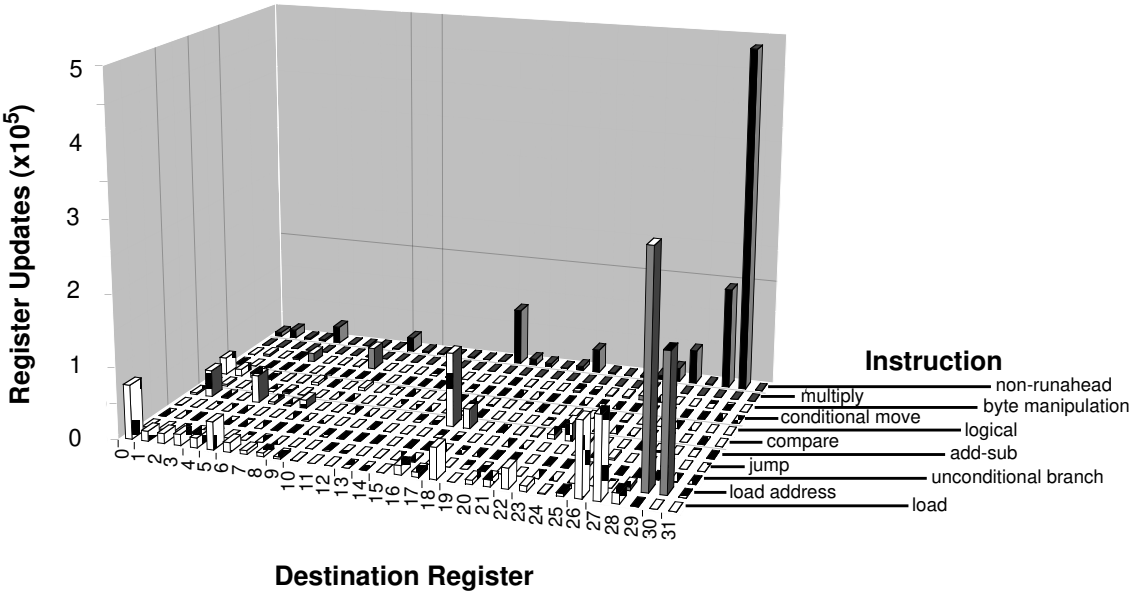


Figure 5.17 **Source of Prefetch Addresses for PERL**

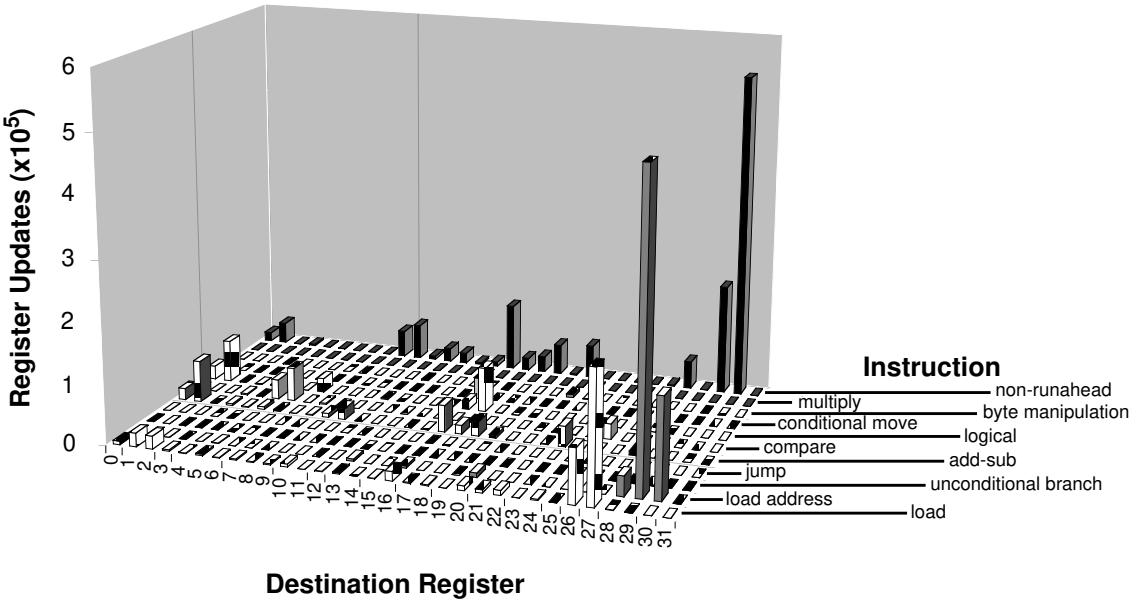


Figure 5.18 **Source of Data Stream Prefetch Addresses for IJPEG**

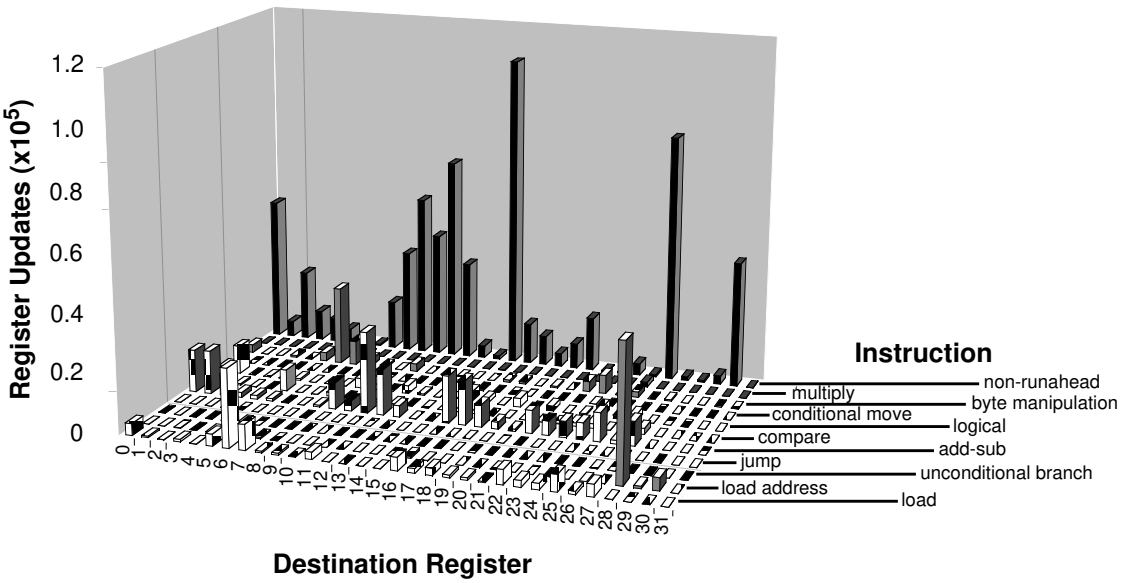
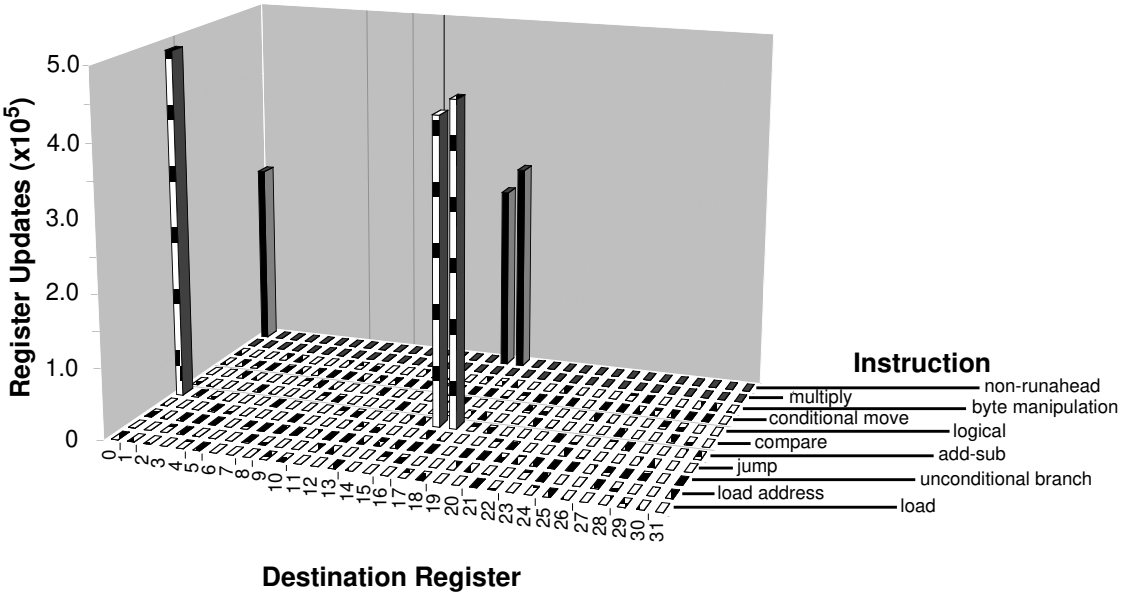


Figure 5.19 Source of Data Stream Prefetch Addresses for STREAM



5.2.2 Branch condition computation

The branch condition source registers for the GO benchmark are shown in Figure 5.20. Note that most of the branch conditions are computed using runahead loads or non-runahead instructions. The remaining values are supplied primarily by add_sub, compare, and load_address instructions. The rest of the SPEC benchmarks use a similar mix of instruction types, with some variation in the registers used. These results are provided in Figures 5.21 through 5.23. Note that IJPEG uses relatively few loads during runahead to compute branch conditions. The correspondingly fewer load misses during runahead increase the likelihood that branches can be resolved with VALID registers, which in turn makes it more likely that the processor will stay on the proper path during runahead.

The branch conditions for the STREAM benchmark are shown in Figure 5.24. As with the prefetch registers, the branch condition calculations require relatively few registers. Add_sub and compare instructions perform most of the work, while the remaining values

are provided by non_runahead instructions. The information in Figure 5.24 can be verified by examining the instructions in Figures 5.12 and 5.13.

Figure 5.20 Source of Branch Conditions for GO

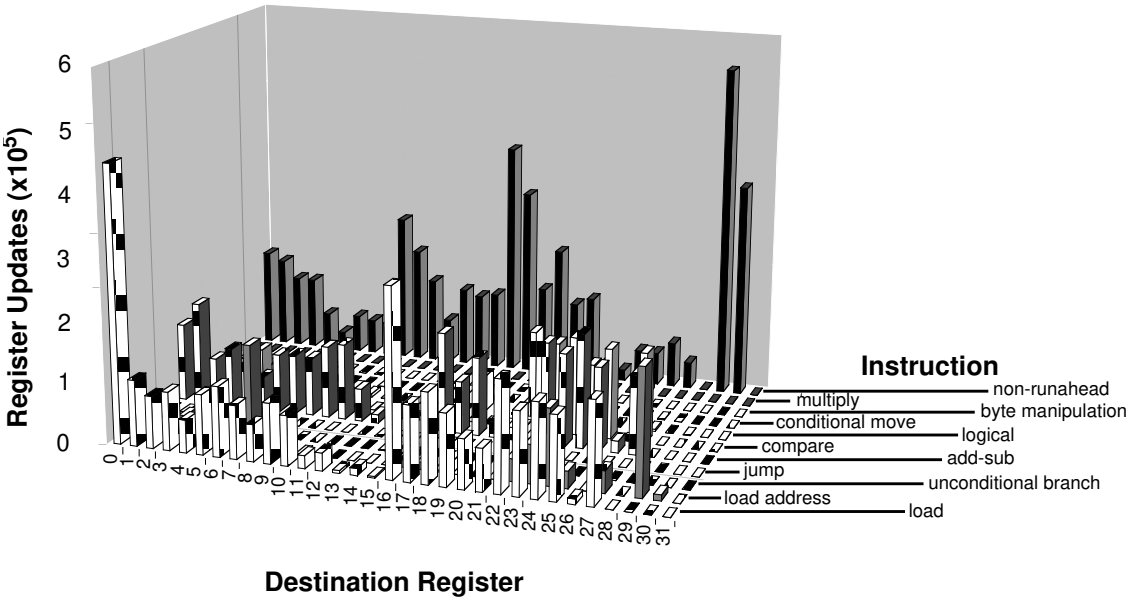


Figure 5.21 **Source of Branch Conditions for VORTEX**

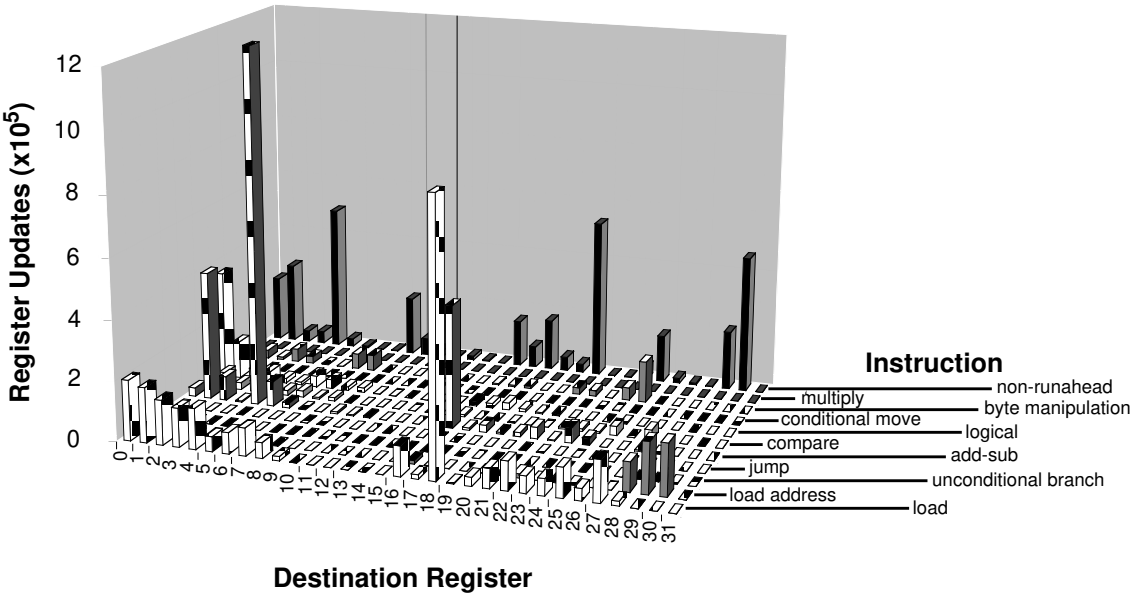


Figure 5.22 **Source of Branch Conditions for PERL**

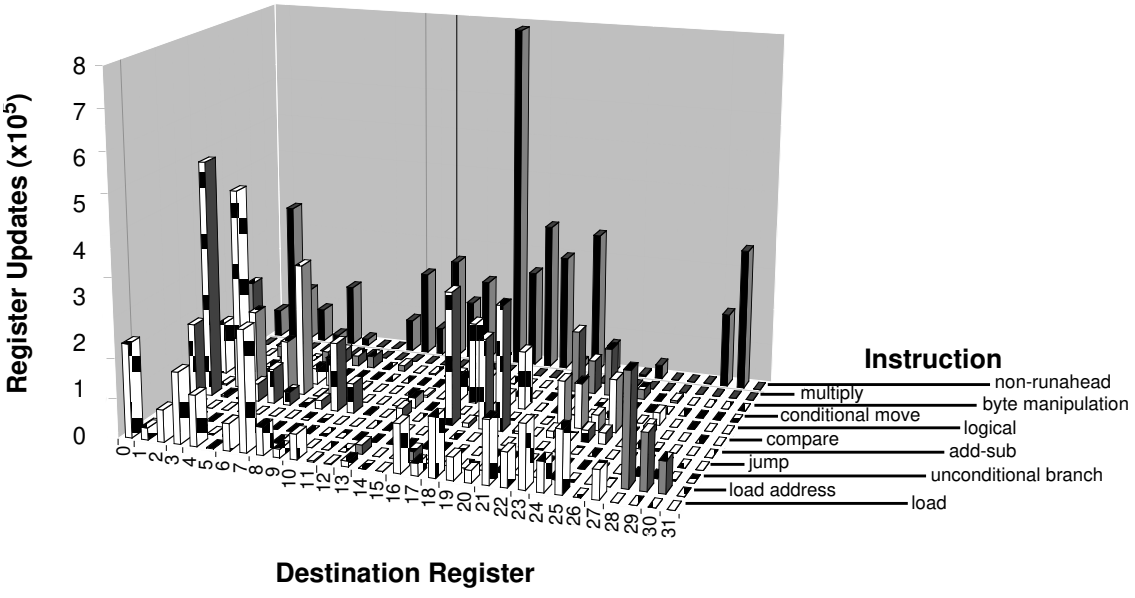


Figure 5.23 **Source of Branch Conditions for IJPEG**

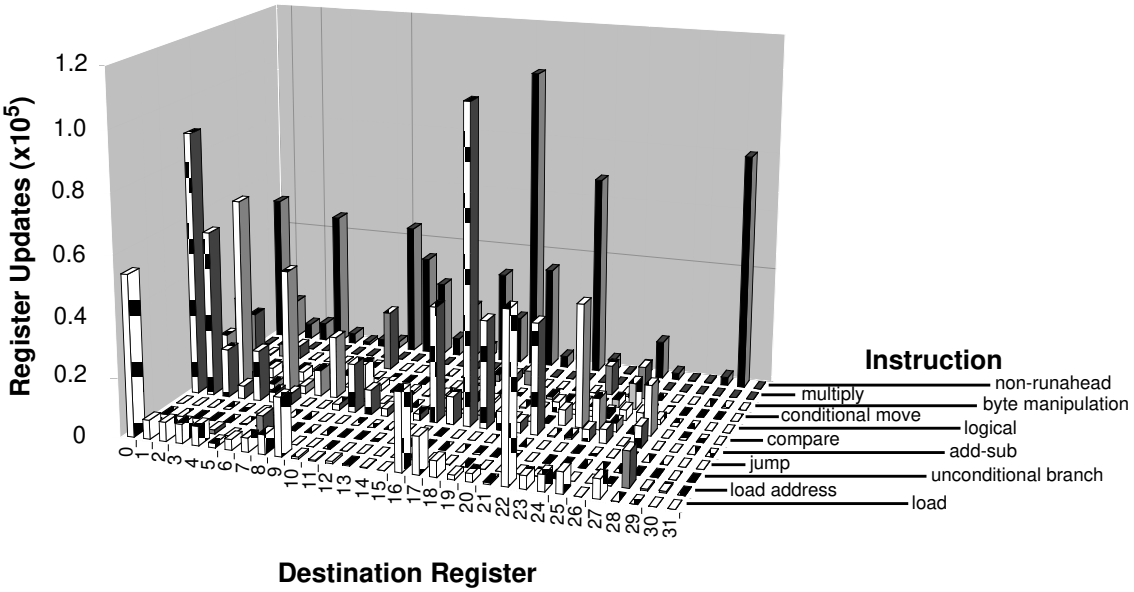
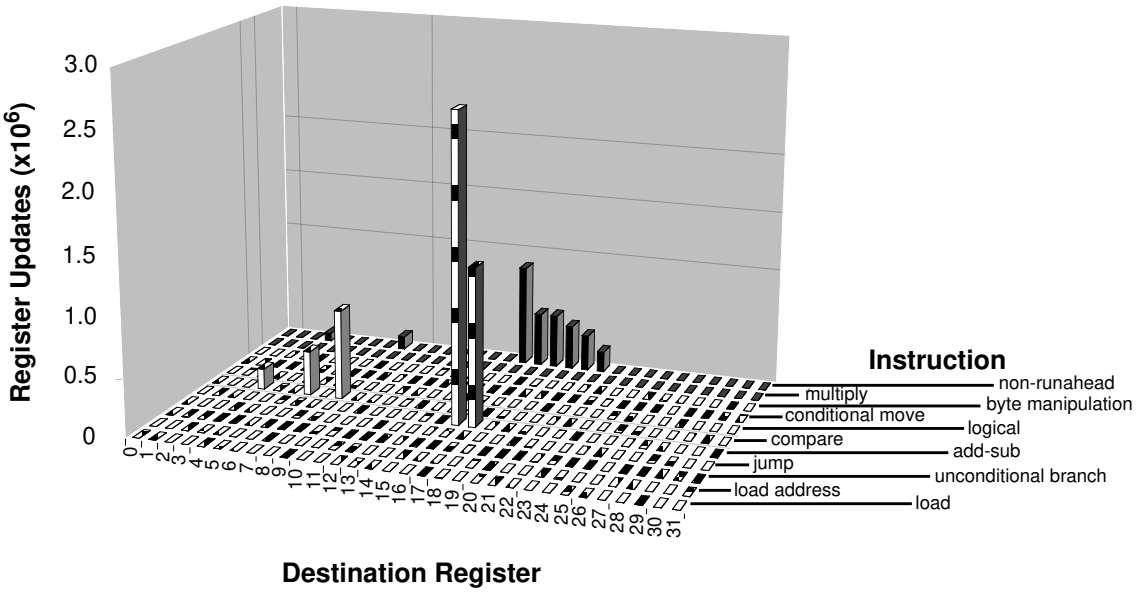


Figure 5.24 **Source of Branch Conditions for STREAM**



5.2.3 Indirect branch target computation

Measurements of the instructions and registers that were used to compute indirect branch targets were performed, the results of which are shown in Figures 5.25 through 5.28. The STREAM benchmark is not shown as it does not contain any indirect branches. The behavior of the GO benchmark, shown in Figure 5.25, is representative of all of the SPEC benchmarks. Load instructions that update r26 (the return address register) or r27 (integer function result register) provide the bulk of the values that are computed during runahead episodes. These are supplemented by non-runahead instructions that provide values already in r26 and r27 at the initiation of runahead. The remainder of the values are largely obtained using r29 (global pointer) or r30 (stack pointer).

Figure 5.25 **Source of Indirect Branch Targets for GO**

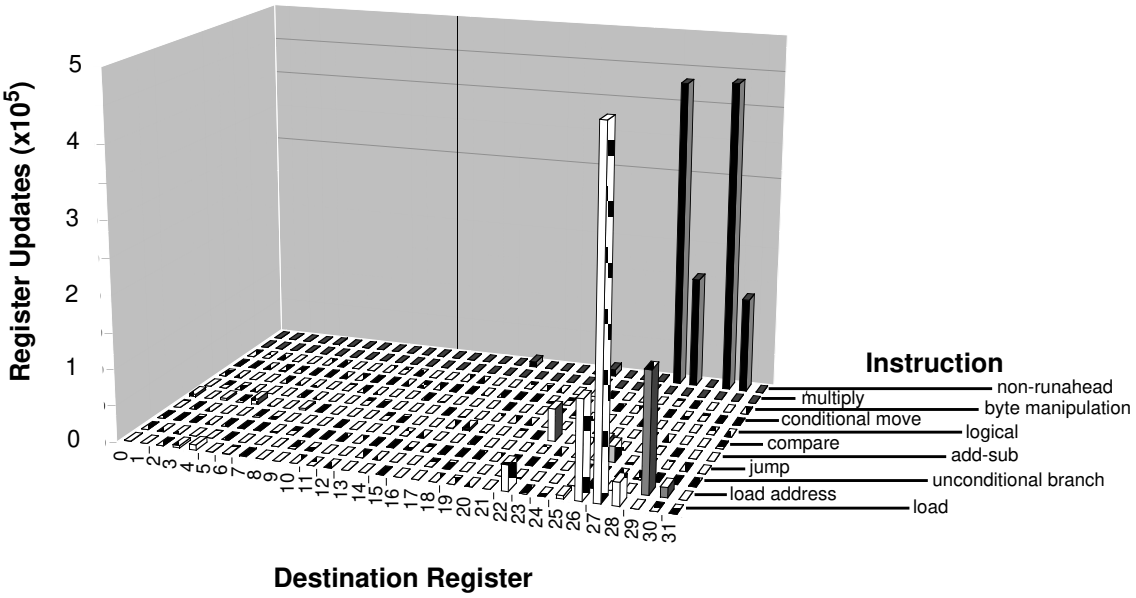


Figure 5.26 **Source of Indirect Branch Targets for PERL**

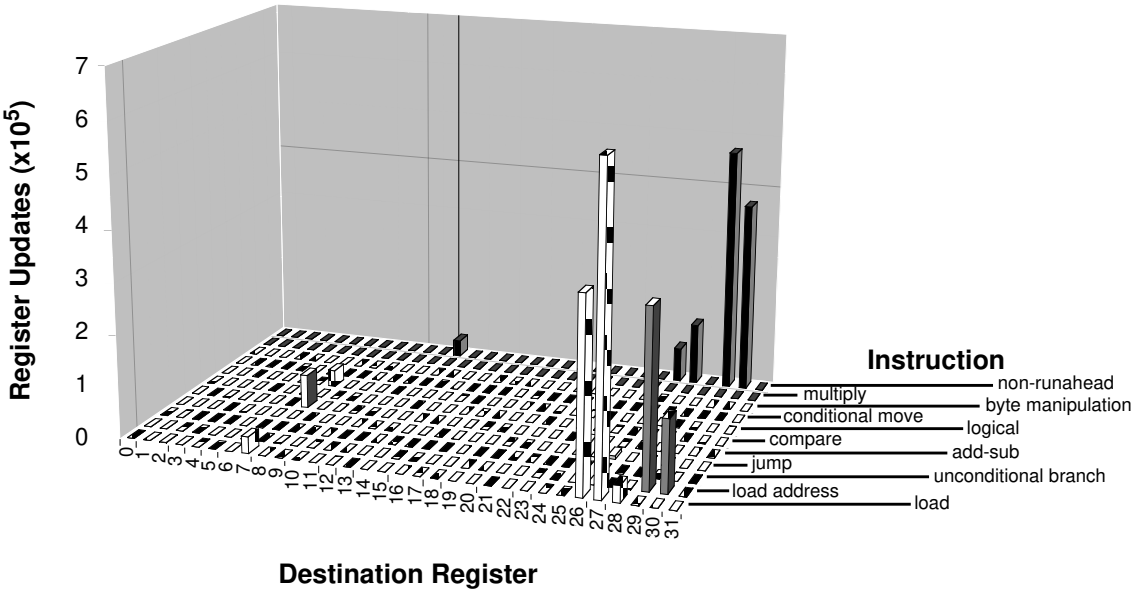


Figure 5.27 **Source of Indirect Branch Targets for VORTEX**

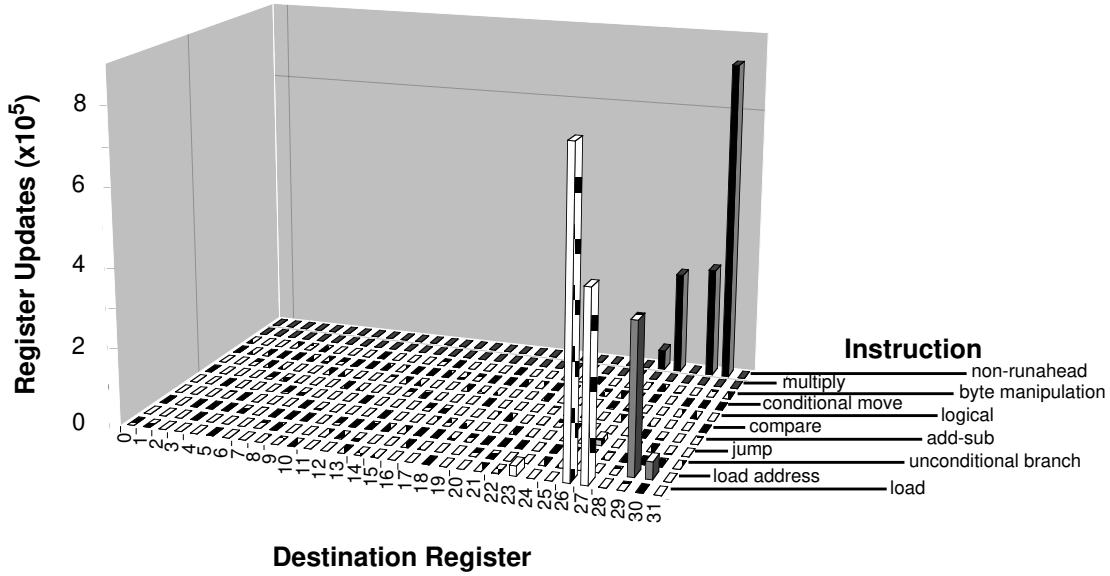
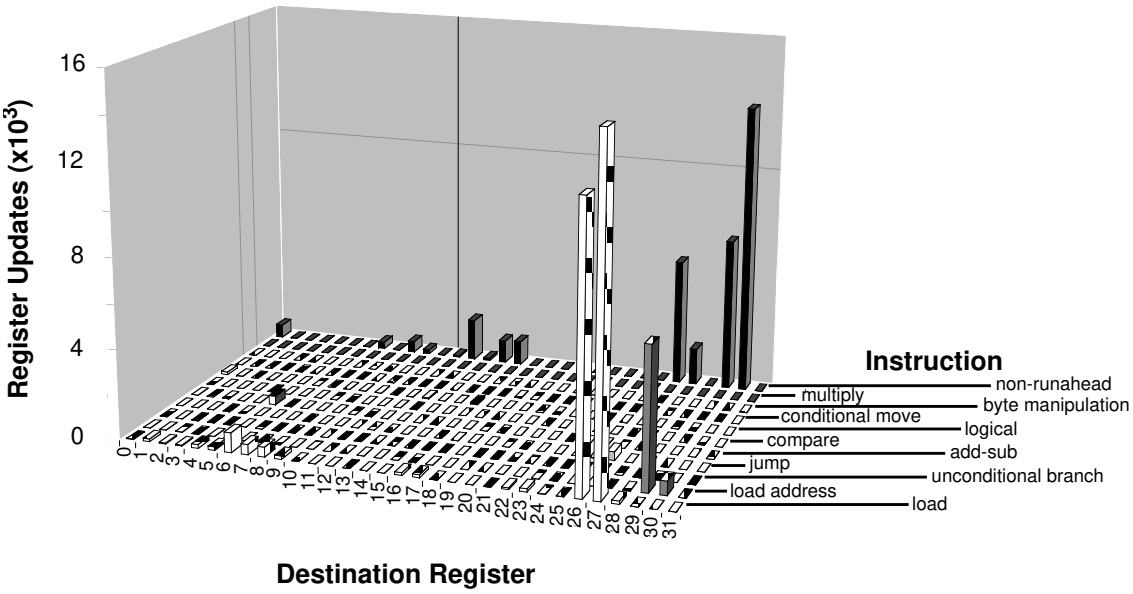


Figure 5.28 **Source of Indirect Branch Targets for IJPEG**



5.2.4 Data stream prefetch base address registers

The data stream prefetch address information provided in Figures 5.15 through 5.19 do not provide a complete picture of how prefetch addresses are formed. Instead, they provide a global view of how prefetch addresses are computed. This can confuse the issue as the 3-D plots place all registers which contribute towards address computation on an equal footing, in which load and store prefetch base address registers are mixed in with other registers that are only indirectly involved in the computation.

We addressed these concerns by recording the number of times each register was used as a runahead load or store prefetch base address register. The counts for each register in the register file is presented in a series of stacked bar charts shown in Figures 5.29 through 5.38. These bars are divided up into usage counts for both useless and useful prefetches.

It is apparent from the plots that the global (r29) and stack pointers (r30) are typically the most commonly used base registers for load prefetches during runahead. The stack pointer is the most commonly used base register for store prefetches, while the global pointer is never used for stores in the benchmarks. These results indicate that the stack and global pointers are the most commonly used base registers for runahead prefetches overall, and that they are not merely used indirectly for prefetch address computation. Registers other than the global and stack pointers are used to hold pointer and array addresses.

From the plots, and from our examination of the benchmark source code, it appears that for most of the benchmarks prefetches are generated for an approximately equal mix of array/pointer, global, and stack references. The IJPEG and STREAM benchmarks are an exception to this rule, as most of their prefetches are generated for array references. The load and store instructions that generate the most prefetches per instruction (i.e. the most efficient instructions) tend to generate prefetches for array or pointer accesses. The global and stack prefetches are generated by less efficient instructions, however this low efficiency per instruction is offset by the large number of instructions in this category.

Figure 5.29

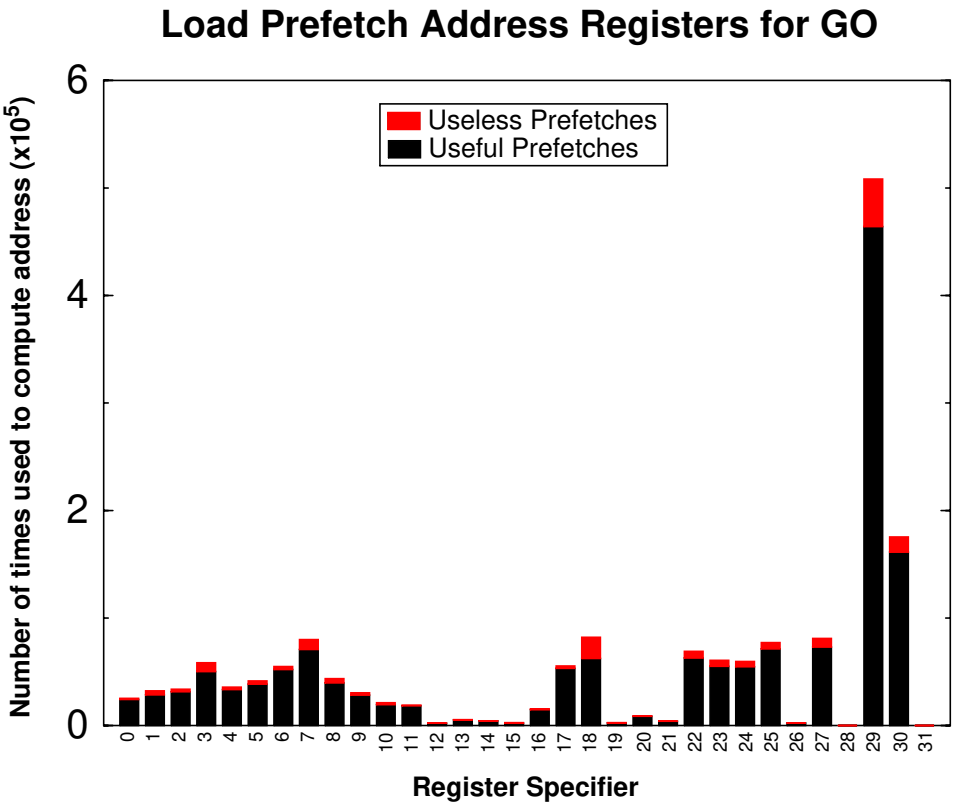
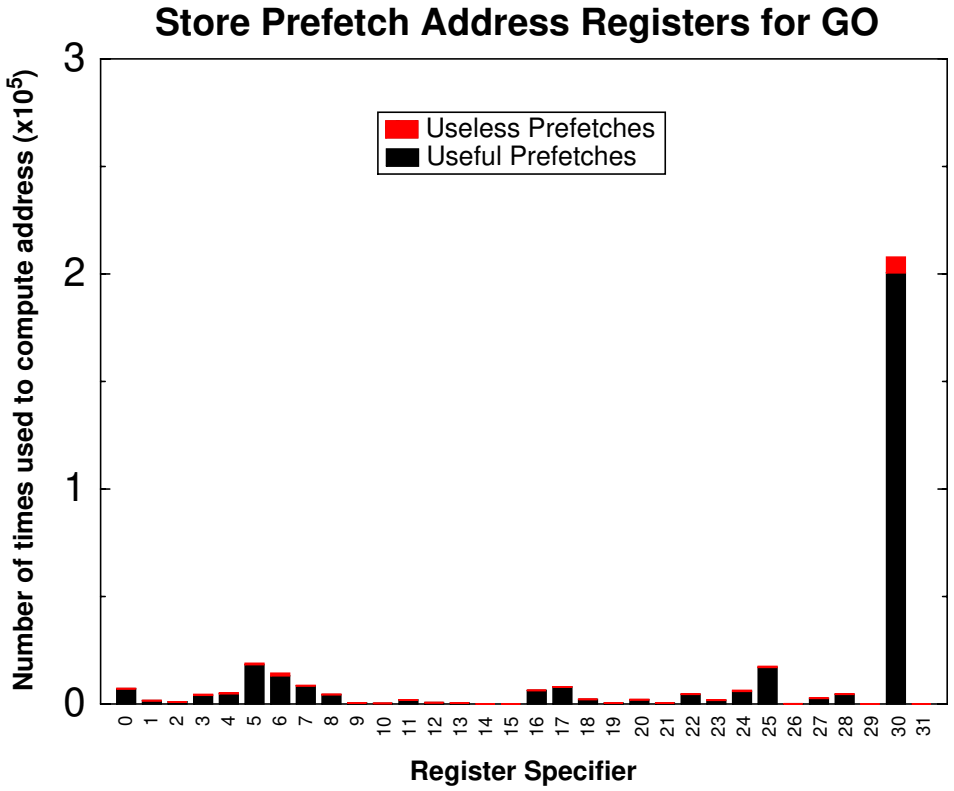


Figure 5.30



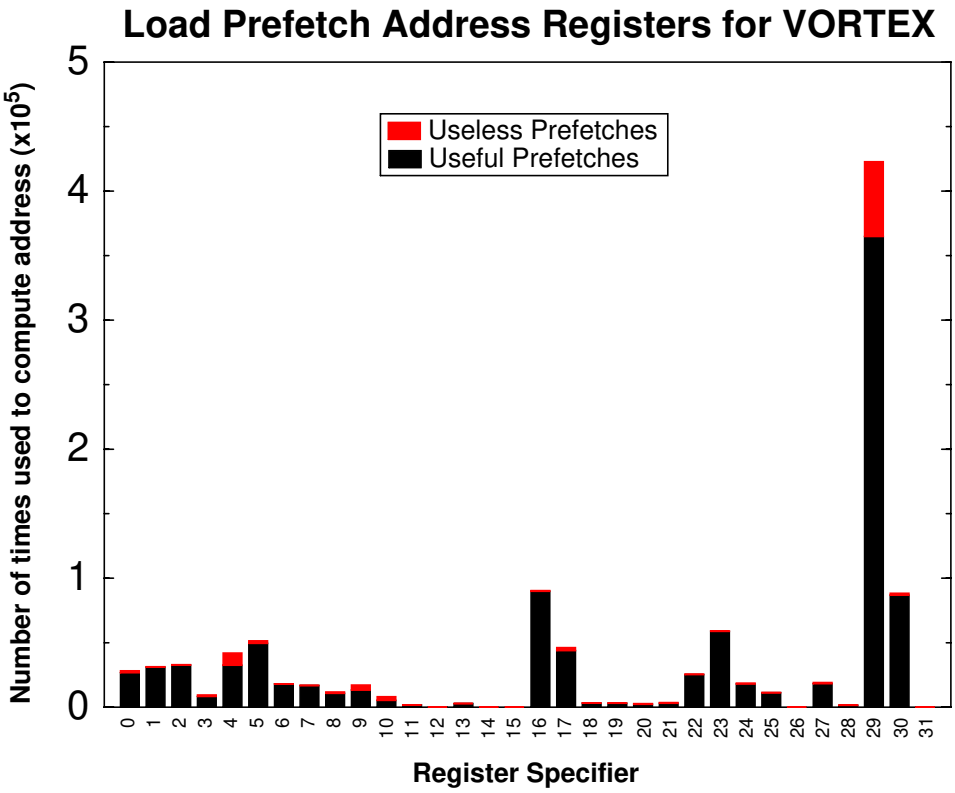


Figure 5.31

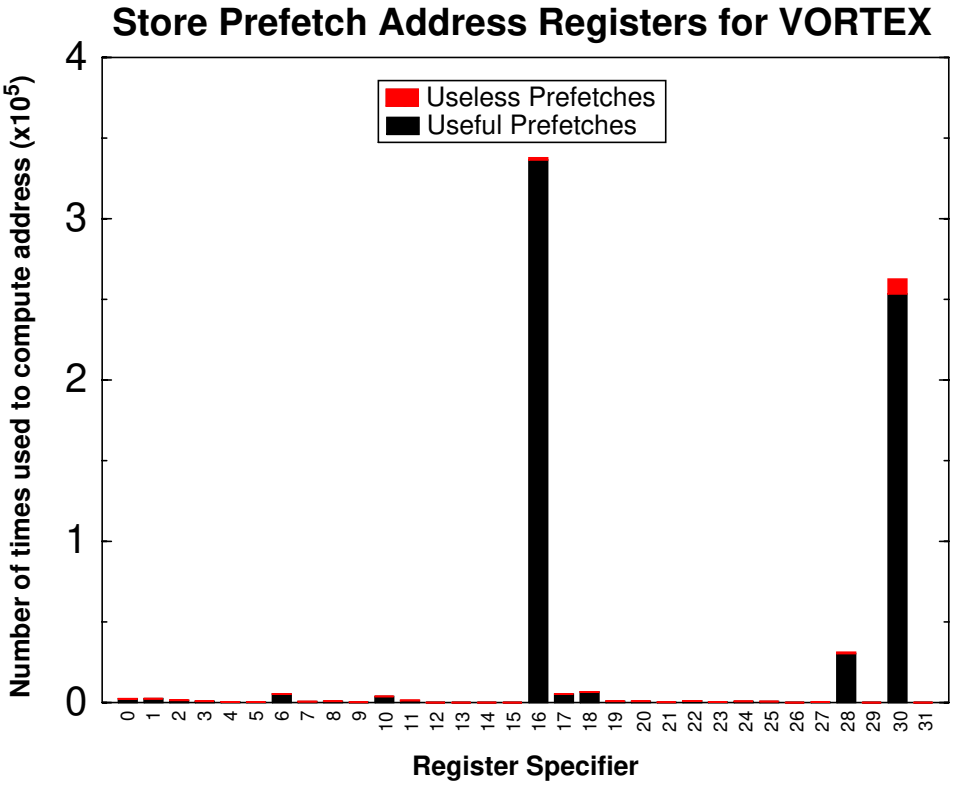


Figure 5.32

Figure 5.33

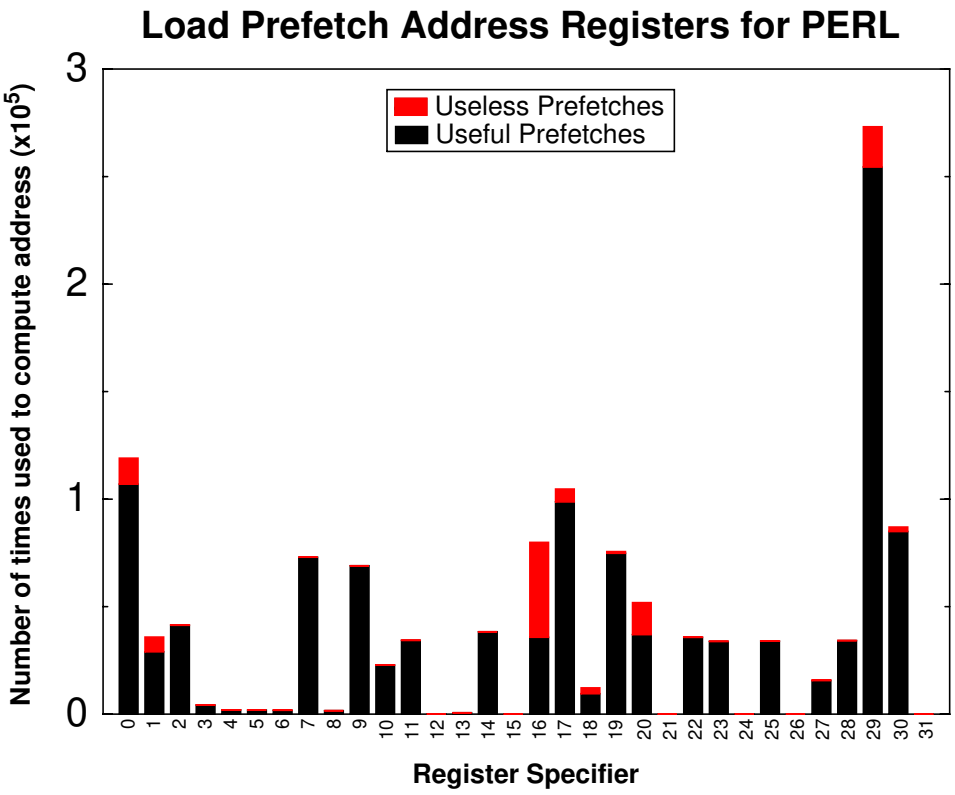
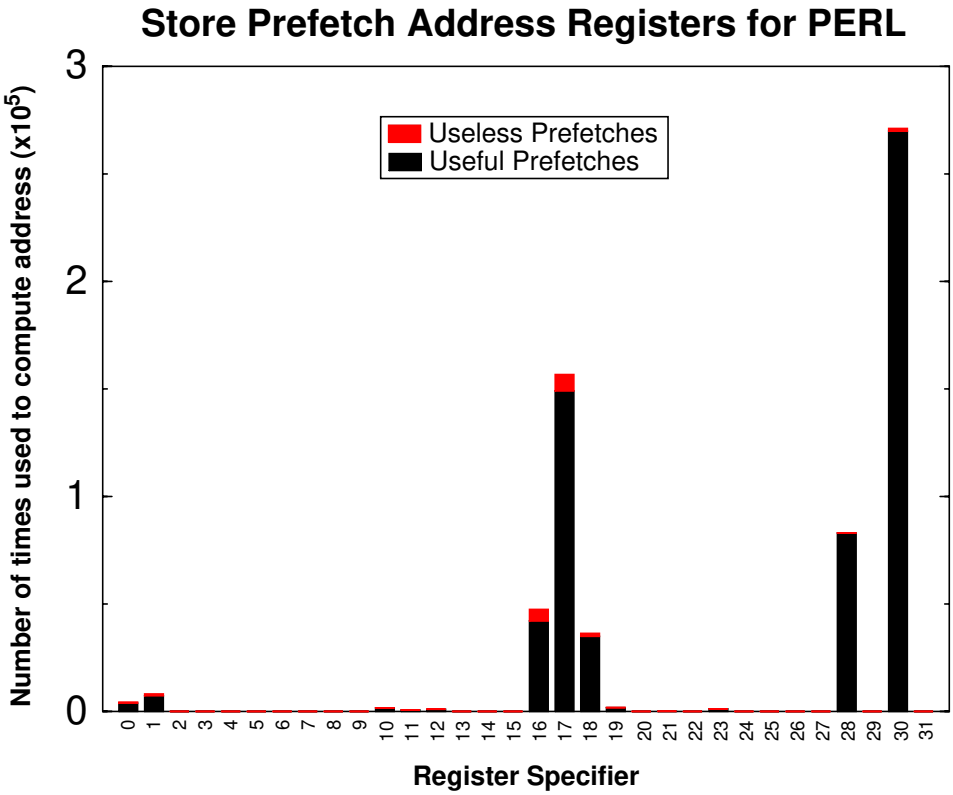


Figure 5.34



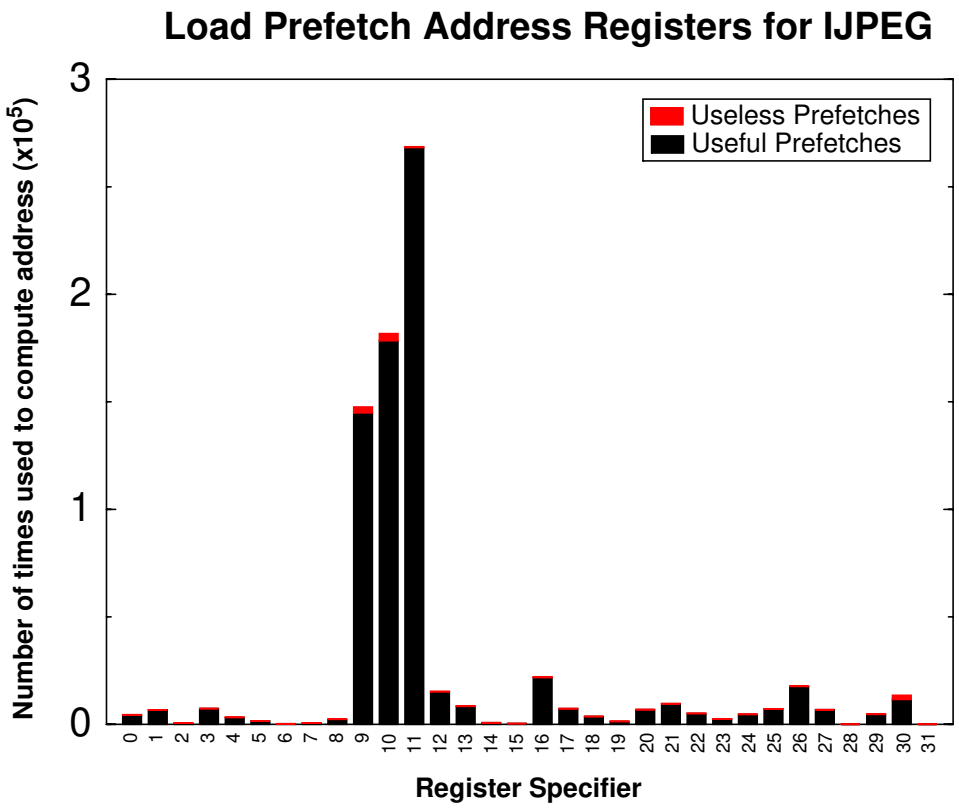


Figure 5.35

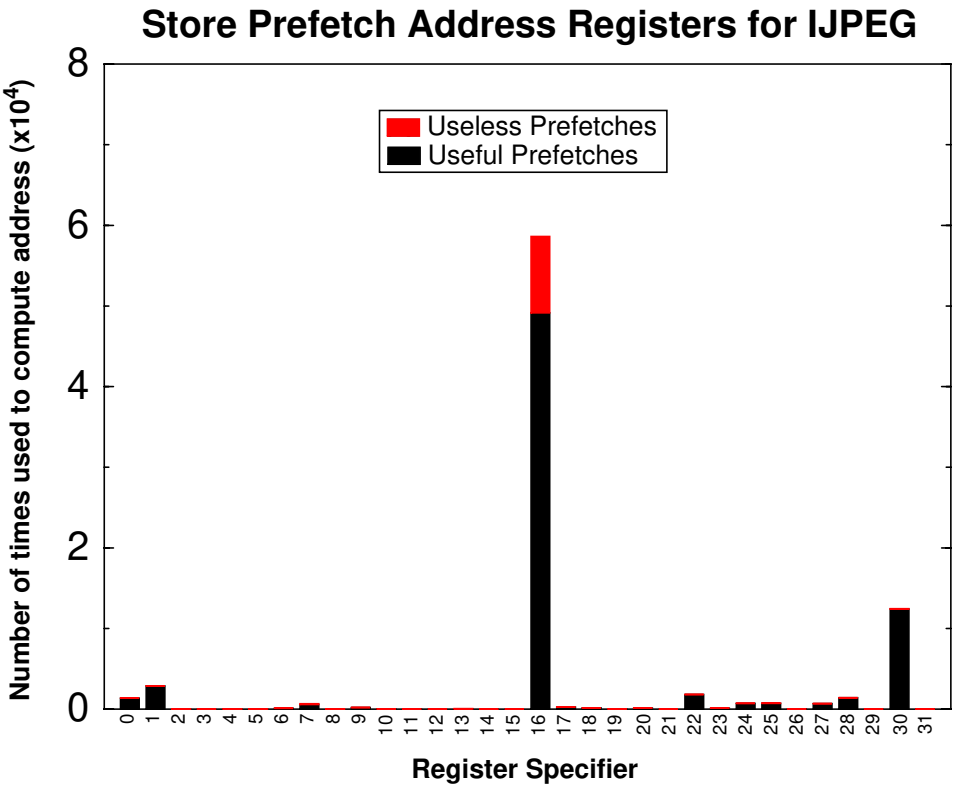


Figure 5.36

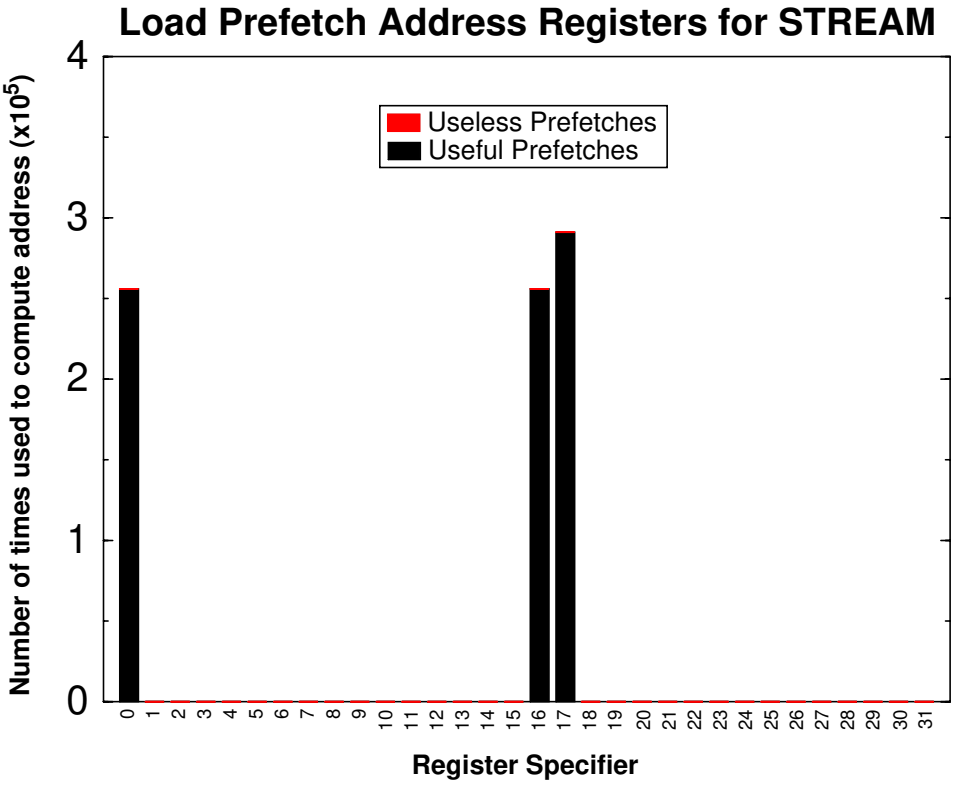


Figure 5.37

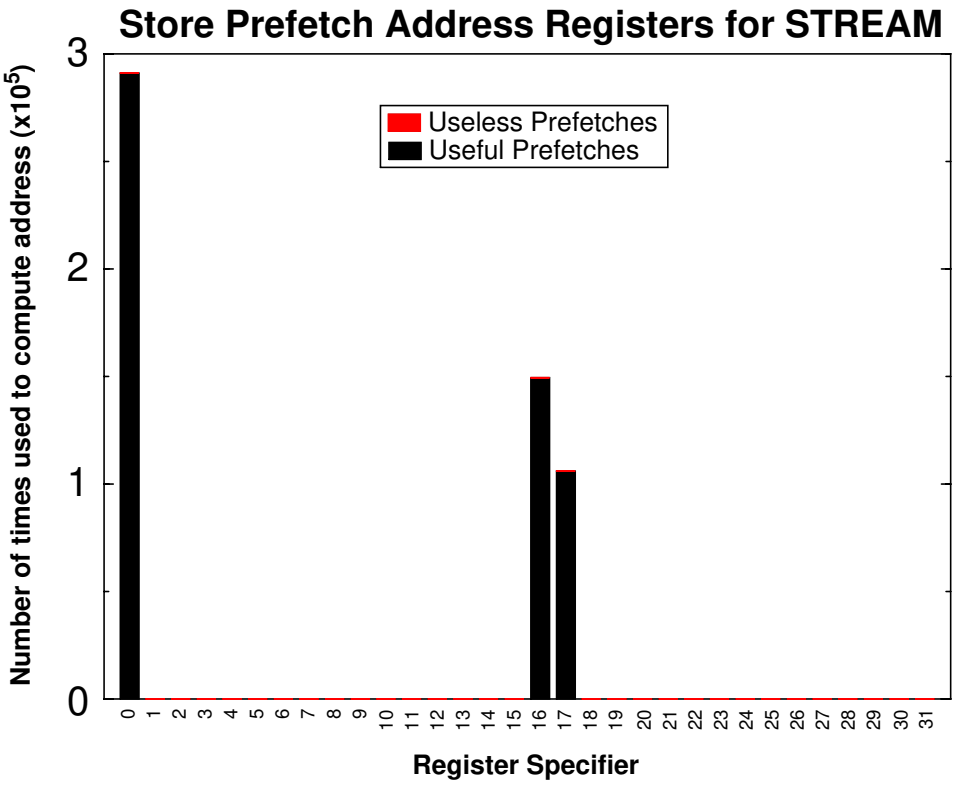


Figure 5.38

5.3 Summary and Conclusions

We have shown that much of the work that is performed during runahead is confined to a subset of the instruction set and register file. Load and store prefetch addresses are almost entirely computed with a combination of load, load_address and add_sub instructions. The resolution of most indirect branches can also be performed with the above combination of instructions. Conditional branches require only the addition of compare instructions to the above mix. The registers that are needed to compute results during runahead are also largely segregated to a subset of the register file. These results imply that it would be possible to perform runahead with a very simple coprocessor that would only implement a small portion of the instruction set. Instructions that are not implemented can simply be treated as NOPs, and their destination registers marked as INV.

We have also shown that for most of the benchmarks prefetches are generated for an approximately equal mix of array/pointer, global, and stack references. The IJPEG and STREAM benchmarks are an exception to this rule, as most of their prefetches are generated for array references. The load and store instructions that generate the most prefetches per instruction (i.e. the most efficient instructions) tend to generate prefetches for array or pointer accesses. The global and stack prefetches are generated by less efficient instructions, however this low efficiency per instruction is offset by the large number of instructions in this category.

Chapter 6

Probing the Limits of the Baseline Technique

The baseline runahead processor model that was used in the preceding chapters suffers from a variety of effects that can reduce performance. We address these effects in this chapter in an attempt to determine the maximum data stream performance gains that can be had when runahead is employed.

6.1 Instruction cache misses during runahead

Instruction cache misses can affect the performance of a runahead processor in several ways. The most obvious effect is that they can cause the processor to suffer from a dearth of instructions to pre-process. If the processor can find other instruction cache lines to pre-process, then it can continue the runahead episode. Unfortunately, dependences upon the instructions that are not pre-processed can cause the processor to generate useless prefetches. These dependences can also cause the processor to incorrectly resolve conditional branches, sending the processor down an incorrect path. Wrong path effects also occur when the processor cannot pre-process taken branches in the skipped instruction cache lines.

Eliminating the effects of instruction cache misses is simple enough for the purposes of this study: simply simulate a processor that has a perfect L1 instruction cache.

6.2 Increasing the length of the average runahead episode

The length of the average runahead episode is approximately equal to the average access time to off-chip memory. Increasing this access time may allow the processor to increase the number of prefetches that are generated per runahead episode, and may even lead to an improvement in performance. The simplest way to increase the length of the average runahead episode is to simply eliminate the L2 data cache from the memory hierarchy.

Unfortunately, eliminating the L2 data cache from the baseline processor can cause other problems. The high bandwidth and store coalescing effect of the write-back L2 data cache kept the store-through policy of the L1 data cache from inhibiting performance. Eliminating the L2 data cache can make the store-queue a bottleneck, causing the processor to stall. One solution is to modify the store queue such that it can coalesce requests, allowing store-through requests that map to the same line to be collapsed into a single request. We also consider increasing the bandwidth of the main memory interface in order to alleviate bandwidth effects.

6.3 Wrong path effects during runahead

Unresolvable conditional branches during runahead episodes will send the processor down the wrong path if they are mispredicted. This can result in the generation of useless prefetches, as well as in missed opportunities for the generation of useful prefetches.

These effects can be eliminated in simulation if an oracle conditional branch predictor is used during runahead episodes. This can be done by saving a trace of conditional branch outcomes during a training run, and then using this trace as an oracle predictor during subsequent simulation runs. Unresolvable indirect branches also limit performance gains by

forcing the processor to stall during runahead. This problem can be solved in a similar manner by using a trace of indirect branch targets during runahead.

6.4 Runahead Models

We created several new runahead processor models based upon the modifications described in the preceding sections.

Runahead Processor (RA)

This is the same as the runahead processor used in Chapter 4, with the addition of a coalescing store queue and a perfect L1 instruction cache. There is also a non-runahead version of this processor (NORA).

Runahead + No L2 Data Cache (NOL2)

This is the same as the above RA processor, only with the deletion of the L2 data cache. This reduces the minimum access time to off chip memory to 102 cycles from 132 cycles for the RA processor model (assuming an L2 data cache miss). See Figures 3.5 and 3.6 for more details of the timing of the memory hierarchy. There is also a non-runahead version of this processor (NORA_NOL2).

Runahead + No L2 Data Cache + High Bandwidth Main Memory (HIGHBW)

This is the same as the NOL2 processor, only with the main memory interface modified such that it can accept a new fetch, prefetch, or store-through request every processor cycle. This corresponds to a main memory with a peak bandwidth of 32 GB/s as compared to the 1.6 GB/s assumed for all other simulations. The latency of main-memory accesses is unchanged. This extremely high figure is unattainable in a practical implementation at the present time, but we wanted to see what would happen if this was not the case. There is also a non-runahead version of this processor (NORA_HIGHBW).

Runahead + No L2 Data Cache + High Bandwidth Main Memory + Perfect Control (PERFECT)

This is the same as the HIGHBW processor, with the addition of perfect conditional and indirect branch resolution during runahead episodes. Note that the processor only uses the branch traces during runahead episodes: it has to use the two-bit counters and RBR to predict branches during normal operation. As this processor model only employs oracle prediction during runahead its non-runahead equivalent is the same as that for HIGHBW (NORA_HIGHBW).

6.5 Simulation Results

6.5.1 CPI

The CPI results for the processor models described in the previous section are provided in Figures 6.1 through 6.5. As before, the CPI for each simulation is represented as a stacked bar, which is broken down into its component contributions by instruction class. The overall CPI for each configuration is provided as a number on top of each bar, while the percentage reduction in load and store CPI (not counting cycles in which loads and stores are retired) is provided to the right of each bar. These percentage reductions in CPI are relative to that of the equivalent non-runahead processor, which is indicated with a diagonal line.

We had hoped initially to show that deleting the L2 data cache would improve performance by providing the processor with more opportunities to pre-process instructions. This is true for the STREAM benchmark for one configuration shown in Figure 6.1. It turns out that a runahead processor without an L2 data cache and with a very high bandwidth main memory (HIGH_BW) does indeed have a CPI (3.28) significantly lower than that of the runahead processor with an L2 data cache (RA) (4.35). Unfortunately this improved perfor-

mance is largely due to the higher bandwidth main memory interface of the HIGH_BW processor. When a standard bandwidth main memory interface is employed, the CPI of a runahead processor without an L2 data cache (NOL2) is high enough (5.19) to make its performance somewhat worse than that of the runahead processor with an L2 data cache (RA). As virtually all off-chip memory requests for this benchmark have to go out to main memory the L2 data cache simply gets in the way. The RA processor forces off-chip accesses to check the L2 data cache for a hit before starting main memory accesses. This means that for the STREAM benchmark having an L2 data cache increases the latency of off-chip memory accesses by 30 cycles. This increases the average runahead episode length for the RA processor relative to that of the NOL2 processor, resulting in more opportunities to prefetch and a correspondingly lower CPI. Paradoxically, the latency increasing effect of the L2 data cache for STREAM works *against* the non-runahead processor (NORA), causing it to have a higher CPI than its counterpart that does not (NORA_NOL2). Note that the runahead processor with perfect branch and jump prediction during runahead (PERFECT) achieves exactly the same performance as its counterpart that employs conventional conditional branch prediction and indirect branch resolution during runahead (HIGHBW). This is a consequence of the highly predictable conditional branches and lack of indirect branch instructions in the STREAM benchmark.

The results for the VORTEX benchmark, shown in Figure 6.2, are more interesting than those for STREAM. The percentage reduction in overall CPI is significantly greater for the runahead configurations that do not have L2 data caches. The RA processor has an overall CPI of 1.90, 21% lower than its non-runahead counterpart (NORA). A runahead processor without an L2 data cache (NOL2) has a CPI of 4.27, 38% less than that of its non-runahead

counterpart (NORA_NOL2). Unfortunately the NOL2 CPI is significantly greater than that of either the RA or NORA processors. Increasing the bandwidth of the main memory leads to a significant reduction in overall CPI, with the HIGHBW runahead processor coming in with a CPI of 3.24, but this is only a 39% reduction over that of the equivalent non-runahead processor (NORA_HIGHBW). Adding perfect conditional branch prediction and indirect branch resolution during runahead improves performance somewhat (PERFECT) with an overall CPI of 2.98, 44% less than its non-runahead counterpart (NORA_HIGHBW). The CPI results for the GO and PERL benchmarks, shown in Figures 6.3 and 6.4 respectively, are similar to those reported for VORTEX. Note that we were unable to simulate the HIGHBW and PERFECT processor models for PERL due to the extremely large amount of memory required to record runahead episode information for this benchmark.

The CPI results for the IJPEG benchmark are shown in Figure 6.5. As with the VORTEX benchmark, the percentage reductions in CPI for the runahead processors without L2 data cache are significantly greater than those that have L2 data caches. The RA processors has an overall CPI of 1.27, 12% less than its non-runahead counterpart (NORA). The equivalent processor without an L2 data cache (NOL2) has a CPI of 1.93, 35% less than its non-runahead counterpart (NORA_NOL2). Interestingly, the NOL2 CPI is only 34% greater than that of the NORA processor. Increasing the main memory bandwidth led to modest improvements in CPI, with the non-runahead NORA_HIGHBW processor obtaining a CPI of 2.65, only 10% less than its normal bandwidth counterpart NORA_NOL2. The high bandwidth runahead processor HIGHBW achieved a CPI of 1.66, 37% less than NORA_HIGHBW. This is only slightly better percentage-wise than the 35% reduction that the NOL2 processor obtained. Note that the runahead processor with perfect conditional

branch prediction and jump resolution during runahead (PERFECT) obtained only slightly better performance than its HIGHBW counterpart. This is a consequence of the high probability that the processor will stay on the proper path during runahead episodes, as can be seen in Figure 6.14. Finally, note that deleting the L2 data cache made runahead significantly more effective for IJPEG, even though the data cache miss rate for IJPEG is relatively low.

Figure 6.1

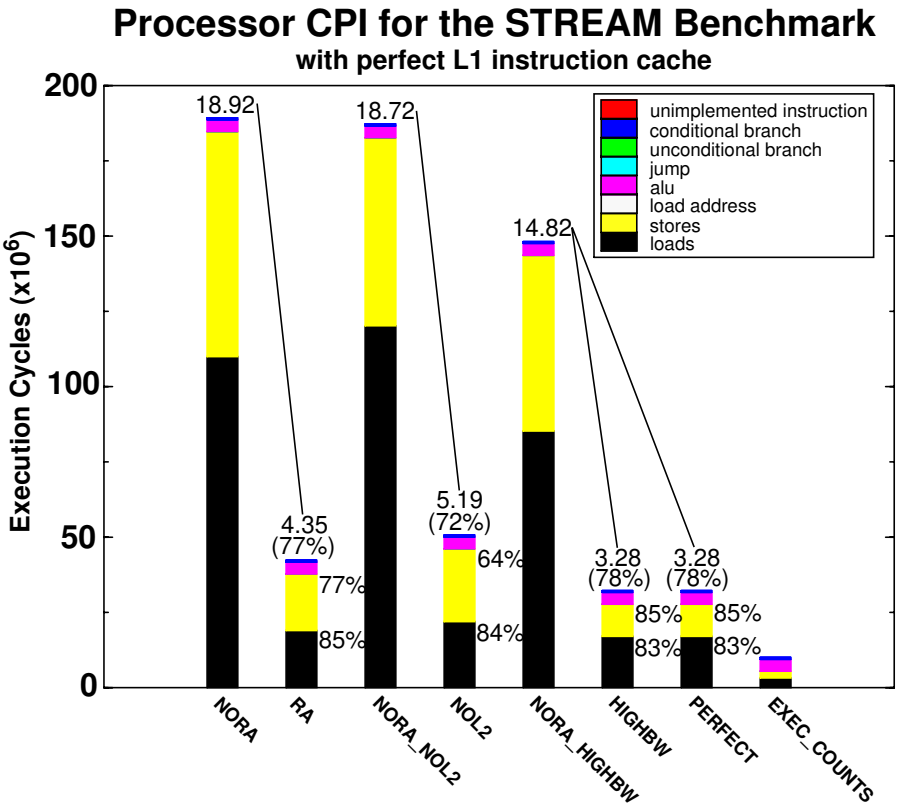
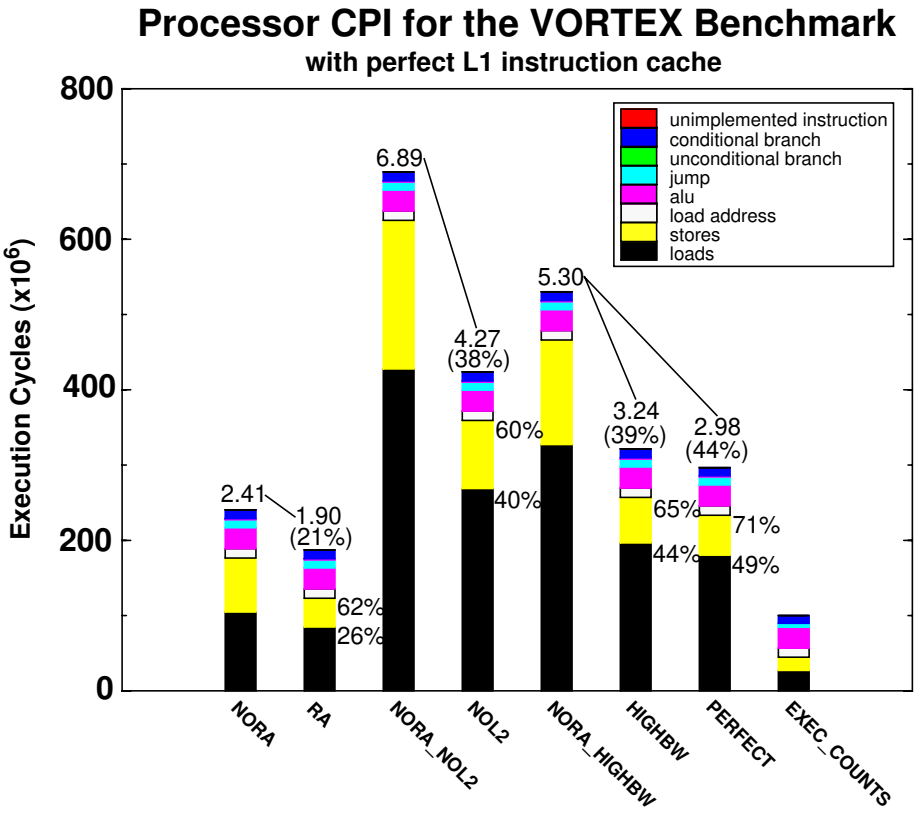


Figure 6.2



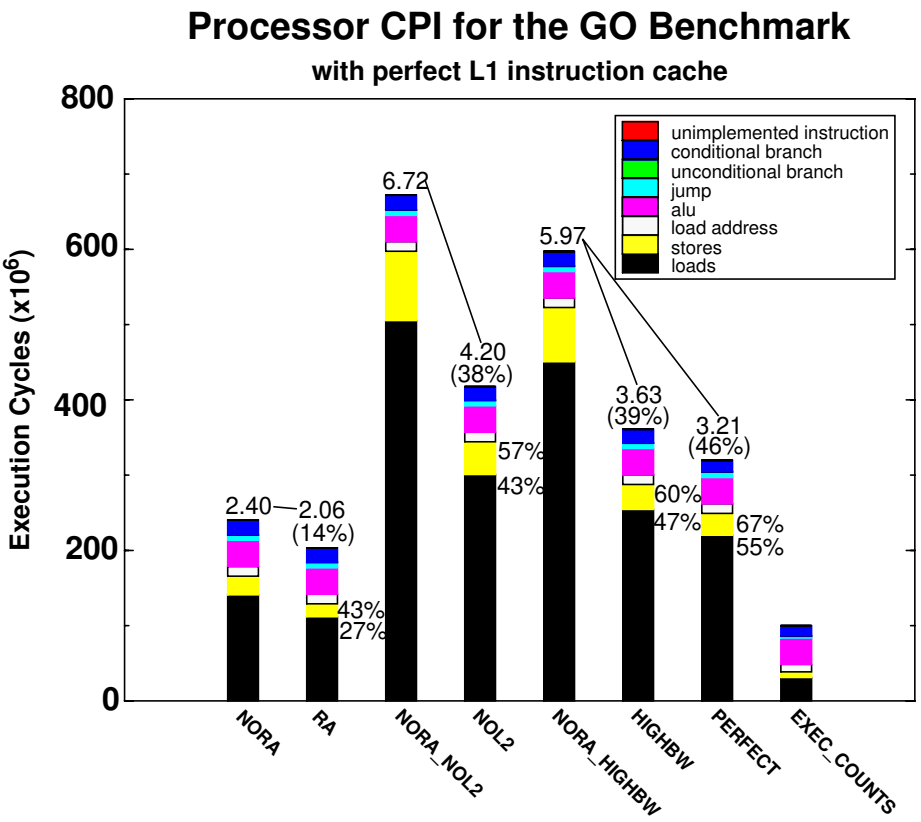


Figure 6.3

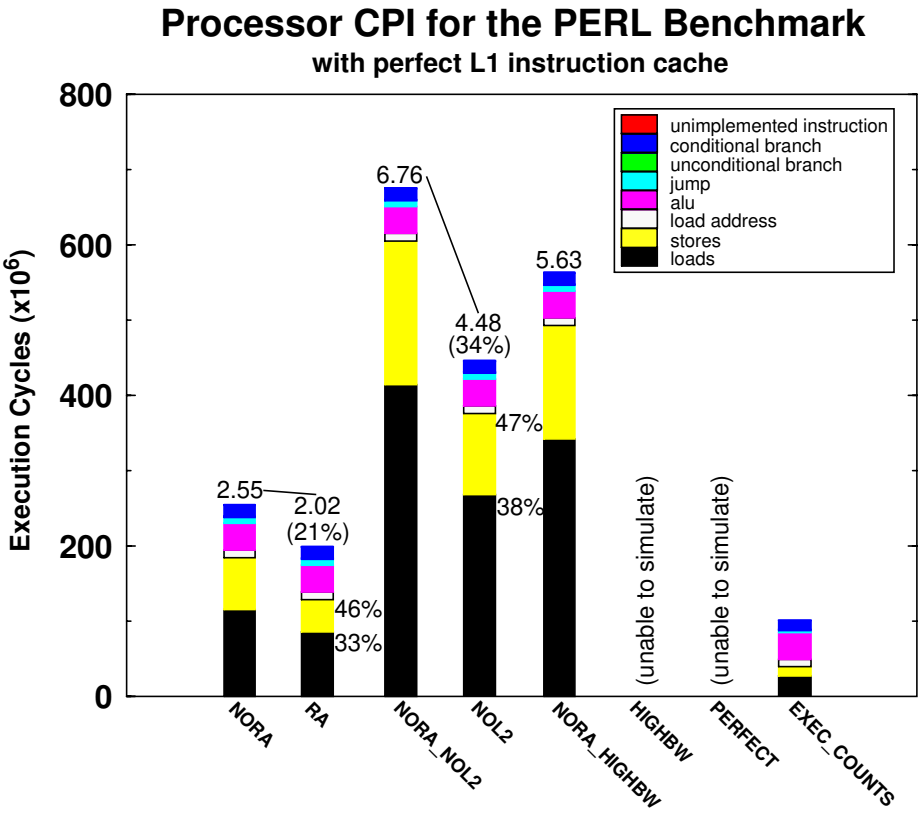
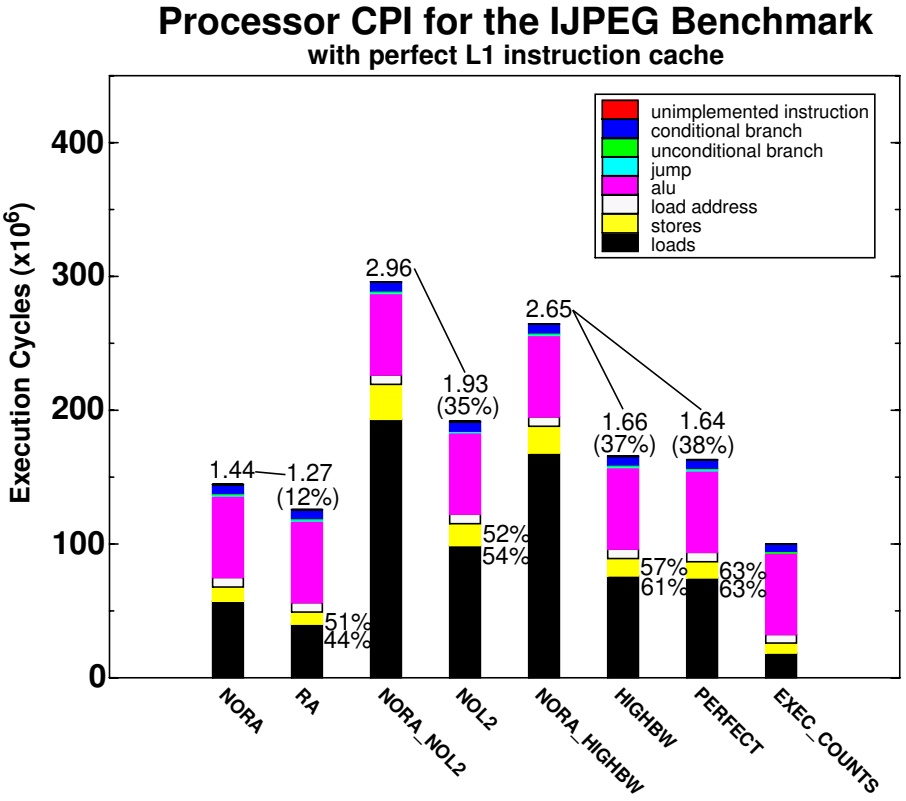


Figure 6.4

Figure 6.5



6.5.2 Prefetching utility over the course of runahead episodes

The ability of runahead to generate useful prefetches over the course of the average runahead episode is of particular interest. This information is provided in Figures 6.6 through 6.10. We have included the prefetch utility for the baseline runahead processor model used for the studies described in Chapter 4. This model (BASELINE_RA) differs from the RA model used in this chapter in that it has a real, as opposed to perfect, L1 instruction cache, as well as a non-coalescing store queue.

The results for the GO benchmark are shown in Figure 6.6. Note that the cumulative number of useful prefetches for the BASELINE_RA processor is virtually identical to that of the RA processor for about the first 10-15 instructions into the average runahead episode. This is a result of the relatively few instruction cache misses that occur early in the average

BASELINE_RA runahead episode. This makes it more likely that the BASELINE_RA processor will stay on the right path and have instructions to pre-process. As expected, the RA processor is more effective than the BASELINE_RA processor, producing slightly more useful prefetches and slightly fewer useless prefetches. The processor configurations that do not include L2 data cache (NOL2, HIGHBW, and PERFECT) are more interesting. These processors generate about 50% more useful prefetches than the RA and BASELINE_RA processors. This is due to the increased length of the average runahead episode, providing more opportunities to prefetch. We had hoped that the useful prefetch curves for the processors without L2 data cache would have the same slope as the early portion of the BASELINE_RA and RA curves, but without a drop off in the number of prefetches generated due to L2 data cache hits. Unfortunately this was not the case: the no-L2 processors have only produced about half as many useful prefetches as the with-L2 processors 25 cycles into the average runahead episode. This is a consequence of increased overlap between runahead episodes. Overlap comes about when runahead episodes are initiated on load or store misses whose instructions have already produced a runahead prefetch in a preceding runahead episode. These episodes spend a great deal of time pre-processing instructions that were already pre-processed in preceding episodes, and may only be able to generate useful prefetches towards the end of the episode where there is little or no overlap. This explains the lower slope of the useful prefetch curves for the no-L2 processors. The PERFECT processor curves are particularly interesting. The total number of useful prefetches is nearly the same as those of the NOL2 and HIGHBW processors, while the number of useless prefetches for PERFECT is less than half of that of NOL2 and HIGHBW. This is a result of the perfect conditional branch prediction and indirect branch target resolu-

tion of the PERFECT processor during runahead episodes. Interestingly, the useful prefetch curve for the PERFECT processor appears to be nearly linear. The perfect branch prediction increases the amount of overlap between episodes, resulting in fewer prefetches being generated during the early portion of the average episode. This is compensated for later in the average episode, where there is less overlap, and the other processor configurations are less likely to be on the proper path. The plots for the VORTEX, PERL, and IJPEG benchmarks, shown in Figures 6.7 through 6.9, are very similar to that for GO.

The plot for the STREAM benchmark, shown in Figure 6.10, is particularly interesting. By design STREAM does not benefit to any meaningful extent from an L2 data cache. This can be seen in the plot for the STREAM benchmark, in which the useful-prefetch plots for all of the runahead processors are very similar.

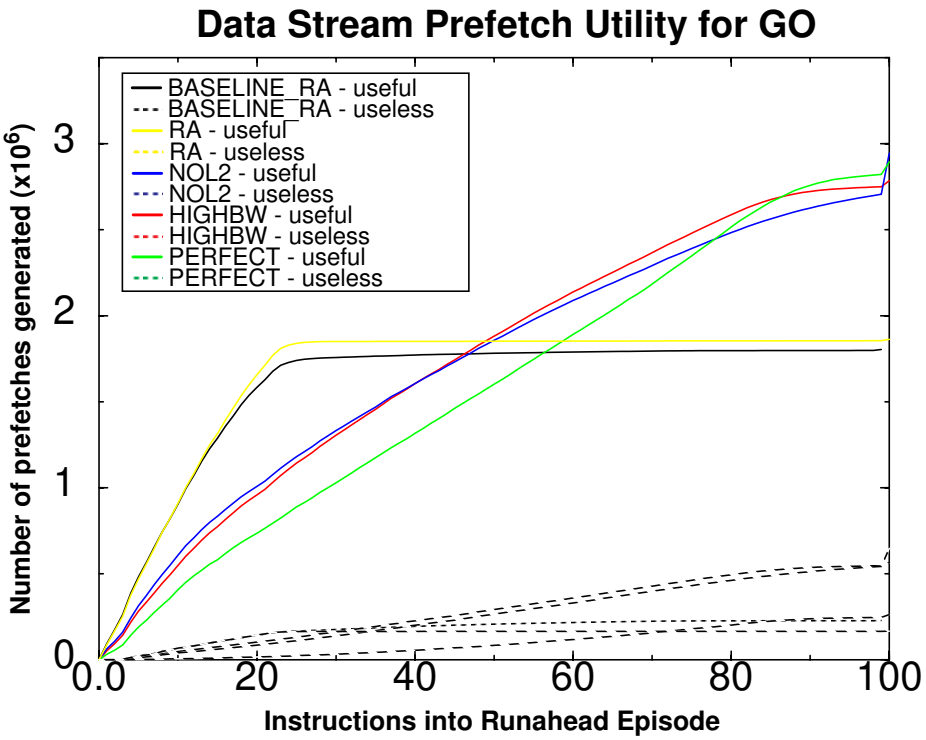


Figure 6.6

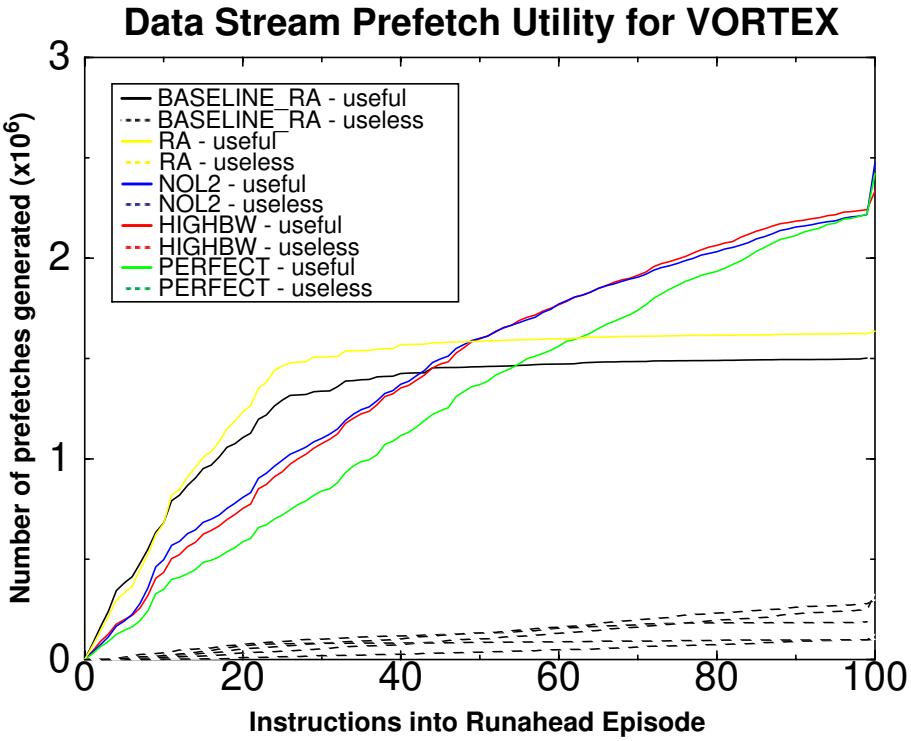


Figure 6.7

Figure 6.8

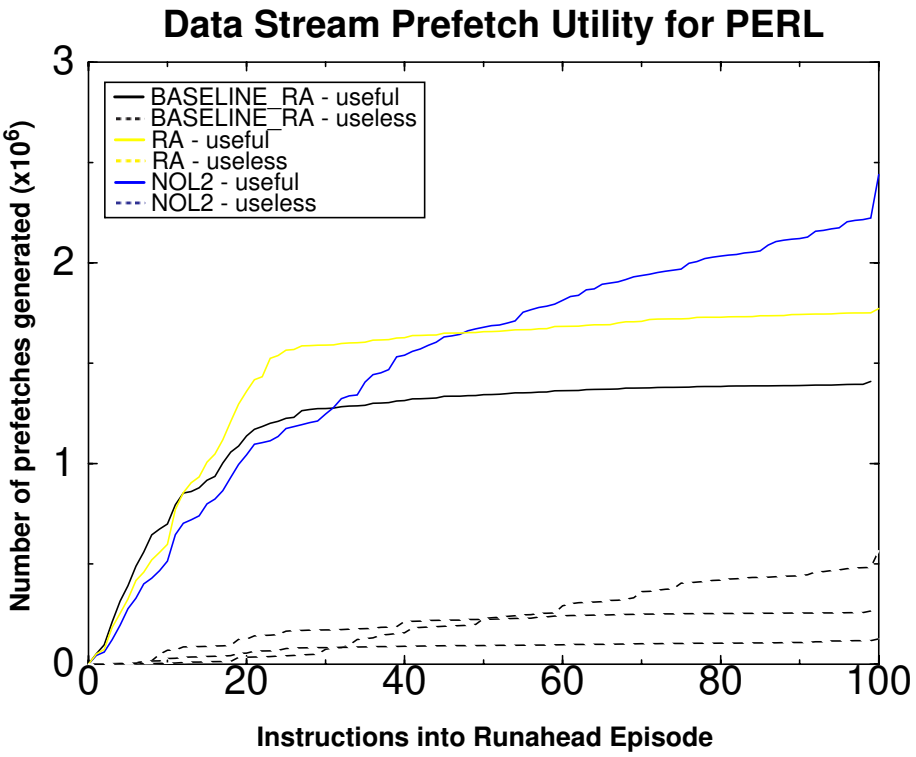


Figure 6.9

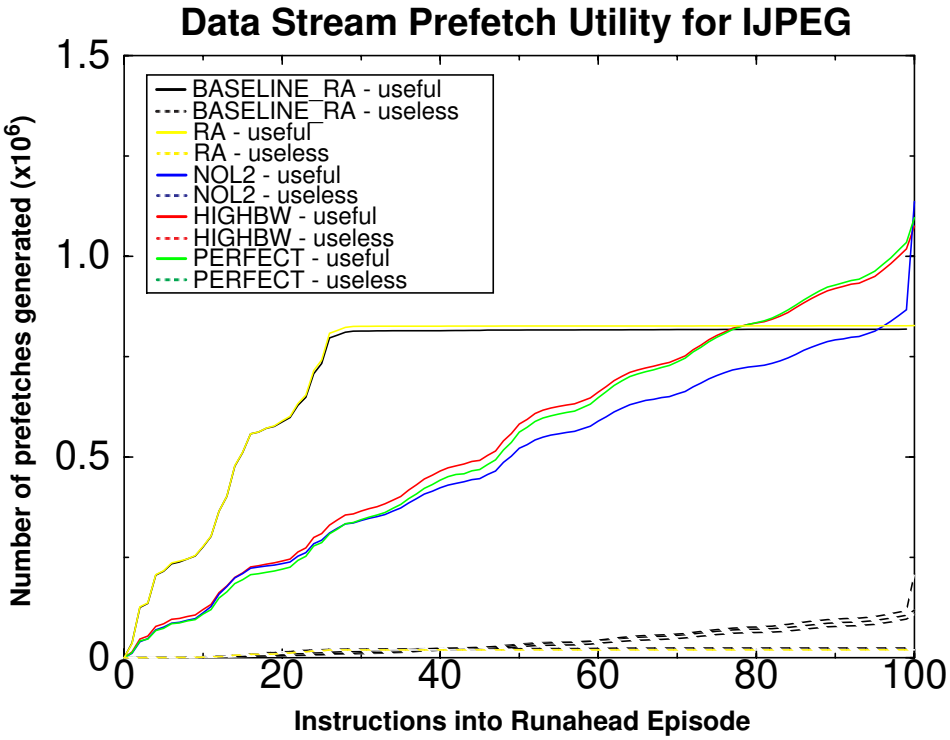
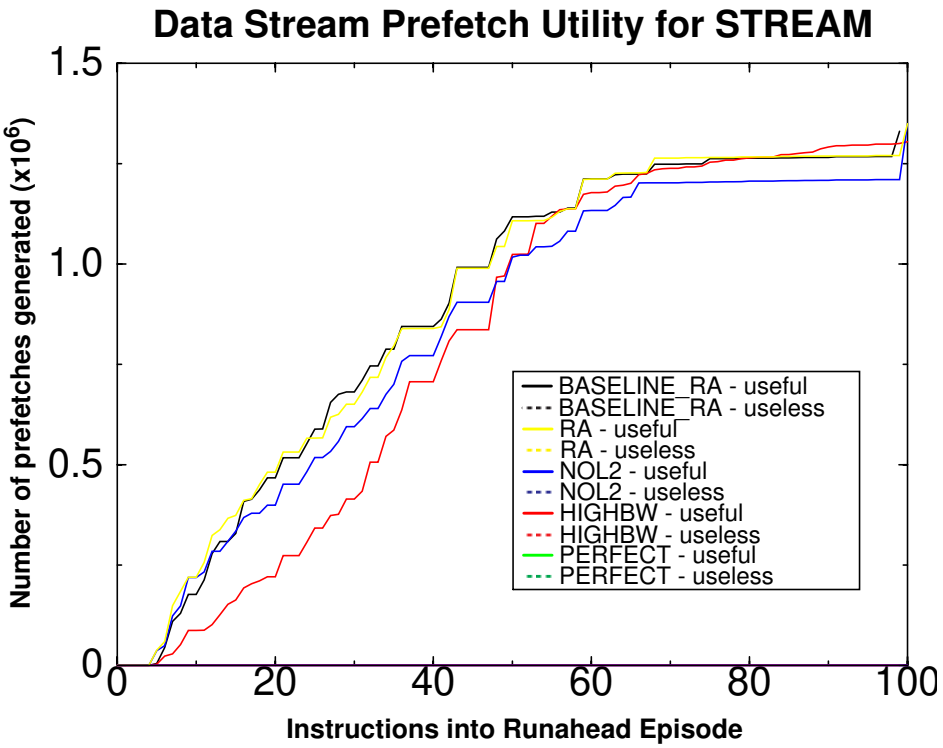


Figure 6.10



6.5.3 Probability of remaining on the correct path during runahead

We present here plots that illustrate the probability that the processor is on the right path at a given distance into the average runahead episode. An instruction is on the proper path if all of the instructions in the runahead episode up to and including the instruction in question are executed during normal operation at the conclusion of the runahead episode, starting with the first instruction in the episode. We do not include RA processor model results as they are virtually identical to those for the BASELINE_RA model presented in Chapter 4. These similar results indicate that instruction cache misses during load and store miss initiated runahead episodes are not a significant source of wrong path effects. The plots in Figures 6.11 through 6.14 are for the runahead processor models without L2 data cache and with perfect L1 instruction cache (NOL2 and HIGHBW). For most of the plots the

HIGHBW and NOL2 processors exhibit nearly identical behavior. We will refer to both models together unless stated otherwise. We do not present STREAM results as the processor always stays on the right path during runahead. Note that the plots can exhibit erratic behavior near the x = 100 position on the x-axis. Prefetching reduces the average memory latency on a miss during normal operation, resulting in a relatively small number of long-latency episodes. The smaller number of episodes contributing towards the average creates the seemingly erratic behavior.

The first plot is for the GO benchmark, and is shown in Figure 6.11. As was reported in Chapter 4, the probability of remaining on the correct path drops off rather precipitously for load-miss initiated runahead. The drop off for store-miss initiated runahead is less severe, however both runahead episode types have only about a 20% chance of being on the correct path after pre-processing 100 instructions. Fortunately, most of the wrong path prefetches generated by GO are useful.

The plot for the VORTEX benchmark, shown in Figure 6.12, exhibits a fairly high probability of remaining on the correct path even after 100 instructions. Store-miss initiated runahead has about an 80% probability of remaining on the correct path, while load-miss has about a 60% probability. The plot for PERL, shown in Figure 6.13, is similar to that for VORTEX, only with the probability of remaining on the correct path about 50% after 100 instructions.

As was noted in Chapter 4, IJPEG has a high probability of remaining on the proper path during runahead. This can be seen in Figure 6.14. Note that the probabilities remain high throughout the life of the typical episode, and that the values are nearly identical for both load- and store-miss initiated runahead.

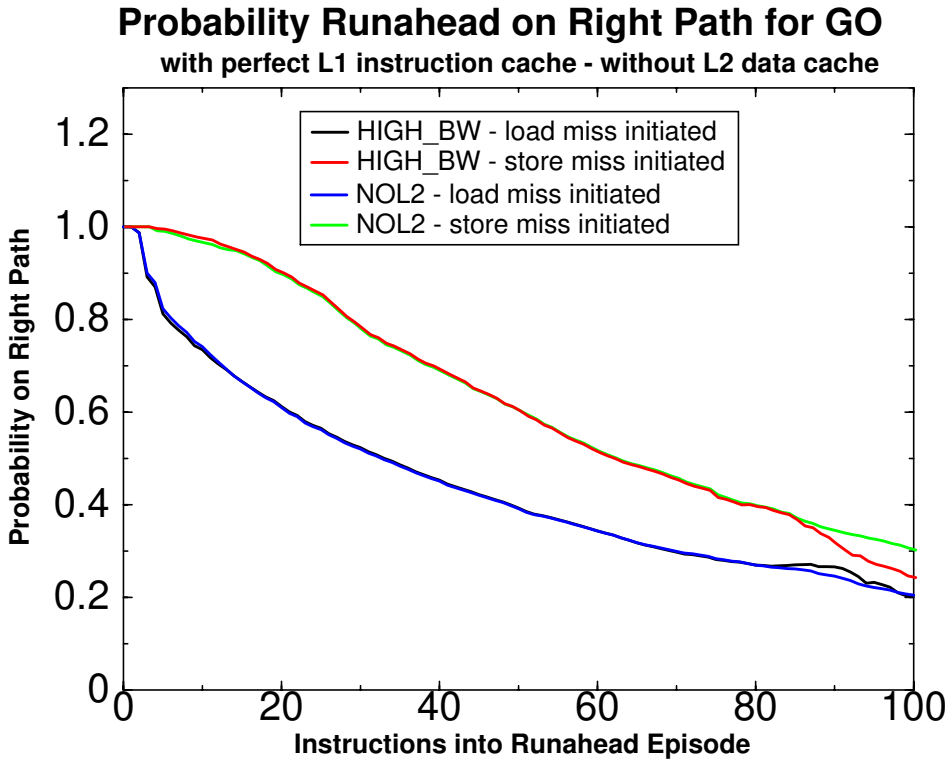


Figure 6.11

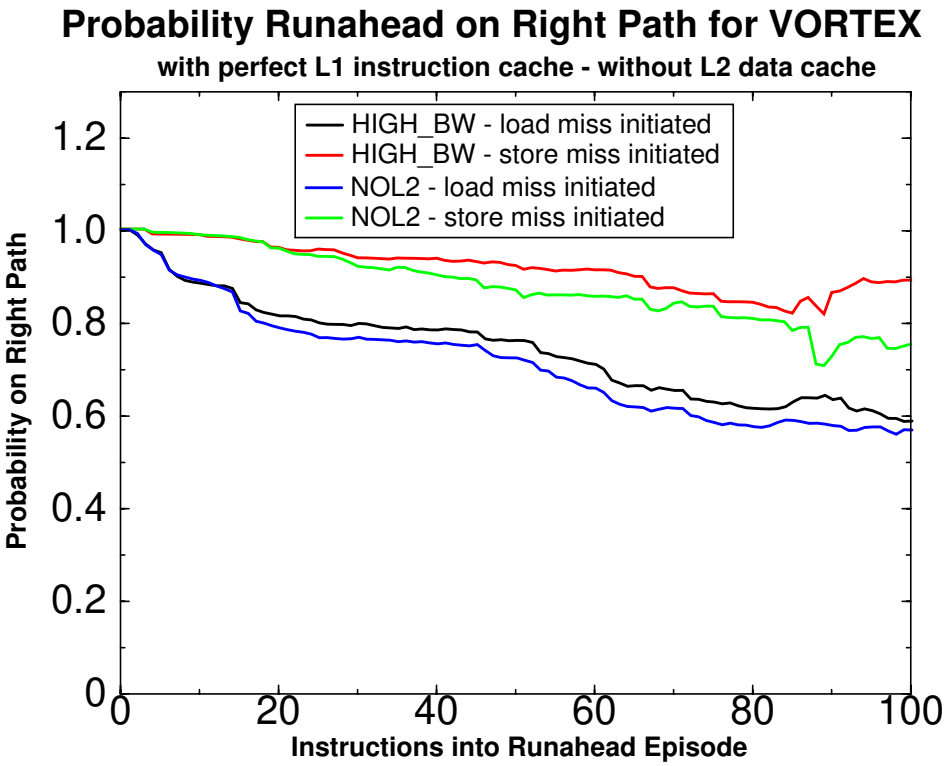


Figure 6.12

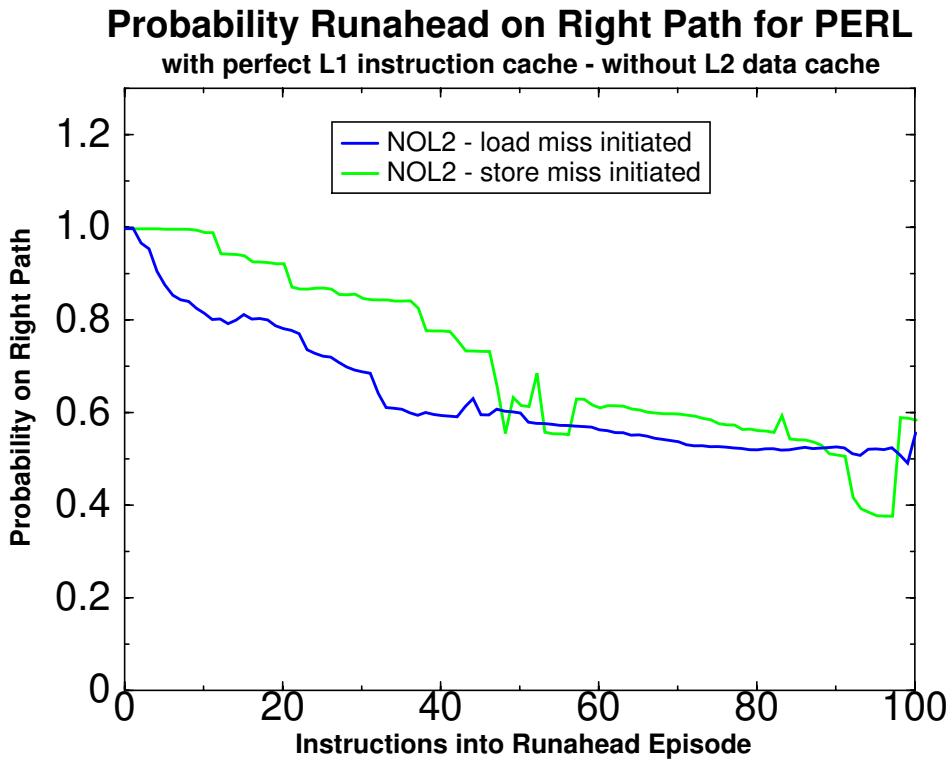


Figure 6.13

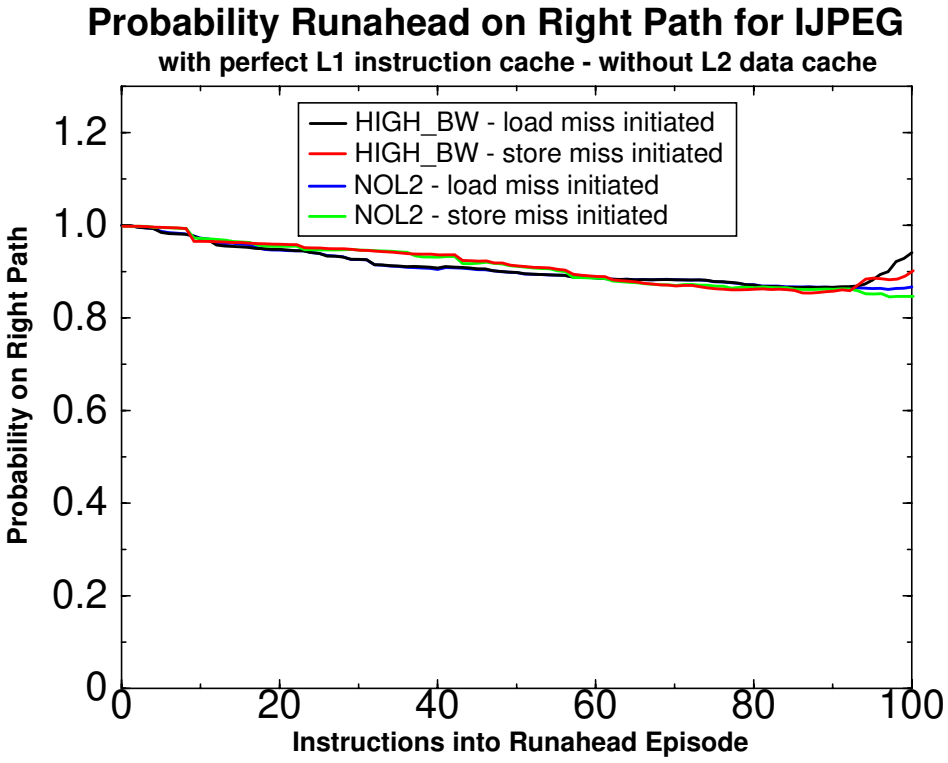
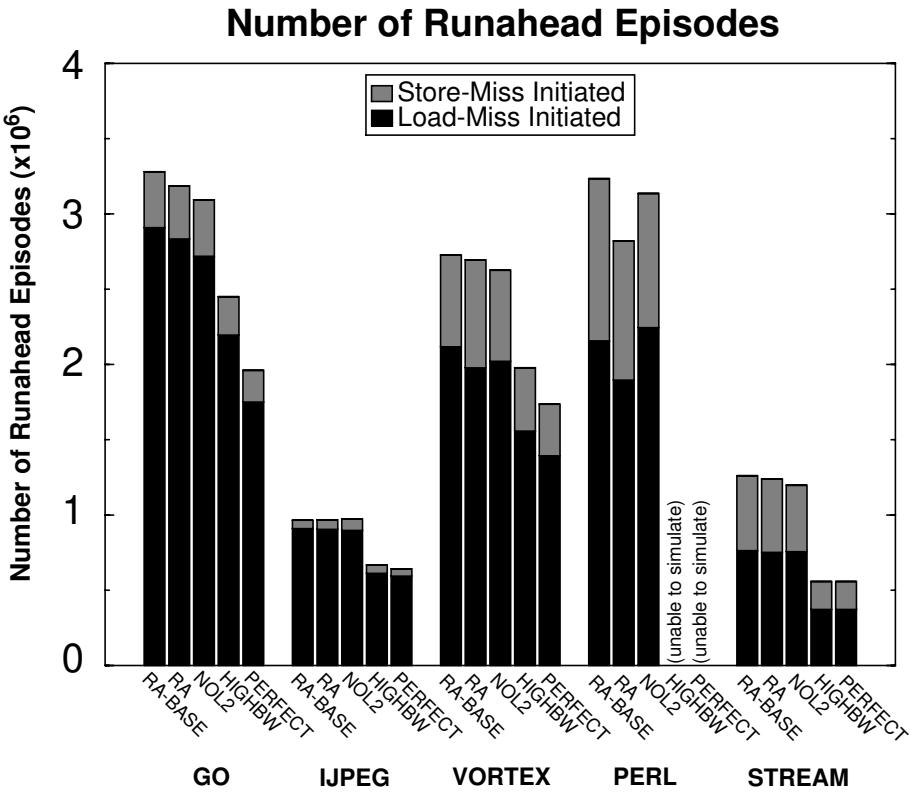


Figure 6.14

6.5.4 Number of runahead episodes

The number of load- and store-miss initiated runahead episodes for each benchmark and runahead processor configuration are provided in Figure 6.15. Generally speaking, the number of runahead episodes decreases when runahead is more effective due to the elimination of instruction cache misses, deletion of the L2 data cache, addition of a higher bandwidth main-memory, or the elimination of wrong path effects. This is true for all of the benchmarks, with the exception of the NOL2 configuration for PERL, which has a slightly larger number of episodes than either the RA-BASE or RA configurations. The larger number of prefetches that are generated when the L2 data cache is deleted can increase the total number of runahead episodes if the prefetches are not serviced by the time that the processor actually executes the prefetch-generating instructions. Also, note that the rather low number of runahead episodes for STREAM is due to the lower number of instructions that were simulated for this benchmark: 10M vs. 100M for the others. STREAM spent by far the largest fraction of its execution time pre-processing instructions.

Figure 6.15



6.5.5 Average number of prefetches generated per runahead episode

Figures 6.16 and 6.17 provide the average number of instruction, load, and store prefetches that are generated during each type of runahead episode for all of the processor configuration and benchmark combinations. Note that the average number of prefetches generally increase as we make runahead more effective by adding a perfect instruction cache, deleting the L2 data cache, increasing main-memory bandwidth, and eliminating wrong path effects. The only exception to this rule is the STREAM benchmark, where the average number of load prefetches falls instead of rising for the NOL2 processor. It turns out that the runahead processors were dropping about 75% of all prefetch requests for STREAM due to a full prefetch queue, as can be seen in Figure 4.76. This, plus the fact that STREAM does not benefit from the L2 data cache, indicate that the main-memory band-

width is the performance bottleneck for STREAM. This was expected due to the nature of the benchmark. The HIGHBW and PERFECT processor models simulate a much higher main-memory bandwidth, 32 GB/s vs. 1.6 GB/s, allowing the processor to handle more prefetches. This shows up as a significant increase in the average number of prefetches generated per runahead episode. These additional prefetches lower the L1 data cache miss rate, resulting in a significant drop in the number of runahead episodes. This offsets the increase in the average number of prefetches generated to some extent.

Note that load-miss initiated episodes generally generate fewer prefetches per-episode than store-miss initiated episodes. Load misses immediately seed the RF with INV values, reducing the ability of the processor to compute prefetch addresses. This is offset by the significantly larger number of load-miss episodes, as can be seen in Figure 6.15. Also note that load-miss initiated episodes tend to generate more load than store prefetches, indicating that load-misses tend to be clustered. Store-miss initiated episodes behave in a similar manner, as we expected.

Figure 6.16

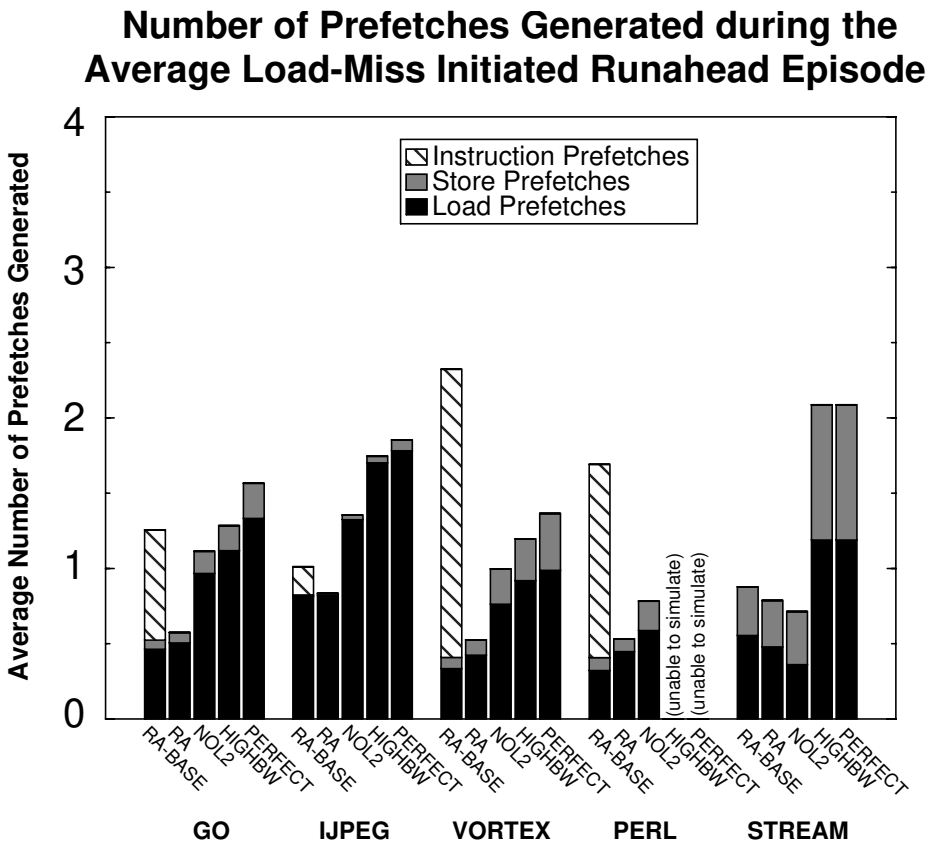
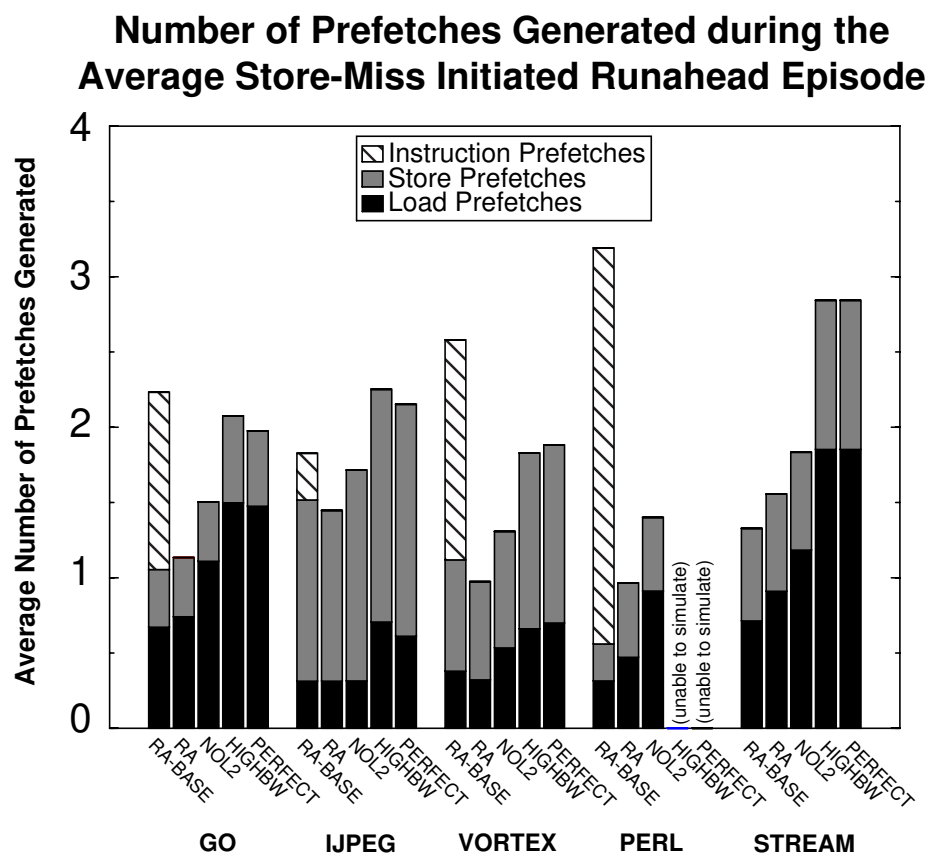


Figure 6.17



6.5.6 Off-chip fetch and prefetch traffic

The total off-chip data stream fetch and prefetch traffic is provided for each benchmark in Figure 6.18. A stacked bar is used to represent the individual contributions of prefetches and demand fetches towards the total traffic. Note that off-chip traffic refers to all prefetch and fetch requests from the L1 data cache to either the L2 data cache or main memory.

In all cases the amount of demand fetch traffic is reduced as runahead is made more effective by adding a perfect L1 instruction cache, deleting the L2 data cache, increasing main-memory bandwidth, or eliminating wrong path effects. Note that the runahead processor models without L2 data cache replace more than half of their demand fetch requests with prefetch requests. Virtually all of the demand fetch traffic is replaced with prefetches in the STREAM simulations. Also, the total data stream traffic generally increases to some extent when runahead is employed. There is a small jump in overall traffic for the NOL2 processor: this is due to an increase in wrong path prefetching during the longer runahead episodes. The PERFECT model eliminates most of this increase.

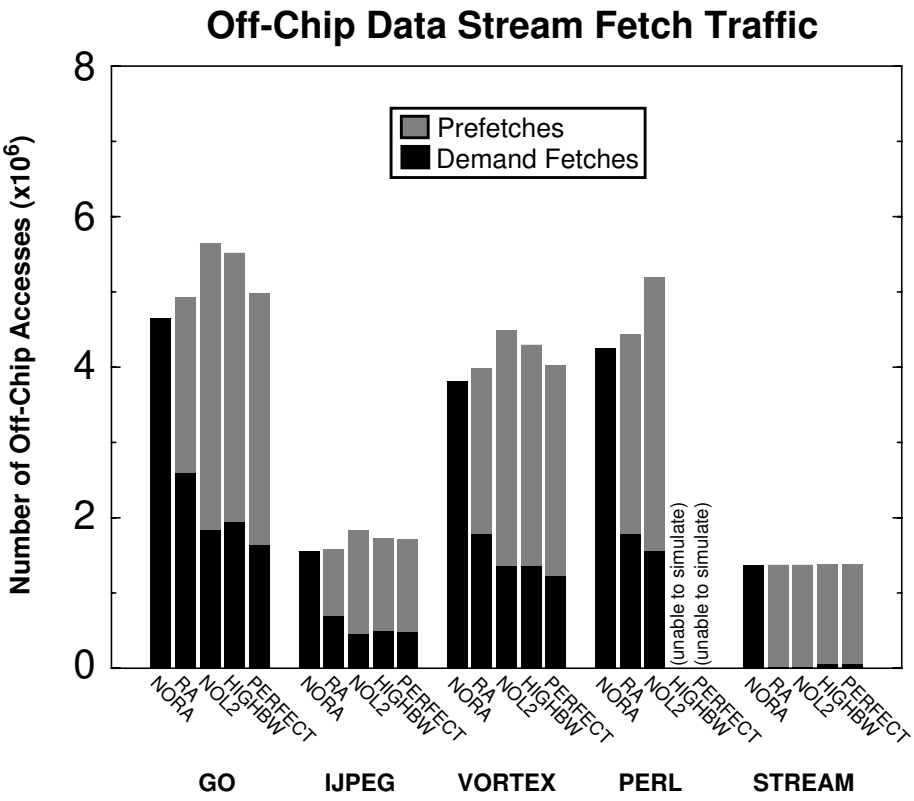


Figure 6.18

6.6 Summary and Conclusions

We had hoped that by extending the length of the average runahead episode by deleting the L2 data cache we would be able to achieve runahead processor performance that would surpass that of a runahead processor that had an L2 data cache. This was not the case, even though the effectiveness of runahead is typically significantly increased by removing the L2 data cache. Failing at this, we hoped that at the very least we would be able to match the performance of non-runahead processors that incorporated L2 data cache. This was not true, even when we eliminated wrong path effects during runahead by using a trace of conditional branch outcomes and indirect branch targets, in addition to dramatically increasing the bandwidth of the main memory interface.

The runahead processor models without L2 data cache are competitive in the sense that they do achieve performance that is reasonably close to their non-runahead counterparts that include L2 data cache, yet at a considerably lower cost to the user. The cost of implementing runahead is certainly much lower than the cost of purchasing large off-chip L2 data cache SRAMS and the requisite interface, even with the potentially expensive cost of integrating the BRF with the architected register file. This last concern is addressed in Chapter 7.

We have also demonstrated that the probability of the processor remaining on the correct path during load- and store-miss initiated runahead episodes is fairly high, even when the average runahead episode length is significantly increased by deleting the L2 data cache. The GO benchmark is a notable exception to this rule, however this is mitigated by the large fraction of wrong path prefetches that are useful rather than useless. We have also shown that instruction cache misses are not a significant source of wrong path effects during load- and store-miss initiated runahead episodes.

Chapter 7

Simplifying the Runahead Processor

The runahead processor models that have been used thus far achieve good performance, but this comes at a cost. There are a number of modifications that may lead to a significant reduction in the implementation cost and complexity of runahead. We address these concerns in this chapter.

7.1 Eliminating the register file save and restore operation

The runahead processor implementations that were used in the previous chapters all assumed that the sequential state of the architected register file could be checkpointed by saving the contents of the RF in the BRF at the initiation of runahead, and restoring the contents at the resumption of normal operation. These operations were assumed to take place in a single cycle, which may be unrealistic.

However, the register file save and restore operation is not needed to guarantee proper program execution during normal operation. All the processor has to do is guarantee that the sequential state of the architected register file and memory hierarchy are not corrupted during runahead. No other guarantees (other than not causing exceptions) have to be made, as runahead episodes are entirely speculative and all results computed during runahead are discarded.

One way to simplify the register file checkpoint operation is to simply do away with the register file save and restore operations, while retaining the BRF. The RF is then used to hold the sequential state of the architected register file, while the BRF is used during runahead episodes. The processor directs register reads to only the RF during normal operation. Writes are committed to both the RF and BRF during normal operation. When the processor is in runahead mode registers are only read from and written to the BRF. Thus the sequential state of the architected register file in the RF is protected, while allowing runahead instructions to have access to a register file.

The drawback with this approach is that the BRF will typically contain some number of stale values at the conclusion of each runahead episode. These stale values are generated by writing runahead instruction results to the BRF, which will almost always differ from their checkpointed counterparts in the RF. If these stale values in the BRF are overwritten with fresh results during normal operation before the next runahead episode begins, then the performance of this register file configuration will be on a par with that of the baseline runahead register file configuration. Any stale values are not overwritten will be INV at the start of the next runahead episode. This can affect performance by reducing the number of load and store addresses that can be computed during runahead, as well as introducing wrong path effects. The generation of additional erroneous prefetches can be prevented by adding a "stale" bit to every register in the BRF. These extra bits are contained in a vector, which we call the SRV, or Stale Register Vector, which is similar to the IRV. The exact operation of this new interface to the register files, which we refer to as RA_NOCOPY, is described using pseudo-code in Figure 7.1.

Figure 7.1 Operation of the RA_NOCOPY register file implementation

1. When runahead is initiated:

```
for(X = 0; X < 31; X++) {
    if(SRV[X] == STALE)
        IRV[X] = INV; // stale values in the BRF show up as INV here
                      // at the start of the runahead episode
    else
        IRV[X] = VALID;
    SRV[X]    = NOT_STALE;
}
SRV[31] = NOT_STALE; // r31 is hardwired to zero
IRV[31] = VALID;
```

2. Instruction I writes Result R to register X:

```
if(I->runahead == TRUE) {
    SRV[X]    = STALE;  // BRF[X] is only stale after end of RA episode
    IRV[X]    = R->valid;
    BRF[X]    = R->value; // BRF value is now "ahead" of seq state in RF
}
else {
    SRV[X]    = NOT_STALE;
    BRF[X]    = R->value;// previously stale values get fresh data now
    RF[X]     = R->value;
}
SRV[31] = NOT_STALE;
IRV[31] = VALID;
```

3. Instruction I reads operand O from architected register X in the register set:

```
if(I->runahead == TRUE) {
    O->value  = BRF[X];
    O->valid  = IRV[X];
}
else {
    O->value  = RF[X];
    O->valid  = TRUE;
}
```

7.2 Eliminating the BRF altogether

It is possible to further reduce the cost of implementing runahead by eliminating the BRF. Checkpointing the sequential state of the RF can then be accomplished by simply preventing runahead instructions from updating the sequential state of the architected register file in the RF, and relying upon the pipeline forwarding paths to supply runahead instructions with the results of prior runahead instructions.

A drawback of this approach is that register updates during runahead, that are not forwarded before their instruction exits the pipeline, are forever lost to any subsequent RAW dependent instructions in that runahead episode. Since these instructions do not update the RF, any attempt to read the destination register in the RF by the dependent runahead instruction will result in the reading of a stale value. The processor should keep track of these stale values in order to keep the number of erroneous prefetches to a minimum. This can be done by adding a "stale" bit to every register in the RF, which as with the RA_NOCOPY implementation is referred to as the SRV, or Stale Register Vector. If a register is the target of an attempted update during runahead, then its stale bit is set to the STALE state. Runahead instructions treat operands marked STALE in the same way that INV register operands were in the previous runahead processor models. Note that register file valid bits are not needed as INV values are a proper subset of STALE values. The operation of this register file implementation, which we refer to as RA_NOBRF, is described using pseudo-code in Figure 7.2.

Figure 7.2 Operation of the RA_NOBRF register file implementation

1. When runahead is initiated:

```
for(X = 0; X < 32; X++) {
    SRV[X]    = NOT_STALE;
}
```

2. Instruction I writes Result R to register X

```
if(I->runahead == TRUE) {
    SRV[X]    = STALE;// This bit also indicates that the RF[X] is INV
    // The value in the RF is now "stale" for all subsequent instrs
    // during this runahead episode
    SRV[31] = NOT_STALE; // r31 is hardwired to zero
}
else {
    RF[X]     = R->value;
}
```

3. Instruction I reads operand O from architected register X in the register set

```
if(I->runahead == TRUE) {
    O->value  = RF[X];
    if(SRV[X] == STALE)
        O->valid= FALSE;
    else
        O->valid= TRUE;
}
else {
    O->value  = RF[X];
    O->valid  = TRUE;
}
```

7.3 Simulation Results

We have performed a number of runahead simulations to evaluate our new register file implementations. These were performed with perfect L1 instruction caches and coalescing store queues, as was done for the studies in Chapter 6. We also decided to perform these simulations both with and without the L2 data cache. Eliminating the L2 data cache increases the length of the average runahead episode. This will increase the number of stale

registers in the BRF at the initiation of runahead for the RA_NOCOPY model, and will cause the RA_NOBRF model to rapidly run out of VALID registers.

7.3.1 CPI

The CPI for each of the benchmarks is provided in Figures 7.3 through 7.7. The CPI for each processor configuration is given via a stacked bar, which illustrates the CPI contribution of each instruction class. The overall CPI of each configuration is provided as a number on top of each stacked bar, along with a percentage reduction in overall CPI over that of the non-runahead processor. The percentage reduction in CPI for loads and stores over that of the equivalent non-runahead processor is provided to the right of each stacked bar. Processor configurations that include an L2 data cache are on the right side of each figure, while those that do not include an L2 data cache are to the left.

The CPI figures for the GO benchmark are shown in Figure 7.3. The simplified register file models do not drastically affect performance for this benchmark as long as the processor has an L2 data cache. The runahead processor that does not perform the register save and restore operation between the RF and BRF (RA_NOCOPY) incurs only a slight increase in CPI (1.5%) over the runahead processor that does perform the save and restore operation (RA). When the BRF is eliminated (RA_NOBRF) the CPI increases over that of the baseline runahead processor by 6%. However this is still a 9% improvement over the CPI of the non-runahead (NORA) processor. These relatively good performance numbers for the simplified register file runahead processors worsen significantly when the L2 data cache is eliminated. This reduces the ability of the RA_NOCOPY processor to generate useful prefetches. This can be seen in Figure 7.3, where the RA_NOCOPY processor without an L2 data cache has a CPI that is 16% greater than that of the RA runahead processor. Note

that this is still a 28% reduction in CPI over the non-runahead (NORA) processor. The RA_NOBRF processor performs even worse, with a 29% increase in CPI over the RA processor, corresponding to a 20% reduction in CPI over the NORA processor. A prefetch utility plot for GO is provided in Figure 7.8. Note that the RA_NOCOPY processor generates almost as many useful prefetches as the RA processor, unfortunately this is offset by a doubling in the number of useless prefetches. As we expected, the RA_NOBRF processor generates most of its prefetches very early in the average episode. These prefetches are generated early enough to keep wrong path effects to a minimum, as relatively few useless prefetches are generated. Unfortunately RA_NOBRF only generates about half as many prefetches as the other processor models.

The performance of the simplified register file runahead processors for the PERL benchmark, shown in Figure 7.4, is similar to that for GO. For the processors with L2 data cache, eliminating the BRF save and restore operation (RA_NOCOPY) results in a 5% increase in CPI over the baseline runahead processor (RA), while eliminating the BRF completely (RA_NOBRF) results in a 12% increase in CPI. Eliminating the L2 data cache worsens performance considerably, with the RA_NOCOPY processor increasing CPI by 14%, and the RA_NOBRF processor increasing CPI by 31% over the RA processor. A prefetch utility plot for PERL is provided in Figure 7.9. Note that RA_NOBRF produces very few useless prefetches. Other than that, the plot is similar to that for GO.

The performance for the IJPEG benchmark, shown in Figure 7.5, was similar in some respects to that obtained for the GO and PERL benchmarks. There was virtually no penalty for either of the simplified register file models as long as an L2 data cache was included in the processor models. Eliminating the L2 data cache led to a modest gap in performance,

with the RA_NOCOPY model incurring a 6% penalty, and the RA_NOBRF model incurring a 10% penalty over that of the RA processor. A prefetch utility plot for IJPEG is provided in Figure 7.10. As with PERL, RA_NOBRF produces very few useless prefetches. Other than that, the plot is similar to those for GO and PERL.

Interestingly, the RA_NOCOPY configurations were able to achieve virtually identical performance to the RA processors for the VORTEX benchmark. This can be seen in Figure 7.6. The RA_NOCOPY incurred a CPI penalty of approximately 2% over that of the RA processor, even when the L2 data cache was eliminated. The performance of the RA_NOBRF processors was not as impressive, with a 32% penalty for the processor model without L2 data cache, and a 12% penalty for the processor model with an L2 data cache. A prefetch utility plot for VORTEX is provided in Figure 7.11. As with PERL and IJPEG, RA_NOBRF produces very few useless prefetches.

As usual the performance of the STREAM benchmark, as shown in Figure 7.7, is particularly interesting. Eliminating the RF save and restore operation, while retaining the BRF, leads to an 8% reduction in CPI for RA_NOCOPY over RA without the L2 data cache, and a 10% reduction when the L2 data cache is included in the processor models. The reason for this is rather subtle. All other things being equal, stale registers in the BRF delay the onset of prefetching in the average runahead episode for RA_NOCOPY, as stale registers must be overwritten with fresh data before they can be used to generate VALID prefetch addresses. This can be seen in Figure 7.12, which illustrates the prefetch utility for the RA, RA_NOCOPY, and RA_NOBRF processors. Note that many of the RA_NOCOPY prefetches are generated late in the average runahead episode, and that both processor configurations produce virtually the same number of prefetches (note the spikes at the x = 100

point for the RA_NOCOPY and RA useful prefetch curves). Delaying the onset of prefetching makes it more likely that the prefetches that are generated will be serviced by the time that the processor actually executes the prefetch-generating instructions during normal operation. A problem with this is that it is highly dependent upon the processor staying on the correct path late into the average runahead episode, which is especially difficult to do when runahead is initiated with stale values in the register file. This is not a problem for STREAM, which has a 0% branch misprediction rate during both runahead and normal operation. Finally, note that the RA_NOBRF model is able to produce nearly as many prefetches as the other processor models. The performance gap is due to two factors. First, there are not quite as many prefetches generated, accounting for some of the gap. Second, the prefetches that are generated are generated early in the episode. These prefetches are less likely to be serviced by the time the processor actually needs the data. At first glance it appears that it should not matter at what point a prefetch is generated in the episode. However, overlap between runahead episodes makes it more likely that prefetches generated late in a given episode will be serviced before their corresponding prefetch generating instruction is executed during normal operation.

In general, the worsening relative performance of the RA_NOBRF processor when the L2 data cache is removed is a result of the increased length of the average runahead episode. As this register file configuration does not allow register file updates during runahead, the number of stale registers in the RF increases dramatically the longer the processor remains in runahead mode. This is of particular importance when the L2 data cache is removed, as the increased length of the average runahead episode increases the degree of overlap with prior episodes. This increased overlap moves the bulk of the prefetches generated with the

RA processor model farther into the average runahead episode. Unfortunately, when the processor is prevented from updating the register file during runahead it is more likely to be unable to compute prefetch addresses at this increased distance into the average episode, resulting in many missed opportunities to prefetch and a corresponding decrease in runahead effectiveness for the RA_NOBRF scheme.

Similarly, the performance of RA_NOCOPY relative to RA generally worsens when the L2 data cache is removed. This is again the result of the longer average runahead episode length, which means that more registers are updated in the BRF on average during runahead, resulting in more stale values in the BRF at the conclusion of the average runahead episode than there would be if the processor had an L2 data cache. The presence of stale registers in the BRF at the initiation of runahead also makes it less likely that the RA_NOCOPY processor will stay on the correct path, further lowering performance.

Interestingly, there is a subtle advantage to not having a BRF in that when register file writes are not allowed during runahead, ALL of the registers in the RF are non-stale until they are the target of a retired runahead instruction, at which point they become stale. This is not true for the RA_NOCOPY scheme, in which on average several registers in the BRF are stale when runahead is initiated. This means that eliminating the L2 data cache will typically result in worsened performance for RA_NOCOPY scheme relative to the RA scheme.

Figure 7.3

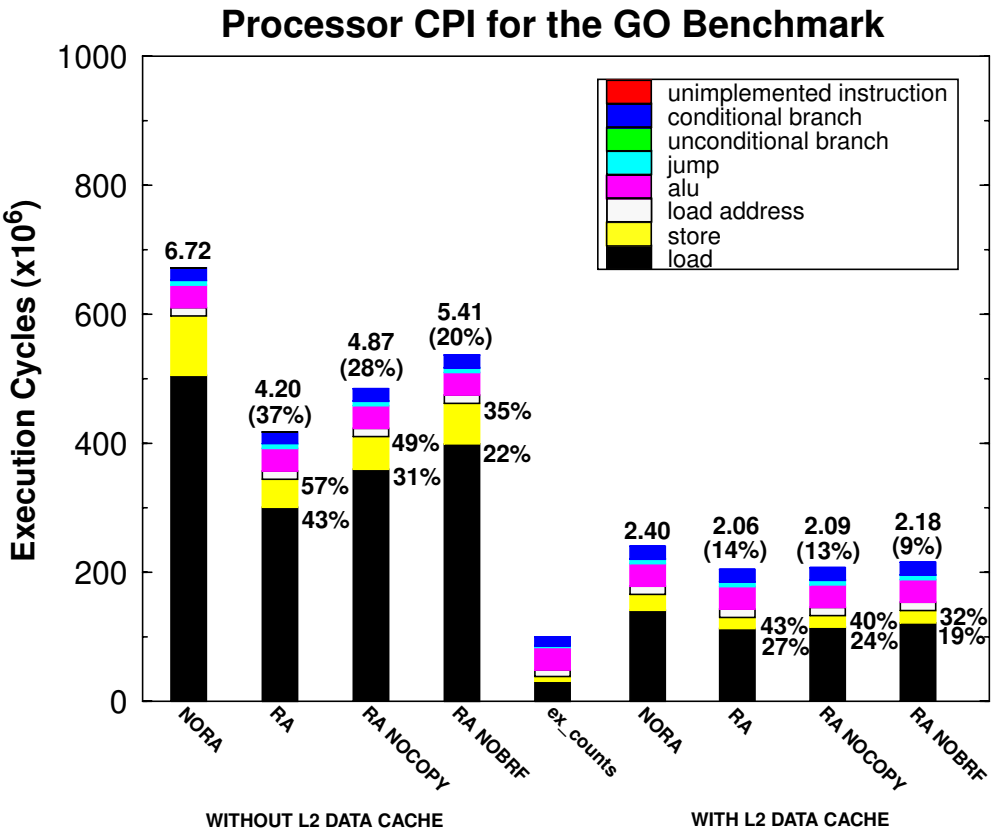
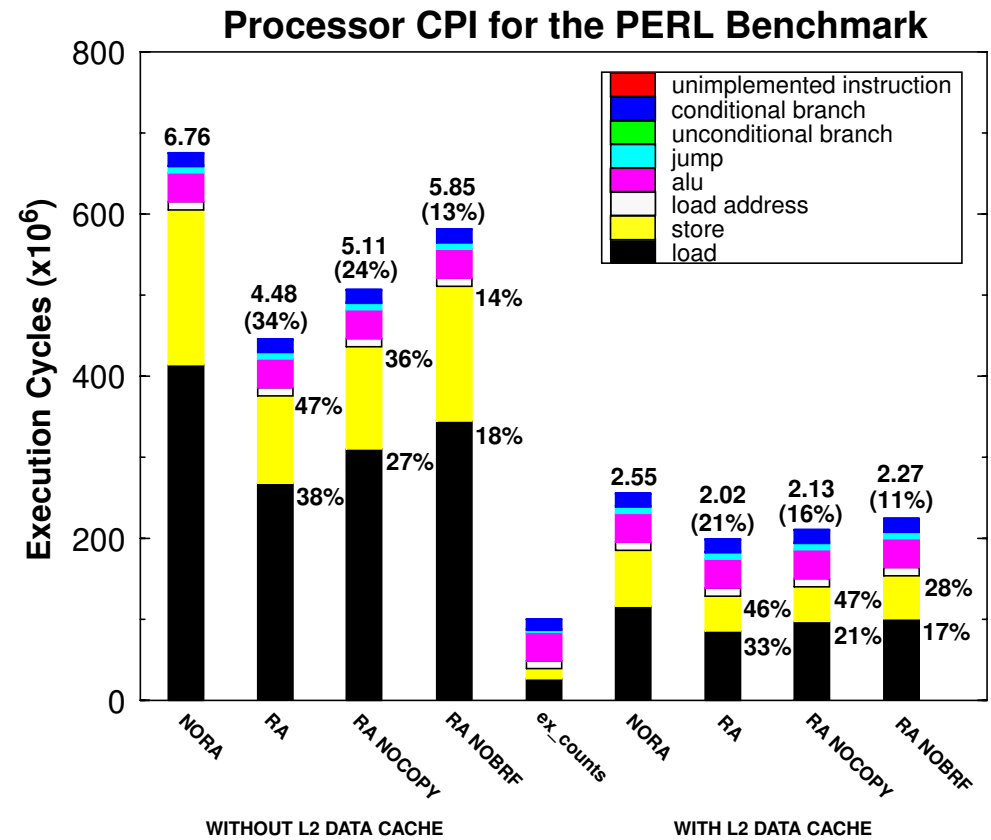


Figure 7.4



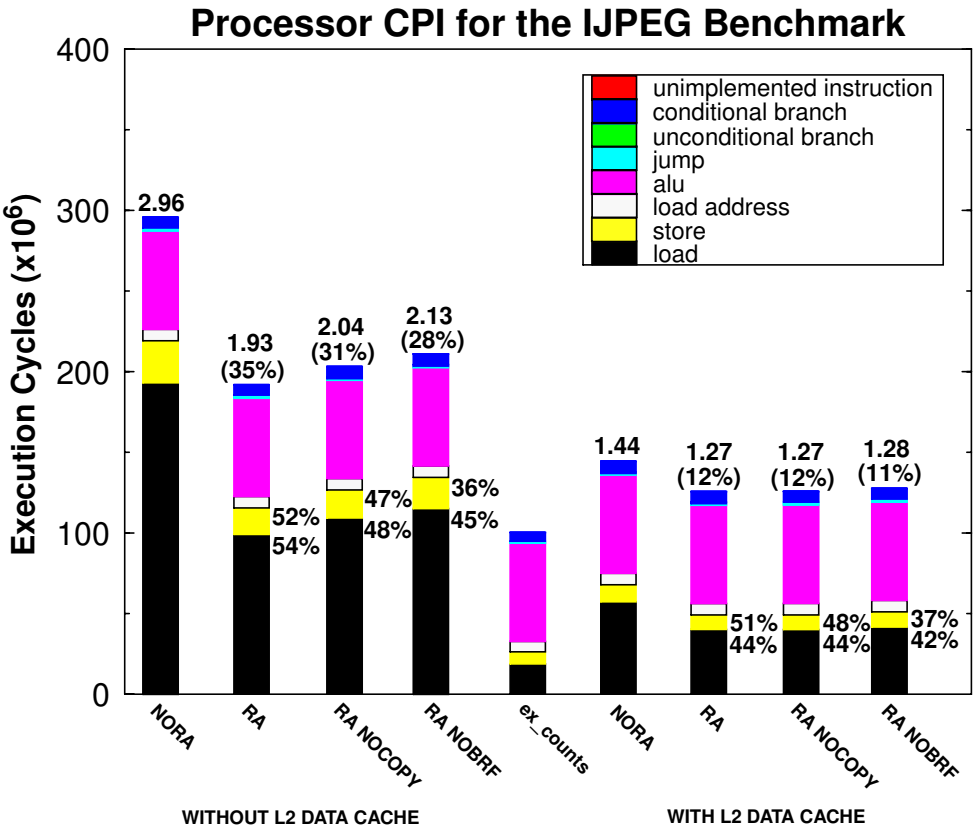


Figure 7.5

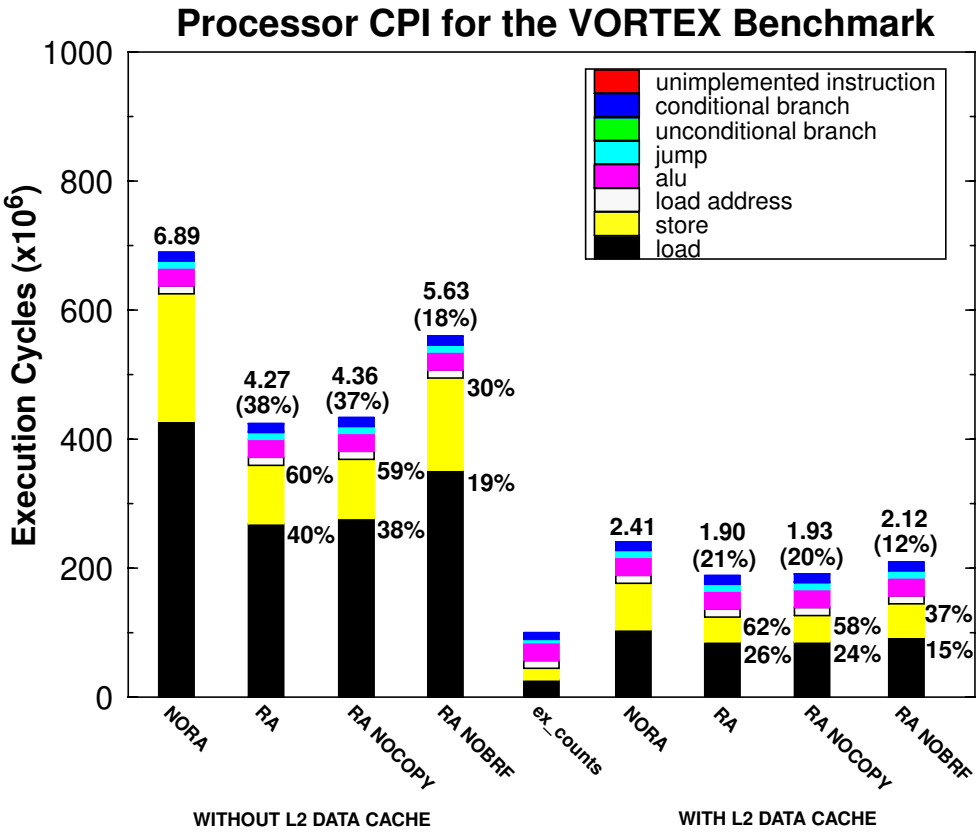


Figure 7.6

Figure 7.7

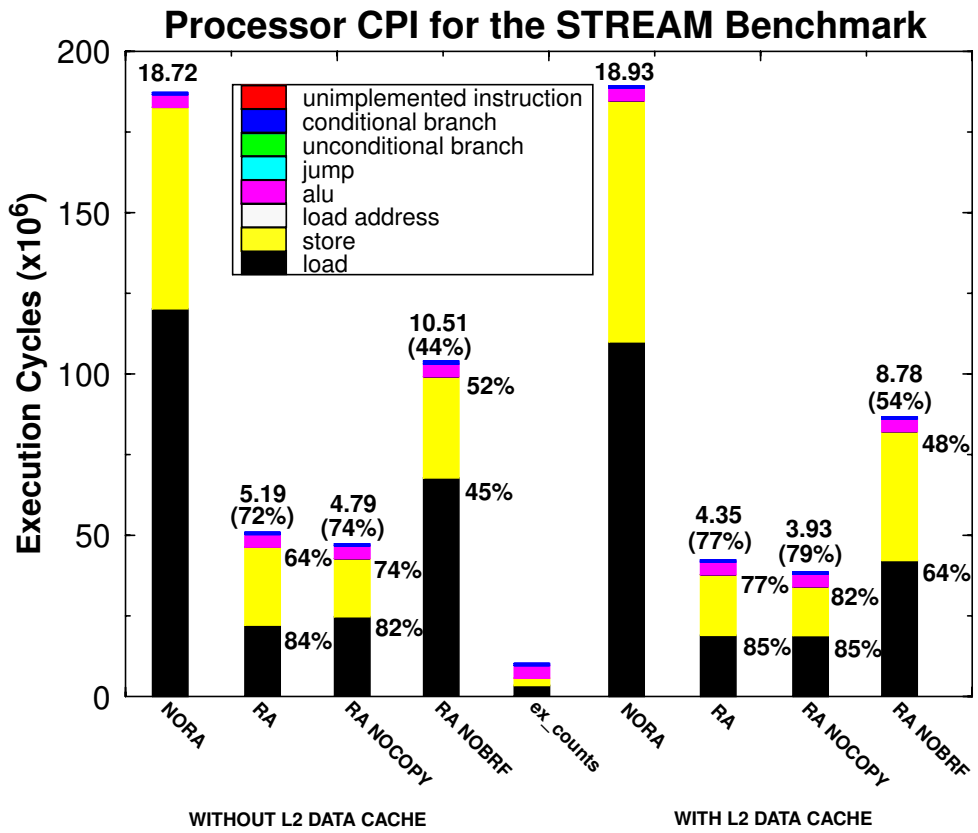
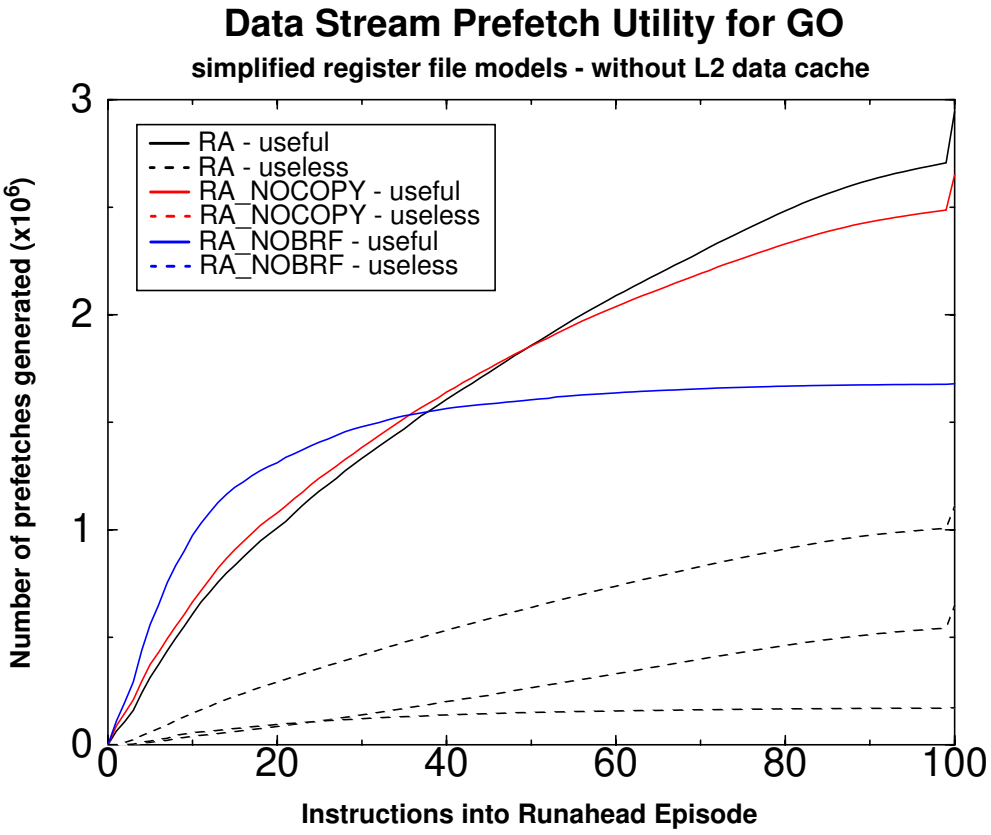


Figure 7.8



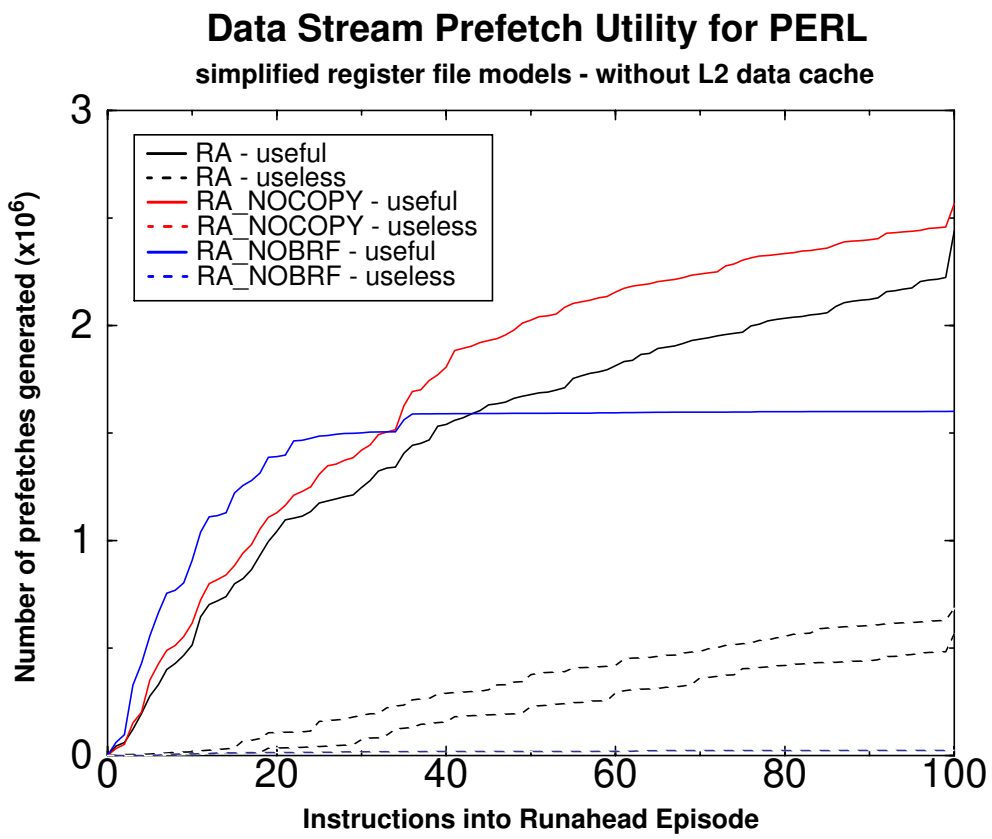


Figure 7.9

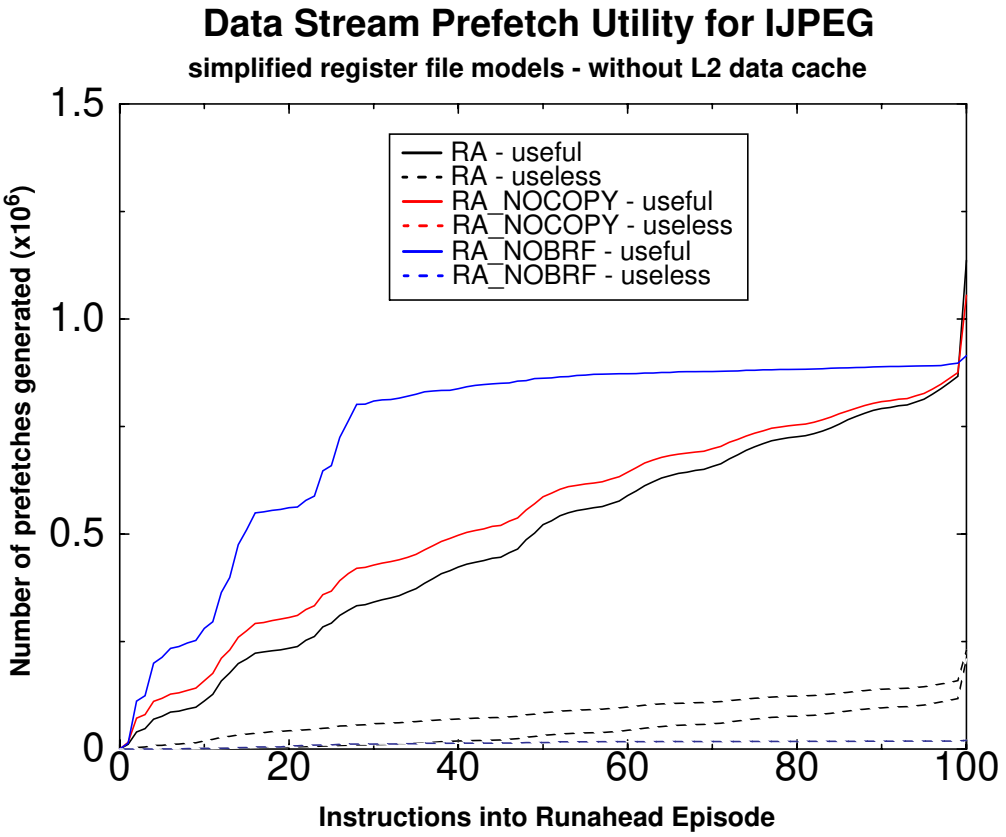


Figure 7.10

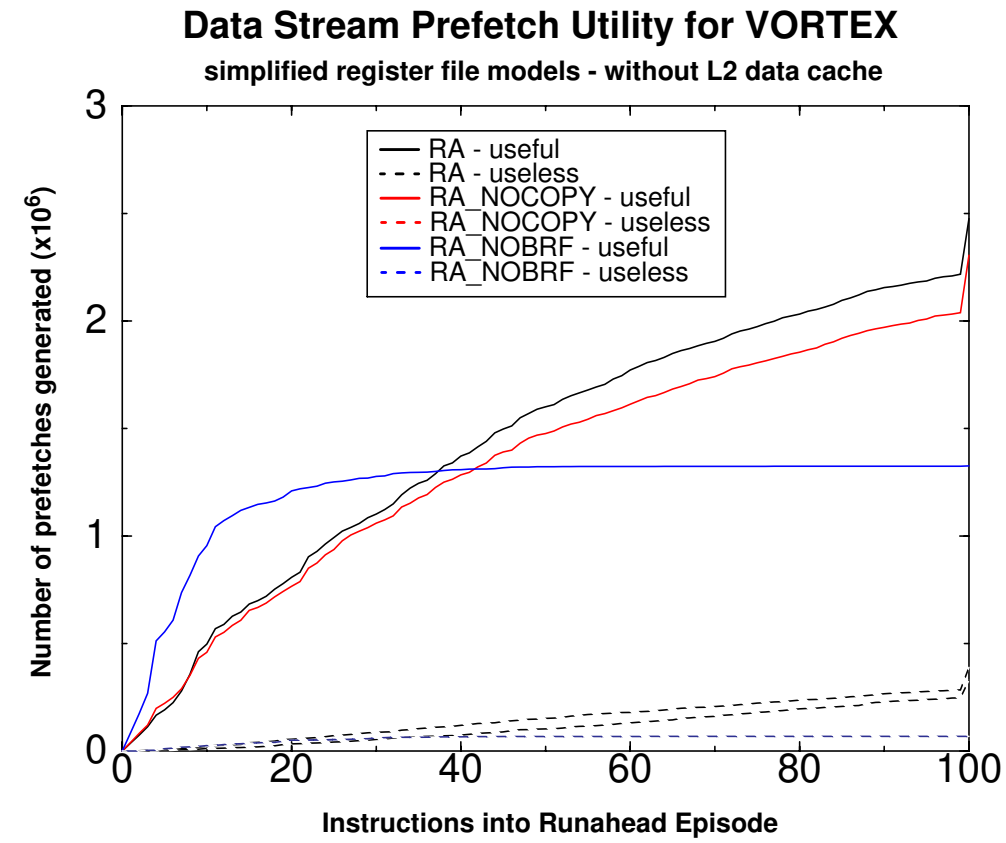


Figure7.11

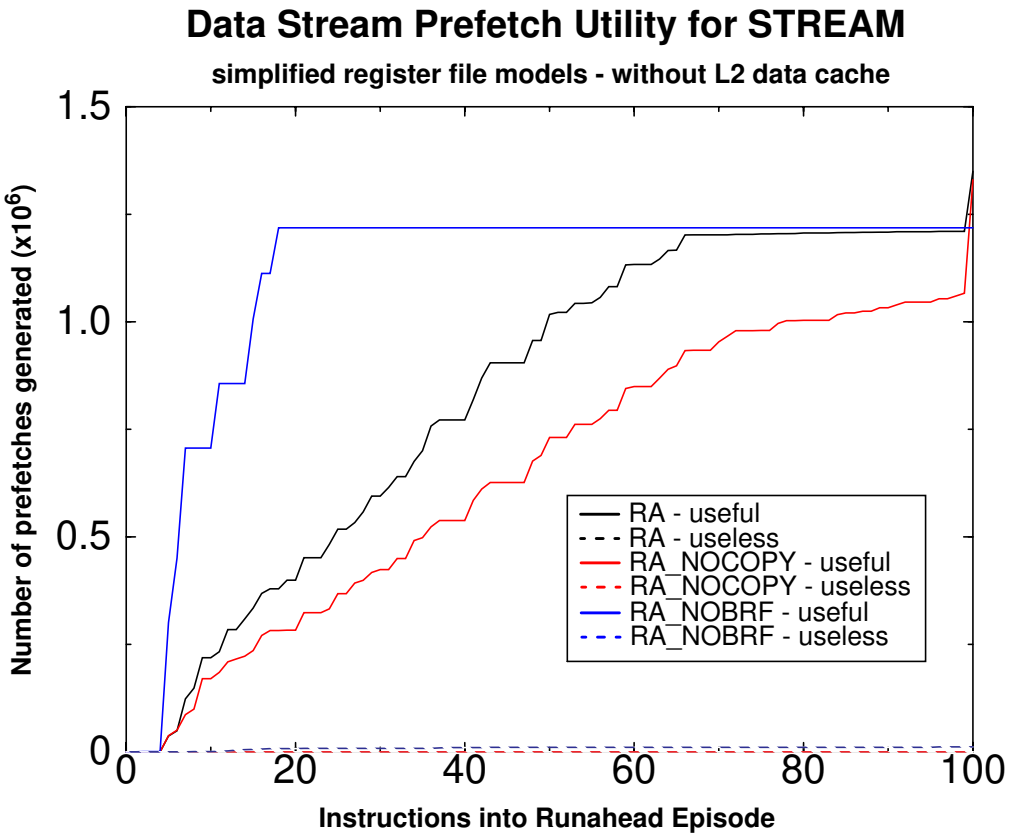


Figure7.12

7.3.2 Stale Registers

As was mentioned earlier, the RA_NOCOPY processor configuration relies upon register file writes to both the RF and the BRF during normal operation to keep the state of the two register files coherent enough to allow the processor to generate useful prefetches during runahead episodes using only the values in the BRF. Any registers in the BRF that have been updated during a given runahead episode are considered stale during normal operation until they have been overwritten with fresh values. Stale values in the BRF at the start of the next runahead episode can cause the processor to go down the wrong path, generate useless prefetches, or miss opportunities to generate useful prefetches.

In order to get an idea of how effective register file updates during normal operation are at keeping the RF and BRF contents coherent, we recorded which registers in the BRF were stale at both the conclusion and initiation of runahead episodes. This gives us a stale count at runahead initiation and conclusion for each register in the BRF. Combining this information with the number of runahead episodes allowed us to compute the probability that a given register in the BRF is stale at both the conclusion and initiation of runahead episodes for the RA_NOCOPY processor configuration.

This information is provided in Figures 7.13 through 7.17. The bars on the top half of each figure represent the probability that each register is stale at the conclusion of the average runahead episode, while the bars on the lower half of each figure represent the probability that each register remains stale at the initiation of the next runahead episode. The lower bars are always of a lower magnitude than the upper bars due to the fact that BRF registers cannot become stale during normal operation. The average probability that a register in the BRF is stale is provided to the right of each collection of bars. Note that this average is for

only 31 registers, since r31 can never become stale as it is hard wired to the value zero. Finally, these figures provide data for two processor configurations: one with an L2 data cache, and one without. Removing the L2 data cache dramatically increases the length of the average runahead episode for most programs, which should also increase the probability that BRF registers are stale.

The stale value probabilities for the GO benchmark are provided in Figure 7.13. Note that, as predicted, the probability that the average register is stale at the conclusion of runahead is significantly higher for the processor that does not have an L2 data cache. These registers are also more likely to remain stale at the initiation of the next runahead episode, delaying the onset of prefetching. The plots for the rest of the SPEC benchmarks are similar; we do not discuss them here for this reason. Note that the data cache behavior of the STREAM benchmark is such that the probabilities are essentially unchanged when the L2 data cache is removed. This can be seen in Figure 7.17.

Figure7.13

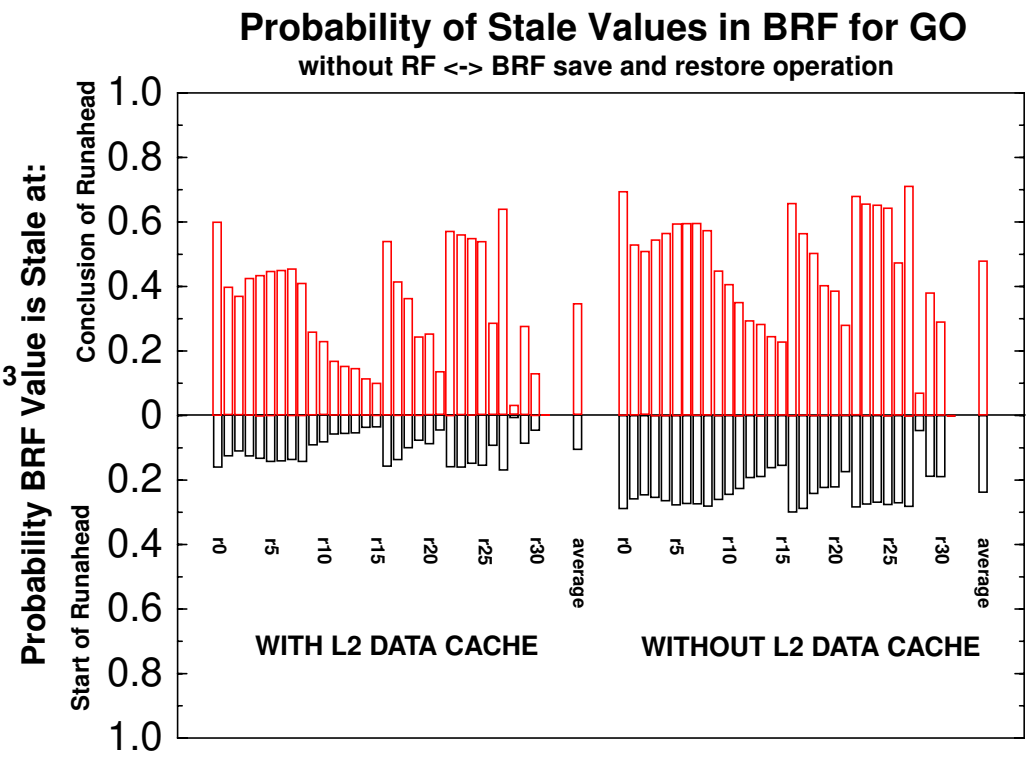


Figure7.14

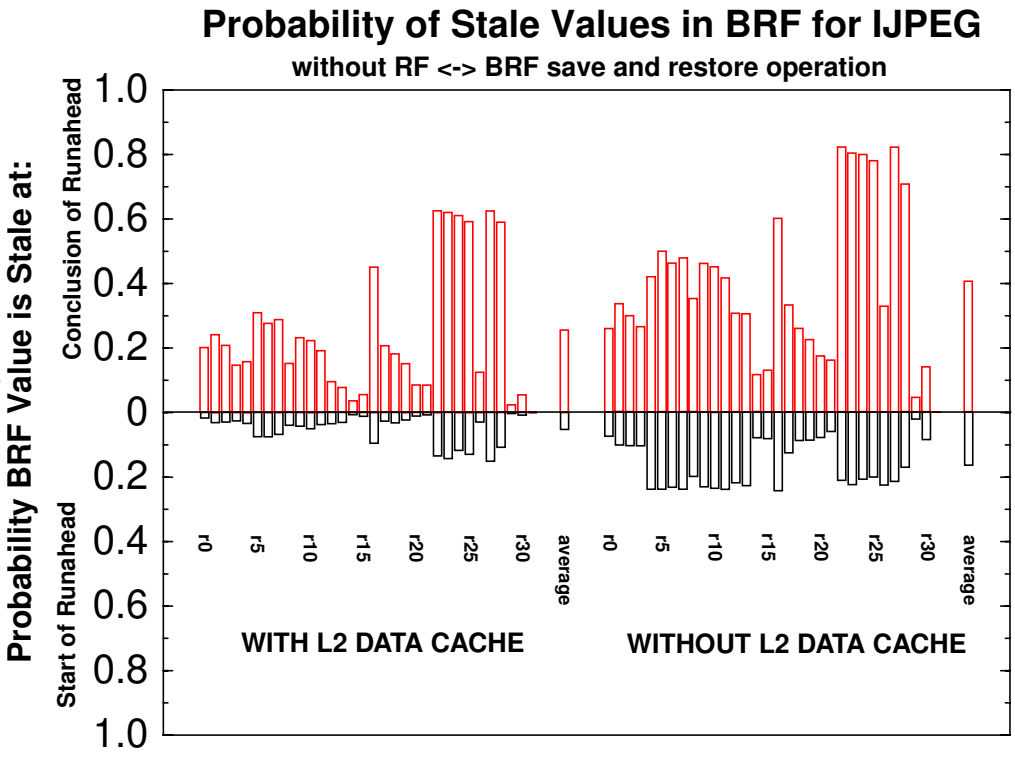


Figure7.15

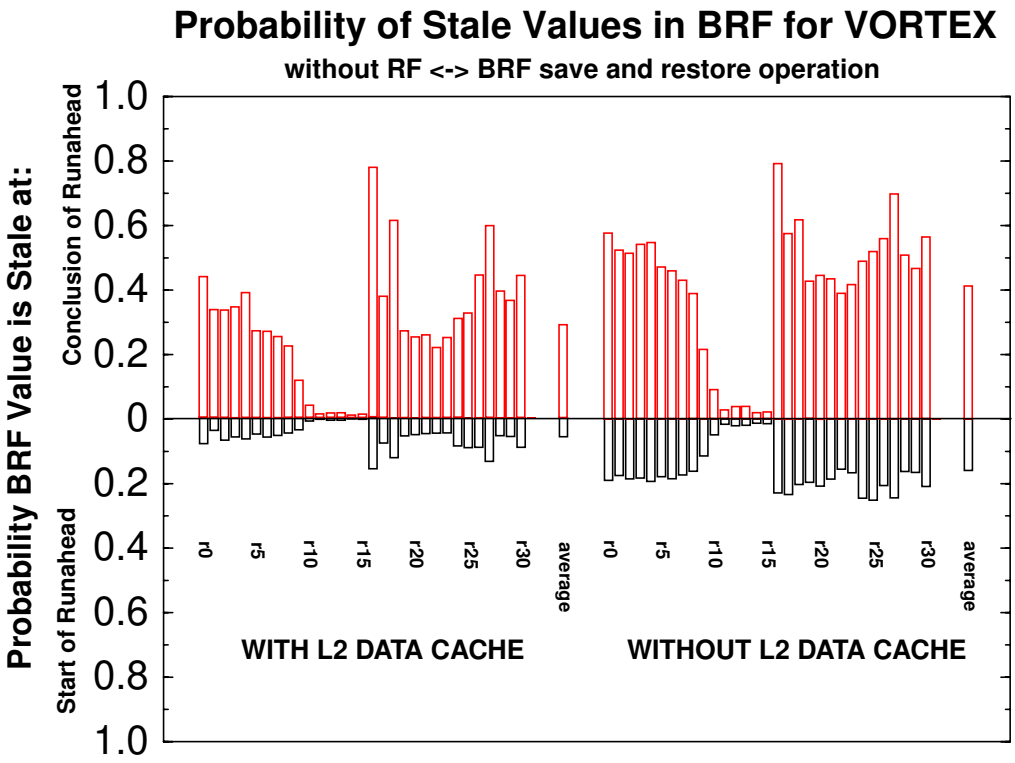


Figure7.16

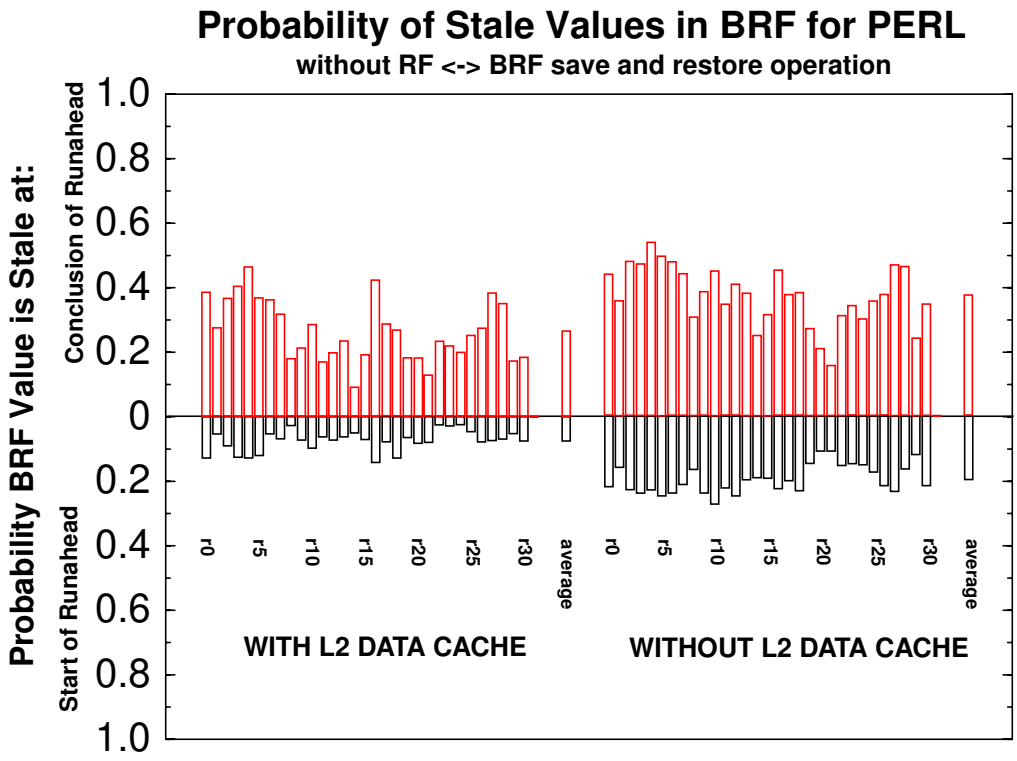
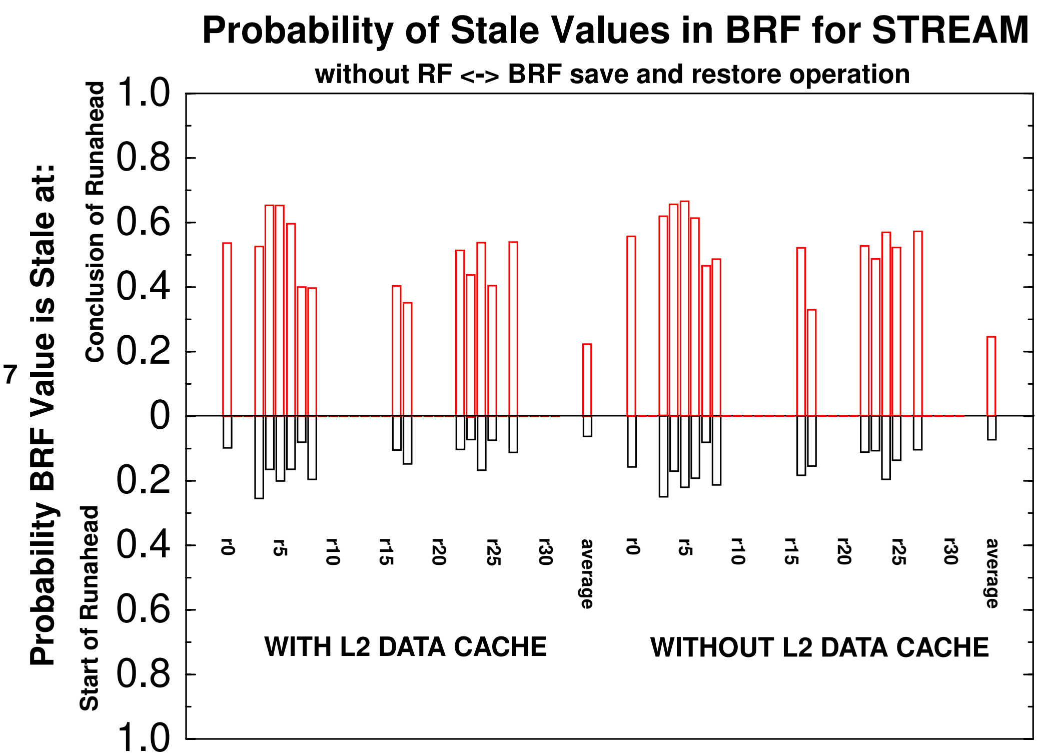


Figure7.17



Improving the performance of the RA_NOCOPY processor configuration

The stale registers in the BRF at the initiation of runahead for the RA_NOCOPY scheme can be compensated for as their counterparts in the RF are, by definition, not stale. As the processor knows which registers are stale in the BRF, it is a relatively simple matter, in theory, for the processor to provide a non-stale value from the RF when its counterpart in the BRF is stale during runahead. This can be done by providing yet another set of bits to the BRF on a per-register basis, which we can call the REVERSE bits. These new bits are used to hold the state of the SRV bits at the initiation of runahead (recall that the SRV bits are set to the NOT_STALE state after using their values to modify their counterparts in the IRV). The REVERSE bits now indicate whether or not a particular register in the BRF is stale as a result of a write during the previous runahead episode. When a register is the target of a write during runahead its REVERSE bit is cleared, and its SRV bit is set (as before).

If a register in the BRF contains a stale value from the last episode, and its REVERSE bit is set, then a non-stale value can be read from the RF instead of the BRF. This complicates the RA_NOCOPY scheme somewhat, and as a result we did not simulate this approach. Even so, it would be much easier to implement the REVERSE bits than the RF-BRF save and restore operation used in the baseline runahead register file configuration.

7.4 Eliminating the L1 data cache runahead valid bits

The L1 data cache valid bits that were employed in all of the runahead processor configurations considered up to this point are another source of potential savings. Requiring every word in the L1 data cache to have a dedicated runahead valid bit can require a significant amount of storage: 2K bits of storage for the 8KB L1 data cache that we assumed for our simulations. This is twice the number of bits required to implement a 32x32 register file, ignoring implementation details of course.

Eliminating the runahead valid bits in the L1 data cache also simplifies the pipeline to some extent, as the validity of load results during runahead is now a function of only the validity of the load address register and the presence of the target line in the L1 data cache: the status of the non-existent runahead valid bit is removed from this logic. Of course, this simplification does not come without potential problems. Removing the L1 data cache runahead valid bits means that there is now no way for the processor to detect dependencies between runahead stores and subsequent RAW dependent loads during the same runahead episode. This cannot cause improper program execution, but it can cause a degradation in the performance gains obtainable via runahead.

In order to evaluate the potentially negative effects of eliminating the runahead valid bits from the L1 data cache we simulated two different runahead processor configurations,

referred to as RA_NOVALID. Neither have runahead valid bits in their L1 data cache. They assume that all data read from the L1 data cache on a cache hit during runahead is VALID and free of dependencies with any stores that may have preceded the load in the runahead episode. As before, we also wanted to see what effect increasing the average runahead episode length has on performance. We use the non-runahead (NORA) and baseline runahead (RA) processor configurations, with and without the L2 data cache, as our basis of comparison. Note that the RA_NOVALID processor is identical to the RA processor in every respect except its lack of L1 data cache runahead valid bits.

7.4.1 CPI

The CPI figures for these simulations are provided in Figures 7.18 through 7.22. Our results indicate that eliminating the L1 data cache runahead valid bits results in virtually no change in performance. The only exception to this rule is the PERL benchmark, which suffered a modest reduction in performance when the L2 data cache was removed (Figure 7.20). These results indicate that eliminating the L1 data cache runahead valid bits can save a significant amount of hardware, without affecting performance.

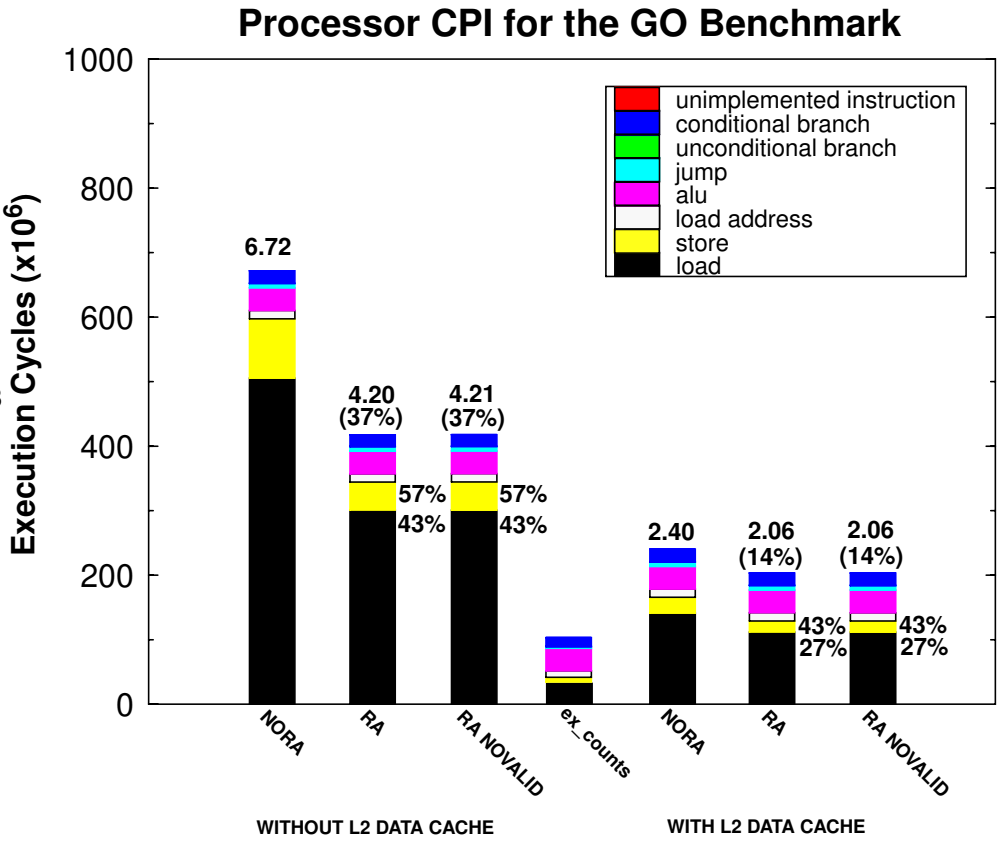


Figure7.18

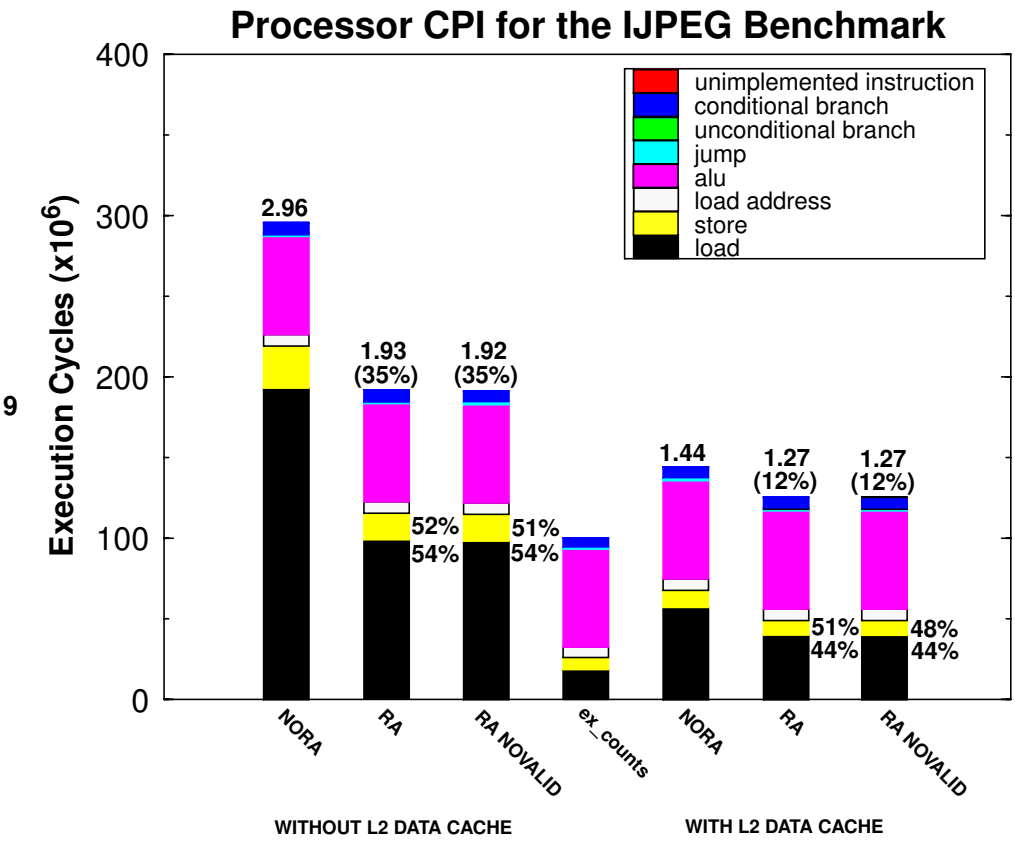


Figure7.19

Figure 7.20

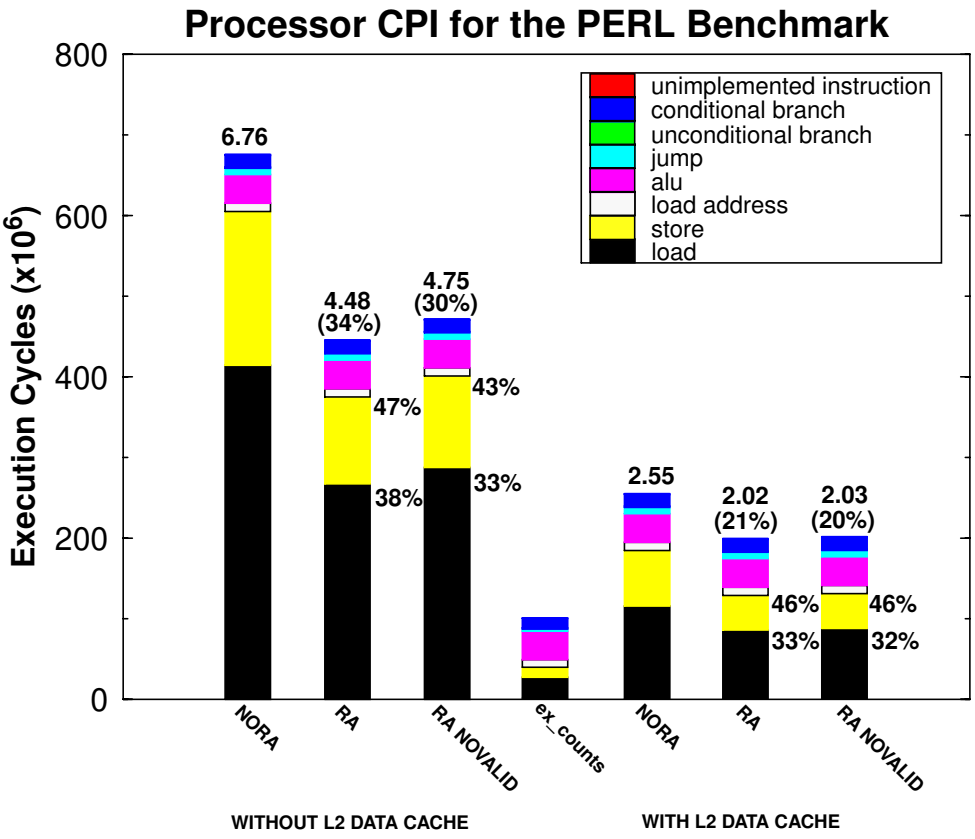


Figure 7.21

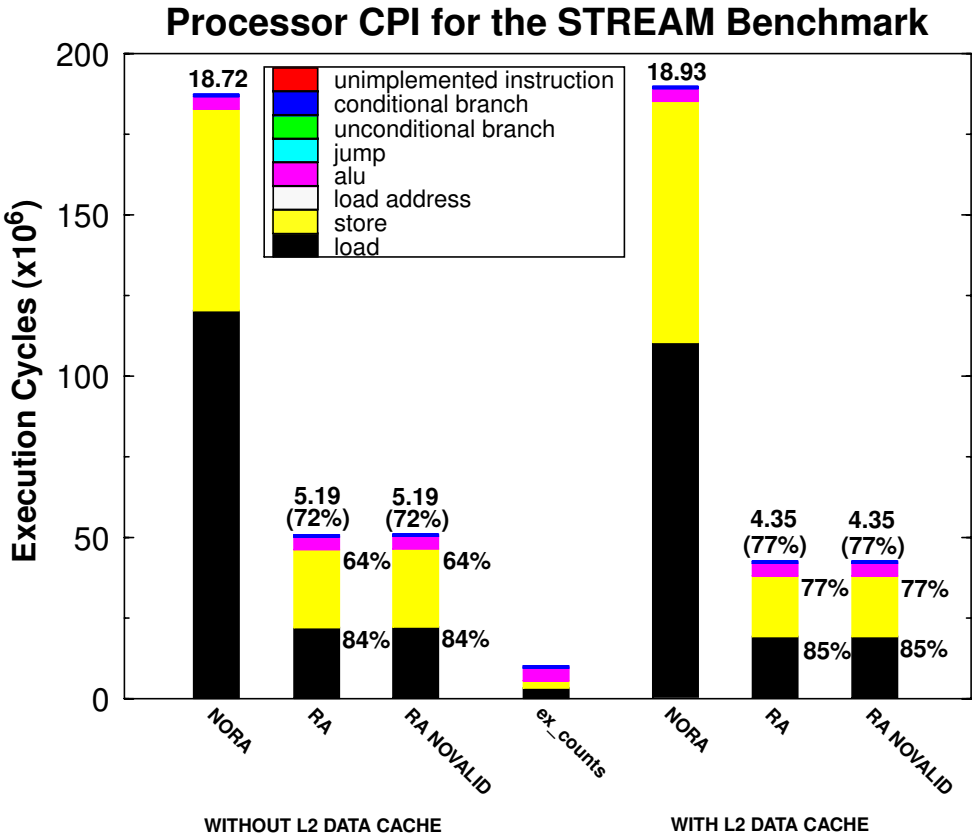
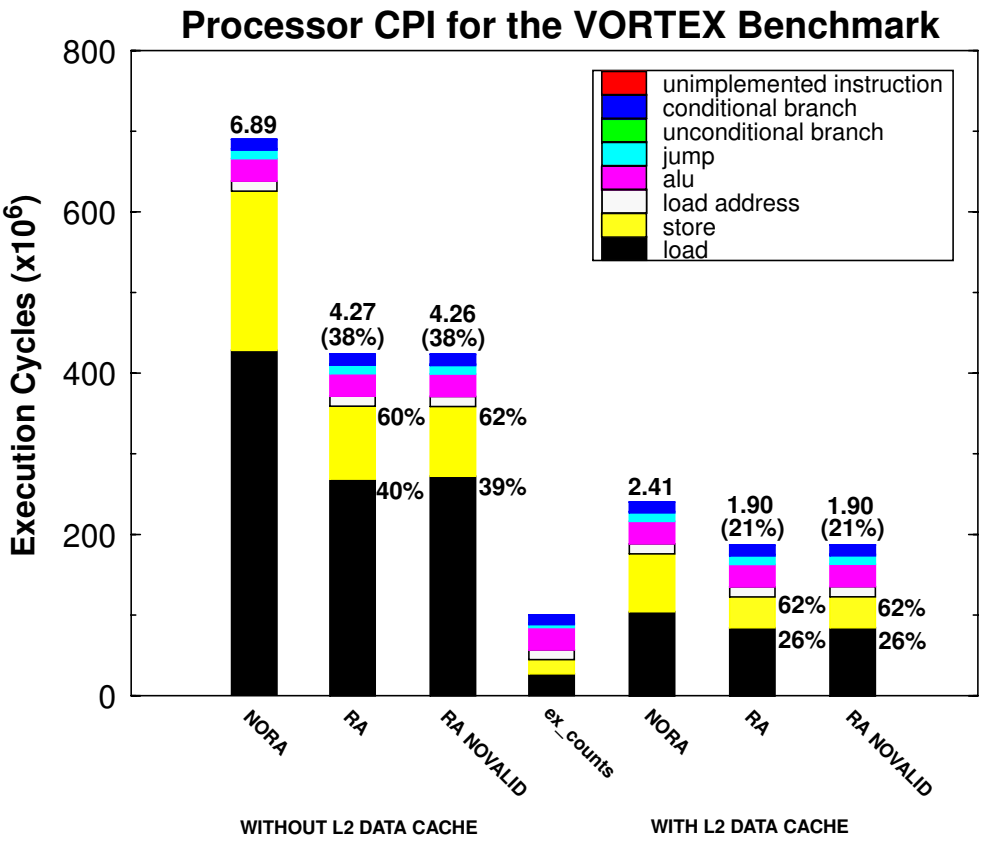


Figure7.22



7.5 Summary and Conclusions

This chapter addresses concerns about the implementation cost of runahead. Our primary concern, was that the register file save and restore operation used to copy the contents of the architected register file between two register files in a single cycle at both the initiation and completion of runahead, was too complex to implement in a reasonable fashion.

We have demonstrated that there is little effect on performance if the register file save and restore operation is eliminated by having the processor write to both the BRF and RF during normal operation, while using only the BRF during runahead. This eliminated the save and restore operation, but it requires the addition of a STALE bit to each BRF register to indicate which values in the BRF are stale at the initiation of runahead.

We also considered a runahead processor implementation that did not employ a second register file (the BRF) for use during runahead episodes. This implementation prevents runahead instructions from updating the RF, which checkpoints the architected register file state. The drawback with this approach is that the only way that would-be register file updates during runahead can communicate their values to dependent instructions is by using the forwarding paths. If a value is forwarded during runahead, then it can be used by a subsequent dependent instruction. If however, the source instruction retires before a dependent runahead instruction can forward the value, the value is lost. We found that this scaled back approach generally achieved slightly better than half of the runahead performance gains that are possible with the baseline runahead register file implementation.

As the sequential state of the memory hierarchy is checkpointed during runahead by preventing pre-processed stores from modifying the contents of memory, it is possible for pre-processed loads to read stale values from the data cache. Our baseline runahead processor appended a runahead valid bit to every word in the L1 data cache, which allows the processor to detect when a load has placed a stale value into the RF during runahead. This information is used to reduce the number of erroneous prefetches that are generated. We found that there was virtually no change in processor performance when these valid bits were eliminated. This is a significant result, as 2Kb of storage is required to implement the runahead valid bits for an 8 KB cache.

These results indicate that it is possible to use the runahead technique to good effect, while reducing the amount of hardware required to implement runahead to a very small amount.

Chapter 8

Summary and Conclusions

We have proposed a collection of processor modifications, collectively referred to as runahead, that can significantly improve processor performance. The basic idea is to use the processor pipeline to pre-process instructions during instruction and data cache miss cycles, instead of stalling. This allows a processor to generate accurate instruction and data stream prefetches by detecting many demand cache misses before they would otherwise occur. All pre-processed instruction results are discarded when normal operation resumes, as we are only interested in generating prefetches: this allows us to obtain a form of very aggressive speculation with a simple in-order pipeline.

The most important result of our experiments is that it is possible to achieve good performance with a simple in-order pipeline when runahead is employed. The implementation cost of runahead is low, with the majority of the hardware cost consisting of a means of checkpointing the sequential state of the architected register file. Checkpointing the state of the memory hierarchy is performed by preventing pre-processed store instructions from modifying the contents of memory.

Some highlights of our results are as follows. These results are for a five stage pipeline with two levels of split instruction and data caches: 8KB each of L1, and 1MB each of L2.

- Instruction and data stream prefetches generated during runahead episodes led to a significant reduction in CPI for all of the benchmarks that were examined. The reduction in CPI attributable to runahead ranges from 16% for IJPEG to 77% for STREAM.
- About 2 to 3 prefetches are generated during the average runahead episode. Increasing the length of the average runahead episode by eliminating the L2 data cache typically doubles the number of data stream prefetches that are generated during the average runahead episode.
- Data stream prefetches generated during load and store miss initiated runahead episodes are highly likely to be useful: typically at least 90% of prefetched lines are accessed at least once during normal operation. Data stream prefetches generated after an instruction cache miss is detected are less likely to be useful, however very few of these are generated.
- Instruction stream prefetches generated during runahead episodes are highly likely to be useful: typically about two-thirds of prefetched lines are accessed at least once during normal operation.
- Runahead can improve conditional branch prediction accuracy to some extent for some benchmarks. Our results indicate that attempting to save pre-processed branch outcomes for use during normal operation as predictors will not significantly improve performance in most cases.
- The price of these improvements is an increase in the bandwidth that the L2 caches and main memory must supply. Fortunately most of the bandwidth dedicated to prefetching is useful.

Some additional highlights from our studies are as follows:

Improving the effectiveness of runahead

We had hoped that by extending the length of the average runahead episode, by deleting the L2 data cache, we would be able to achieve runahead processor performance that would surpass that of a more expensive runahead processor that included an L2 data cache. This

was not the case, even though the longer runahead episodes significantly increased the effectiveness of runahead.

Failing at this, we hoped that at the very least we would be able to match the performance of non-runahead processors that included L2 data cache. This was not the case, even when we eliminated wrong path effects during runahead by using a trace of conditional branch outcomes and indirect branch targets, in addition to dramatically increasing the bandwidth of the main memory interface. The runahead processor models without L2 data cache are competitive in the sense that they do achieve performance that is reasonably close to their non-runahead counterparts that include L2 data cache, yet at a considerably lower cost.

Reducing the implementation cost of runahead

Our baseline runahead processor used a rather simplistic method of checkpointing the sequential state of the architected register file during runahead episodes, in which the RF contents are copied in a single cycle to a backup register file, or BRF, at the initiation of runahead, then restored when normal operation resumes. We found that there is little performance degradation if the register file save and restore operation is eliminated by having the processor write to both the BRF and RF during normal operation, while using only the BRF during runahead. Writing to both register sets during normal operation helps to keep the BRF contents coherent with their counterparts in the RF, however some number of stale values are typically in the BRF at the initiation of runahead.

We also considered an even simpler implementation that did not employ a second register file (the BRF) for use during runahead episodes. This implementation does not allow runahead instructions to update the RF, which checkpoints the architected register file state.

A drawback of this approach is that the only way that would-be register file updates during runahead can communicate their values to dependent instructions is by using the forwarding paths. If a runahead instruction retires before a dependent runahead instruction can forward the value, the value is lost. We found that this scaled back approach generally achieved slightly better than half of the runahead performance gains that are possible with the baseline runahead register file implementation.

Since the sequential state of the memory hierarchy is checkpointed during runahead by preventing pre-processed stores from modifying the contents of memory, it is possible for pre-processed loads to read stale values from the data cache. Our baseline runahead processor appended a runahead valid bit to every word in the L1 data cache, which allows the processor to detect when a pre-processed load has placed a stale value into the RF. This information is used to reduce the number of erroneous prefetches that are generated. We found that there was virtually no change in processor performance when these valid bits were not used. This is a significant result, as 2Kb of storage is required to implement the runahead valid bits for an 8 KB cache.

Chapter 9

Future Work

There are a number of areas in which the performance of runahead can be enhanced. We address some of these areas in this chapter.

9.1 Pre-process more than one instruction per cycle

We have seen that increasing the average latency to off-chip memory can significantly improve the effectiveness of runahead, by increasing the number of instructions that are pre-processed during the average runahead episode. Unfortunately, the increased effectiveness of runahead is not enough to completely offset the loss of the L2 data cache. Increasing the number of instructions that are pre-processed during the average runahead episode can also be accomplished by pre-processing more than one instruction per cycle. The best way of doing this would be to use runahead with an in-order multiple issue processor: an out-of-order runahead processor would be unnecessarily complex.

One of the problems with in-order multiple issue processors is the inflexibility of the in-order model: instruction issue stalls once an instruction in the window cannot issued, which can occur for a variety of reasons. Since runahead episodes are entirely speculative, and all results are discarded, a multiple-issue runahead processor could relax its instruction issue policies during runahead episodes by turning instructions that would otherwise stall the issue logic into NOPs in order to keep the issue rate as high as possible. This, plus the fact

that runahead processors do not stall on cache misses, could keep the runahead IPC significantly higher than it would be during normal operation. This increased IPC during runahead episodes may allow the processor to generate enough additional prefetches to dramatically improve performance, perhaps even enough to provide superior performance when the L2 data cache is deleted.

Finally, cache misses affect conventional in-order multiple issue processors more than their single-issue counterparts due to the insertion of multiple NOPs per cache miss cycle. This implies that a multiple-issue runahead processor may extract more benefit from pre-processing than a single-issue processor.

9.2 Runahead Co-processors

One of the problems with the runahead technique is that it relies upon demand cache misses during normal operation to initiate runahead. This is acceptable for low cost implementations, as the pipeline is productively employed during cache miss cycles. However, it may be possible to obtain superior performance by using co-processors to continually pre-process the instruction stream in parallel with a conventional processor.

It might be practical to place very simple runahead processors physically close to the L2 caches or main memory, perhaps even implementing them on the same die. As these processors would only be used for pre-processing, much of the logic associated with a conventional processor, even a rather simple one, could be discarded. For example, a large fraction of the instruction set could be omitted, with any occurrence of the missing instructions treated as NOPs. Similarly, interrupt and exception support can be deleted.

9.3 Compiler Interaction

Allowing the compiler to influence the behavior of the processor during runahead episodes appears to be very promising. An approach similar to that used with the IA-64 architecture [40] could be used to transmit a great deal of runahead-specific information from the compiler to the processor hardware. Some of the ideas that we have in this area are described in the following sub-sections.

Selectively enable prefetching or runahead for individual load and store instructions

We have seen that some types of prefetching may be inappropriate depending upon the type of runahead that the processor is engaged in. For example, data stream prefetching with loads or stores may be inappropriate during instruction cache miss initiated runahead episodes. We have also shown that prefetching for some load and store instructions may be significantly more effective than for others.

This implies that providing the processor with the ability to selectively enable or disable prefetching on a per-instruction basis may improve performance. This could be done by profiling an application and recording per-instruction runahead statistics. Instruction word bits can then be used to selectively enable or disable prefetching for each load or store instruction based upon the profiling statistics. The initiation of runahead could also be disabled on a per-instruction basis.

Provide prefetch hints to individual load and store instructions

Similarly, it may be possible to use profiling information to append prefetch hints to individual instructions in a benchmark. These hints could be used to specify the number and stride of additional prefetches that can be generated in addition to the line addressed by a specific load or store instruction, allowing the processor to generate multiple runahead prefetches per load or store miss.

Disable the pre-processing of certain instructions

A detailed profile of application behavior may indicate that certain instructions may not be useful during runahead episodes. In particular, long-latency instructions such as multiplies and divides may be of little use during runahead episodes. Also, treating some O/S calls as NOPs during runahead instead of stalling the processor may improve performance. The efficiency of runahead may be improved by disabling the pre-processing of these instructions on a per-instruction basis. This could be done by appending a pre-processing-enable bit to each instruction word.

Skip over uninteresting code

One way of improving the performance of runahead would be to have the processor skip over entire sections of code during runahead that are uninteresting. Suppose, for example, that an inner-loop contains both a load instruction and a function call. The load instruction generates useful prefetches on a regular basis, while the function call is merely a distraction that generates few, if any, prefetches. Performance could be increased by disabling the function call itself. This could be done by treating the branch instruction that performs the function call as a NOP during runahead. As with the previous section, this can be accomplished by appending a pre-processing-enable bit to each instruction.

Another possibility would be to append a skip-count to each instruction word. These could be used to specify a small branch offset for each instruction, allowing the processor to rapidly jump over small sections of code during runahead episodes.

Finally, a runahead-branch instruction could be added to the instruction set. This instruction could be used to direct the processor to an interesting section of code during runahead episodes. The compiler could even generate special runahead-only code for prefetching. Two versions of each function in a program could be generated: one to com-

pute results, and another dedicated to the efficient prefetching of data. The processor would treat runahead-branch instructions as NOPs during normal operation, and the compiler would have to insert them sparingly in order to keep code bloat to a minimum.

Bibliography

[1] Michael Upton, *Architectural Trade-offs in a Latency Tolerant Gallium Arsenide Microprocessor*, Ph.D. Thesis, The University of Michigan, 1996.

[2] John Hennessy and David Patterson, *Computer Architecture a Quantitative Approach*, Morgan Kaufmann Publishers Inc., 2nd Edition, 1996.

[3] Alan J. Smith, "Cache Memories," *ACM Computing Surveys*, vol. 18, no. 3, Sep 1982.

[4] David Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," In the *Proceedings of the 8th International Symposium on Computer Architecture,* May 1981.

[5] Richard Eickemeyer, Ross Johnson, and Steven Kunkel, "Evaluation of Multithreaded Uniprocessors for Commercial Application Environments," In the *Proceedings of the International Symposium on Computer Architecture*, 1996.

[6] Norman Jouppi, *Cache Write Policies and Performance*, Digital Equipment Corporation Western Research Laboratory Research Report 91/12, Dec 1991.

[7] Brian Bray and Michael Flynn, *Writes Caches As An Alternative To Write Buffers*, Stanford University Computer Systems Laboratory Technical Report Number CSL-TR-91-470, April 1991.

[8] Sharon Perl and Dick Sites, "Studies of Windows NT Performance using Dynamic Execution Traces," In the *Proceedings of the USENIX 2nd Symposium on Operating Systems Design and Implementation*, Oct 1996.

[9] Markus Levy, *The dynamics of DRAM Technology*, Electronic Design News, Jan 5, 1995.

[10] David Gallagher, William Chen, Scott Mahlke, John Gyllenhaal, and Wen-mei Hwu, "Dynamic Memory Disambiguation Using the Memory Conflict Buffer," In the *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 1994.

[11] Michael Golden and Trevor Mudge, "Comparison of two common pipeline structures," *IEE Proceedings on Computer and Digital Technology*, May 1996.

[12] Norman Jouppi, *Architectural and Organizational Trade-offs in the Design of the MultiTitan CPU*, Digital Equipment Corporation Technical Report 89/9, July 1989.

[13] Michael Golden, *Hardware Support for Hiding Cache Latency*, The University of Michigan Department of Computer Science and Engineering Technical Report CSE-TR-152-93, 1993.

[14] Todd Austin, Dionisios Pnevmatikatos, and Gurindar Sohi, "Streamlining Data Cache Access with Fast Address Calculation," In the *Proceedings of the 28th International Symposium on Microarchitecture*, 1995.

[15] Todd Austin and Gurindar Sohi, "Zero-Cycle Loads: Microarchitecture Support for Reducing Load Latency," In the *Proceedings of the 28th International Symposium on Microarchitecture*, 1995.

[16] Norman P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," In the *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[17] Fredrik Dahlgren, Michel Dubois, and Per Stenstrom, "Fixed and Adaptive Sequential Prefetching in Shared Memory Microprocessors," In the *Proceedings of the International Conference on Parallel Processing*", 1993.

[18] John Fu and Janak Patel, "Data Prefetching in Multiprocessor Vector Cache Memories," In the *Proceedings of the 18th International Symposium on Computer Architecture*", May 1991.

[19] Jean-Loup Baer and Tien-Fu Chen, “An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty,” In the *Proceedings of Supercomputing*, 1991.

[20] J.W.C. Fu and J.H.Patel, “Stride directed prefetching in scalar processors,” In the *Proceedings of the 25th International Symposium on Microarchitecture*, Dec 1992.

[21] Ivan Sklenar, “Prefetch unit for vector operations on scalar computers,” *Computer Architecture News* vol. 20, no. 4, 1992.

[22] Tien-Fu Chen and Jean-Loup Baer, “Effective Hardware-Based Data Prefetching for High-Performance Processors,” In *IEEE Transactions on Computers*, vol. 44, no. 5, May 1995.

[23] Shlomit Pinter and Adi Yoaz, “Tango: a Hardware-based Data Prefetching Technique for Superscalar Processors,” In the *Proceedings of the 29th Annual International Symposium on Microarchitecture*”, Dec 1996.

[24] C. Scheurich and M. Dubois, “Concurrent Miss Resolution in Multiprocessor Caches,” In the *International Conference on Parallel Processing*, 1988.

[25] D. Callahan, K. Kennedy, and A. Porterfield, “Software Prefetching,” In the *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[26] Alexander C. Klaiber and Henry M. Levy, “An Architecture for Software-Controlled Data Prefetching,” In the *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991.

[27] Edward Gornish, Elana Granston, and Alexander Veidenbaum, “Compiler-directed Data Prefetching in Multiprocessors with Memory Hierarchies,” In the *Proceedings of the International Conference on Supercomputing,* 1990.

[28] William Chen, Scott Mahlke, Pohua Chang, and Wen-mei Hwu, “Data Access Microarchitectures for Superscalar Processors with Compiler-Assisted Data Prefetching,” In the *Proceedings of the 24th International Symposium on Microarchitecture*”, 1991.

[29] Todd C. Mowry, Monica S. Lam, and A.Gupta, “Design and evaluation of a compiler algorithm for prefetching,” In the *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 1992.

[30] Mikko Lipasti, William Schmidt, Steven Kunkel, and Robert Roediger, “SPAID: Software Prefetching in Pointer- and Call-Intensive Environments,” In the *Proceedings of the 28th International Symposium on Microarchitecture*, 1995.

[31] Chi-Keung Luk and Todd C. Mowry, “Compiler-Based Prefetching for Recursive Data Structures,” In the *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 1996.

[32] Ricardo Bianchini and Thomas J. LeBlanc, *A Preliminary Evaluation of Cache-Miss-Initiated Prefetching Techniques in Scalable Multiprocessors*, University of Rochester Computer Science Department Technical Report 515, May 1994.

[33] Tien-Fu Chen, “An Effective Programmable Prefetch Engine for On-Chip Caches,” In the *Proceedings of the 28th International Symposium on Microarchitecture*, 1995.

[34] Alan Eustace and Amitabh Srivastava, *ATOM: A Flexible Interface for Building High Performance Program Analysis Tools*, Digital Equipment Corporation Western Research Laboratory Technical Note TN-44, July 1994.

[35] James Dundas and Trevor Mudge, *Using Stall Cycles to Improve Microprocessor Performance,* University of Michigan Department of Electrical Engineering and Computer Science Technical Report CSE-TR-301-96, Sept 1996.

[36] James Dundas and Trevor Mudge, “Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss,” 11th ACM International Conference on Supercomputing, July 1997.

[37] J. Smith, “A Study of Branch Prediction Strategies, In the Proceedings of the 8th Annual International Symposium on Computer Architecture, May 1981.

[38] Standard Performance Evaluation Corporation, SPEC’95 Benchmark description files distributed with source code as well as http://www.specbench.org/.

[39] John McCalpin, “STREAM: Measuring Sustainable Memory Bandwidth in High Performance Computers,” The University of Virginia, http://www.cs.virginia.edu/stream/.

[40] Linley Gwennap, “Intel, HP Make EPIC Disclosure,” Microprocessor Report, vol. 11, no. 14, Oct. 1997.