Using Stall Cycles to Improve Microprocessor Performance

James D. Dundas and Trevor N. Mudge

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122

September 1996

email: {dundas, tnm}@eecs.umich.edu

Abstract

Contemporary microprocessors typically expend a significant amount of their device budget in an attempt to reduce the detrimental effects of memory latency and branch misprediction. The extra hardware frequently reduces the clock rate and wastes valuable die area that might be more productively employed in other ways. We propose a novel approach to this problem, which is intended to be used in relatively simple microprocessors with very high clock rates. A *runahead* microprocessor continues to execute instructions past a level one data cache miss until the miss is serviced. During this time it is said to be in runahead mode. Any additional level one data cache misses that occur during runahead, whose target addresses can be calculated, can become data stream prefetches. Any conditional branches that are executed during runahead can become trial runs, which may improve branch prediction accuracy. Any level one instruction cache misses that occur during runahead become instruction stream prefetches. After the original data cache miss is serviced, the microprocessor restarts instruction execution at the address of the load or store instruction that caused the original data cache miss. The additional hardware required to checkpoint the sequential state of the microprocessor is rather modest, and should not impact the clock rate of a high-performance implementation.

1 Introduction

A runahead processor attempts to generate prefetches for data items that have a high likelihood of being referenced soon after a data cache miss is detected. This is done by checkpointing the processor state and attempting to speculatively execute instructions that are located after the point that the cache miss was generated. After the original miss is serviced, the processor restores its state and resumes instruction execution at the load or store instruction that caused the miss. Branch prediction performance can also be improved by recording the outcomes of any conditional branches that are executed while the processor is in runahead mode. Finally, any instruction cache misses that are detected while the processor is in runahead mode become instruction-stream prefetches.

1.1 Register file modifications needed to support runahead

In order to protect the architected state of the processor, it is necessary to checkpoint the registers and level one data cache before entering runahead mode. Checkpointing the registers can be done by providing two copies of the register file. The first copy, or non-runahead register file (NRA-RF) holds the sequential non-runahead state of the architected register file. A second register file, called the runahead register file (RA-RF) is used when the processor is in runahead mode. Both of the register files are written simultaneously when the processor is in non-runahead mode. When the processor detects a level one data cache miss the situation changes. Reading from and writing to the NRA-RF is disabled while the processor reads from and writes to the RA-RF exclusively. This allows the processor to attempt to generate load/store target addresses and detect data cache misses by prematurely executing future instructions while the data cache miss is being serviced, without corrupting the sequential state of the processor in the NRA-RF. When the processor leaves runahead mode the contents of the RA-RF are changed to that of the NRA-RF by performing a 1:1 copy of all the registers in the NRA-RF to their counterparts in the RA-RF. This 1:1 copy can be done in one cycle with a suitably designed register file that incorporates both the RA-RF and NRA-RF.

1.2 Keeping track of invalid registers

The primary reason for allowing the processor to enter runahead mode is to generate useful prefetches. A given prefetch is more likely to be useful if its address as generated by the corresponding load/store instruction is correct. This means that the processor should keep track of all RA-RF registers that are known to contain invalid data while the processor is in runahead mode. RA-RF registers can become invalid due to a runahead dependency by being:

- The destination of a load that missed in the L1 data cache.
- The destination of a load that hit in the L1 data cache while the processor was in runahead mode, but corresponding to a memory location that was the target of a preceding runahead store that hit in the L1 dcache.
- The destination of any instruction that occurs in runahead mode that has a source register that is marked invalid.

Most of the invalid registers can be easily tracked by maintaining an invalid register vector (IRV). The IRV is simply a vector that contains a valid bit corresponding to each of the RA-RF registers. If at any point during a runahead episode a given RA-RF register contents are invalid, then its corresponding bit in the IRV will be set to invalid. The destination register for a given instruction in runahead mode will be marked as valid if all of the register sources for the instruction are valid. If any one of the source registers are invalid, or if the instruction is a load that misses in the data cache or reads a data cache word that is marked runahead-invalid, then its destination register is marked invalid in the IRV. Note that a previously invalid register can become valid again if it is the destination of an instruction that produces a valid result. Finally note that the IRV can be stored in either the RA-RF as an extra bit per register, or in a dedicated register.

1.3 Data cache modifications needed to support runahead

Stores to all levels of the memory hierarchy are disabled while the processor is in runahead mode. Store instructions are only allowed to compute target addresses and determine if a cache miss occurred while the processor is in runahead mode. In order to maximize runahead performance, the level one data cache should be augmented with sub-block runahead valid bits for each word in the cache, which are used to detect runahead load/store dependencies. These runahead valid bits are set to valid as a group when the processor is reset or when it leaves runahead mode. The runahead valid bits for an entire dcache line are also set to valid whenever a prefetch to that line completes during a runahead episode. Whenever a word in the data cache is a known target of a store that occurs in a given runahead episode, its runahead valid bit is set to invalid. If a runahead load instruction reads a dcache word that is marked invalid, then the destination register of the load is marked invalid. Note that this approach is imperfect since the only way that a data item can be marked as invalid is if it is in the level one dcache, and that it is possible for prefetches that complete during a runahead episode to erase invalidation information via dcache conflicts. Furthermore, a runahead store cannot mark its target as runahead invalid if it cannot compute its target address due to an invalid index register.

The IRV and dcache runahead valid bits can be used to detect the majority of runahead dependencies corresponding to the cases listed above. However it is possible for a dependency to exist between a load and a store whose target address cannot be calculated with valid registers, or if the store target address can be calculated but does not hit in the level one dcache. The target word of either store situation cannot be marked as invalid with the runahead-valid bits in the data cache. Once one of these undetectable invalid registers is introduced to the RA-RF during a given runahead episode, it can propagate incorrect results to other registers in the RA-RF. These additional invalid registers cannot be detected. This can result in erroneous prefetches, as well as counterproductive updates of the branch prediction.

1.4 Branch prediction modifications

Runahead can improve branch prediction performance by "trial running" branches before they are seen by the processor in non-runahead mode. The hardware requirements for this are rather modest. A shift register is used to record the taken/not-taken outcome of each branch encountered in a given runahead episode. A count register is needed to keep track of how many valid outcomes are in the register. A second count register and two-bits of state are needed to determine when to allow the processor to add outcomes to the shift register. When the processor leaves runahead mode, any outcomes that were recorded in the shift register are shifted out one at a time for each branch that the machine encounters in non-runahead mode. The first counter is incremented every time a branch outcome is added to the shift register, and it is decremented every time a branch outcome is shifted out of the register. The processor can easily detect if the shift register contains any valid outcomes by testing the value of the first counter versus zero: a non-zero count indicates that at least one valid branch outcome is in the shift register.

Since the processor can re-execute the same branch multiple times in successive runahead episodes, it cannot simply add a new branch outcome to the shift register every time a given branch is encountered during runahead. To solve this problem, the processor uses the two-bits of state to keep track of three possible runahead branching states: ADD, WAIT, and TAKE. These three states are needed to ensure that the runahead branch prediction stays in sync with the point at which the machine enters each runahead episode. When the processor is reset, the state is set to ADD. While the processor is in the ADD state, all branch outcomes that the processor computes during runahead are added to the register until the register is full or the processor leaves runahead mode. When the processor leaves runahead mode its state is set to TAKE. While the processor is in the TAKE state during non-runahead, it can predict any conditional branches that it encounters by shifting out branch outcomes in the shift register one at a time, until there are none left or the machine reenters runahead mode. If the processor enters runahead mode with a non-zero number of predictors in the shift register, its state is set to WAIT and the second count register is set to zero. While the processor is in the WAIT state in runahead mode, it counts the number of branches that it encounters using the second count register. If the branch count in the second count register becomes equal to the number of predictors in the shift register, as recorded in the first count register, then the branching state changes back to ADD.

Note that the normal dynamic branch prediction scheme, if any, should be constantly updated whether or not the machine is in runahead mode, since it will be used to guide the processor when the shift register becomes empty and while the machine is in runahead mode. Also, note that as long as the machine stays on the correct path during runahead the shift register will accurately predict any conditional branches for which it has outcomes (to the limits of the detection of invalid registers). If the shift register yields an incorrect outcome it should probably be flushed, since this indicates that the processor was influenced by a register dependency, and that any additional outcomes in the register may be incorrect. Note that we initially considered using per-branch shift registers, which are unnecessarily complicated. Using a single global shift register to hold runahead branch outcomes was subsequently suggested by [1].

1.5 When to halt runahead prematurely

The processor will typically generate more useful prefetches if it stays on the proper path during runahead. If the processor encounters a conditional branch during runahead that is dependent upon an invalid register, then it can do one of two things. The most conservative policy is to simply halt runahead execution. An alternative is for the processor to assume that the branch prediction strategy used is good enough to accurately predict the outcome of the branch. The first policy (to the limits of the detection of invalid registers) will generate fewer useless prefetches which can pollute the dcache. However it will result in somewhat lower performance if the branch prediction could have properly predicted the branch outcome or if the incorrect path of runahead execution would have fetched useful dcache lines. The second policy may deliver good performance if it successfully predicts past the first questionable branch in a given runahead episode. Successive branches of this type will be increasingly harder to properly predict. It might be a good idea to stop updating the dynamic branch prediction scheme when an invalid-dependent branch is discovered in order to avoid corrupting the branch prediction. The processor can continue to shift branch outcomes into the runahead branch shift register safely, since it is assumed that the register will be flushed if a bad prediction bit is detected. Note that the predicted outcome from the dynamic branch prediction scheme can be used to “pad” the runahead branch shift register, if a given conditional branch cannot be resolved in runahead.

Previous work reported in [2] found that wrong path speculative loads and stores that miss in the level one data cache prefetch useful data more often than not. This counterintuitive result indicates that it may be practical for a runahead processor to simply ignore conditional branches in runahead that cannot be resolved since any additional runahead prefetches may typically prefetch useful data even if they were on the wrong path of execution. However, note that [2] assumed a constant fifty cycle speculation depth while a runahead processor can potentially runahead much farther down a wrong path.

Another reason to halt runahead is when a level one instruction cache miss occurs. The runahead icache miss then becomes an istream prefetch. While it might be possible to have the processor skip over the instructions in the missing icache line and continue runahead at the next sequential line, the possibility of introducing undetectable register dependencies due to the un-executed instructions may make the likelihood of generating additional useful prefetches low. The size of the level one icache line is an important factor since a shorter line that is skipped will introduce fewer undetectable register dependencies on average than a longer line. This trade-off will be examined as part of our future work. Another possible action on an icache miss is to continue runahead down a different path of execution which may be resident in the icache. By checkpointing the runahead registers and saving the non-predicted target address of the most recent conditional branch encountered in runahead mode, the processor can “back-up” around a runahead icache miss, and restart execution on another path. Note that this additional checkpointing essentially doubles the hardware cost of implementing runahead.

1.6 Some Runahead Examples

An example sequence of code is shown in Figure 1. Note that the sub-block valid bits in the dcache are not shown for this example, and that only the first eight general purpose registers are considered.

Figure 1. Basic runahead example

```
                                   .
                                   .        r r r r r r r r
                                   .        0 1 2 3 4 5 6 7
dcache miss        load r1, 0(r2)  | 1 0 1 1 1 1 1 1 |  IRV
invalid result     add r3, r1, r2  | 1 0 1 0 1 1 1 1 |  IRV
bad address        load r4, 4(r3)  | 1 0 1 0 0 1 1 1 |  IRV
correct result     sub r5, r6, r2  | 1 0 1 0 0 1 1 1 |  IRV
miss -> prefetch   load r5, 0(r5)  | 1 0 1 0 0 0 1 1 |  IRV
                                   .
                                   .
                                   .
```

The first instruction in the sequence is a load that misses in the level one dcache. The bit in the IRV corresponding to the destination register of the load (r1) is marked invalid (with a zero) in the IRV. The second instruction sources an invalid register (r1). Its destination register (r3) is subsequently marked invalid. The third instruction is another load. This load cannot properly form its target address, since it sources an invalid register (r3). As a result this load cannot generate a runahead prefetch, and has to mark its destination register (r4) as invalid. 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 yet another load. This time the load can properly form its target address using (r5). If the load hits in the dcache and the sub-block runahead valid bit for the target word in the dcache is marked valid, then it marks its destination register (r5) as valid since it was able to load a valid dcache word into (r5). If the load hit in the dcache and the sub-block runahead valid bit for the target word is invalid, then the processor marks the load destination register (r5) as invalid. If the load misses in the data cache (as the Figure assumes), then it marks its destination register (r5) as invalid and generates a prefetch using the target address that it calculated using the previously valid register (r5).

This process continues until the memory hierarchy is able to service the dcache 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.) Before the processor can leave runahead mode it has to reset all of the IRV and dcache sub-block valid bits to the valid state and performs the 1:1 copy of the NRA-RF contents to the RA-RF registers.

Once the processor has left runahead mode it restarts execution at the PC of the first instruction shown in Figure 1. Since the miss corresponding to the load has already been serviced, it is guaranteed to hit in the dcache. The following add instruction is then able to execute normally. The third instruction may generate a dcache 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 didn't generate a dcache miss, then the fourth instruction (the subtract) can execute normally. Finally, if the prefetch corresponding to the last instruction (the third load) has been serviced then the load will not generate a dcache miss. If the prefetch has not been serviced, then the processor will re-enter runahead mode. Note that since the last instruction already generated a prefetch in the previous runahead episode it is not necessary for the processor to issue a miss request to the memory hierarchy if the instruction generated another dcache miss.

Another runahead example is shown in Figure 2.

Figure 2. Stores during runahead

```
                                   .
                                   .        r r r r r r r r
                                   .        0 1 2 3 4 5 6 7
dcache miss        load r1, 0(r2)  | 1 0 1 1 1 1 1 1 |  IRV
bad dcache word    store r2, 0(r1) | 1 0 1 1 1 1 1 1 |  IRV
is r4 valid?       load r4, 4(r3)  | 1 0 1 1 ? 1 1 1 |  IRV

                                   .
                                   .
                                   .
```

Note that the first instruction in the sequence is the same as that in the previous example. It has generated a runahead-initiating dcache miss, and it marked its destination register as invalid exactly as before. The second instruction is a store that needs to use an invalid register (r1) to calculate its target address. Since it cannot determine its target address, it cannot mark the corresponding word in the dcache as invalid if it is in the cache. As a result, the third and final instruction in the sequence cannot know for certain if the word it attempts to load from the dcache is valid or not since the previous runahead-store was unable to mark its target word as invalid. Since the case where a runahead-load reads invalid data as a result of a runahead-store being unable to mark its target as invalid is relatively rare, the processor should always assume that any word in the level one dcache that it reads that is not marked as invalid, is actually valid. Once an undetectable invalid register is introduced into the RA-RF it can propagate invalid values to other runahead registers.

A third runahead example is shown in Figure 3. This example shows the effect of branches during runahead.

Figure 3. Dependent branches during runahead

```
                                     .
                                     .
                                     .          r r r r r r r r
                                                0 1 2 3 4 5 6 7
dcache miss          load r1, 0(r2)            |1 0 1 1 1 1 1 1|  IRV
branch dependency    blt loop r1               |1 0 1 1 1 1 1 1|  IRV
prefetch if miss?    load r4, 4(r3)            |1 0 1 1 1 1 1 1|  IRV

                                     .
                                     .
                                     .
```

The first instruction is assumed to miss in the dcache as before. Note that the conditional branch is dependent upon the invalid register generated by the load miss. A conservative runahead processor would simply halt runahead execution at the invalid-dependent branch in order to avoid polluting the dcache. A more aggressive processor with good branch prediction might continue to execute instructions past the branch dependency if it had a high degree of confidence in its branch prediction scheme. Note that the dynamic branch predictor can be used to insert a predicted branch outcome into the runahead shift register if the processor is in the ADD state, allowing the processor to continue generating runahead branch outcomes. If the predicted outcome inserted into the shift register is subsequently found to be incorrect when the processor reenters non-runahead mode, then the processor can avoid any additional mispredictions by flushing the shift register.

A fourth runahead example is shown in Figure 4. This example demonstrates how runahead can improve the performance of dynamic branch prediction schemes.

Figure 4. Updating the branch prediction during runahead

```
                                     .
                                     .
                                     .          r r r r r r r r
                                                0 1 2 3 4 5 6 7
dcache miss          load r1, 0(r2)            |1 0 1 1 1 1 1 1|  IRV
cond branch          blt loop r2               |1 0 1 1 1 1 1 1|  IRV
prefetch if miss     load r4, 4(r3)            |1 0 1 1 1 1 1 1|  IRV

                                     .
                                     .
                                     .
```

The first instruction initiates the runahead sequence by generating a dcache miss. Note that the conditional branch is not dependent upon the cache miss. This means that the processor can do one of two things with the branch. First, it can allocate a branch prediction entry and initial outcome for the branch in its dynamic branch predictor, if it does not already have one. Second, if the processor already has an entry for the branch it can update the entry with the outcome of the runahead branch. The processor could also cache the branch target address in a branch target buffer. The updated state of the conventional dynamic branch predictor is used to guide the processor when the branch shift register is empty in non-runahead mode, and while the processor is in runahead mode. If the processor is in the ADD state the actual outcome of the conditional branch is shifted into the branch shift register once it is resolved, and the branch counter is incremented. When the processor leaves runahead mode and re-executes the conditional branch it will already have the actual outcome of the branch in the branch shift register. By shifting the outcome of the branch out of the shift register when the branch instruction is decoded the processor can predict the outcome of the branch with a high degree of accuracy (to the limit of the ability of the processor to detect invalid registers). Note that if the processor executes past branches that are dependent upon invalid registers, then it can introduce potentially incorrect outcomes into the branch shift register.

2 Simulation

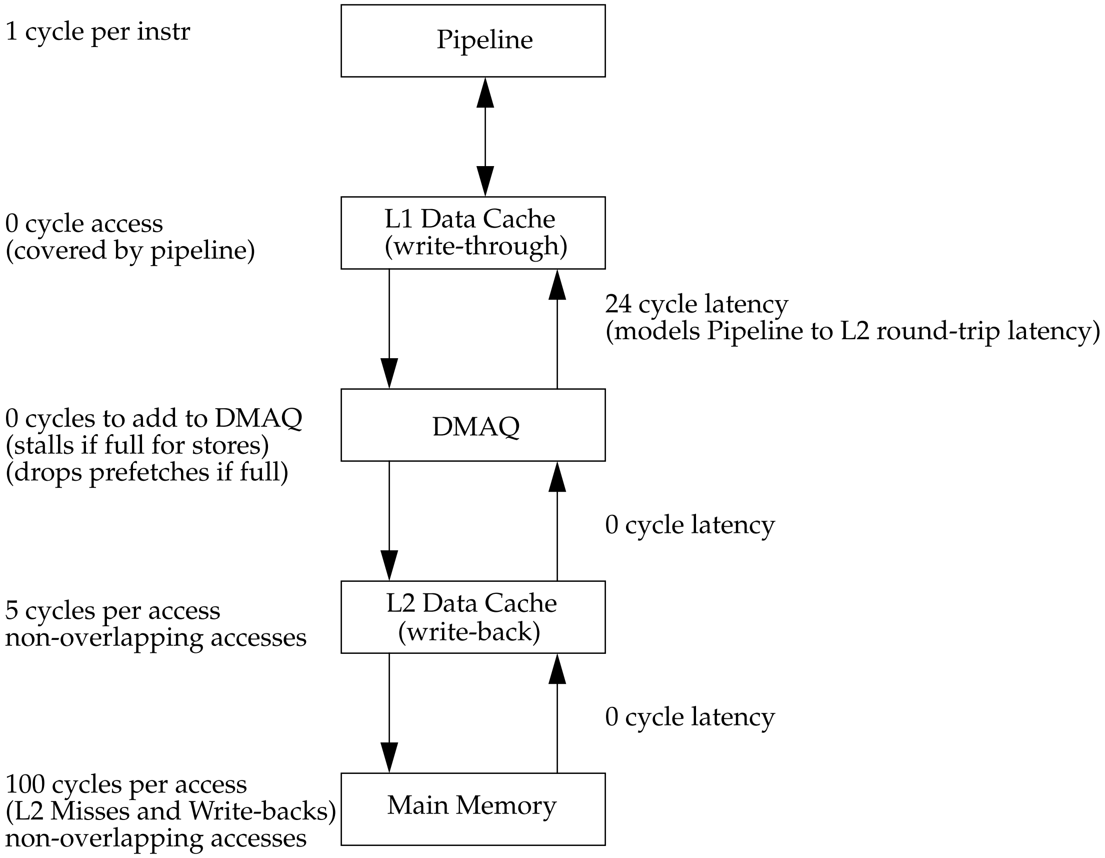
A simulator was written to evaluate how runahead can improve processor performance. The simulator uses the ATOM [3] tool to instrument application binaries. Instruction cache misses are not modeled. It also ignores pipeline effects by assuming that each instruction is fetched, decoded and executed sequentially in a single cycle. Only virtual addresses are used to access the caches and main memory. Also, all required memory pages are assumed to reside in main memory, i.e. page faults cannot occur.

2.1 Description of the simulated processor

The memory hierarchy corresponds closely to that which will likely be used in the PUMA [4] microprocessor. It consists of a non-blocking level one data cache, a non-blocking level two data cache, and a main memory. Both data caches are direct-mapped. The L1 dcache is write-through/write-allocate, while the L2 dcache is write-back/write-allocate. A data memory access queue (DMAQ) is provided between the level one and level two caches, and is used to send miss and prefetch requests, as well as store-throughs from the L1 cache to the L2 cache. Note that the DMAQ also performs the function of the outstanding request list (ORL) described in [5]. The DMAQ cannot coalesce store-throughs or squash prefetches, and all requests in the DMAQ are processed in the order in which they were placed in the DMAQ. The line and transfer sizes of the caches and DMAQ are 32 bytes. The processor stalls for store throughs that occur when the DMAQ is full. Potential prefetches that are generated when the DMAQ is full are dropped. A diagram which shows the data stream hierarchy is shown in Figure 5, and includes a description of the latencies between the different levels of the hierarchy. L2 accesses require a minimum of 24 cycles to access the L2 dcache before they can be removed from the DMAQ. L2 dcache misses require an additional 100 cycles to fetch the missing data from main memory. If dirty data must be written back from the L2 dcache to main memory, then another 100 cycles are required on a L2 dcache miss. Finally, L2 accesses must be spaced five cycles apart, while main memory accesses must be spaced 100 cycles apart.

The branch prediction scheme used in the simulated processor is the two-bit counter method first described in [6], augmented with the runahead branch shift register, state bits, and counters that were described earlier. The two-bit counters are indexed by the low-order bits of the program counter. No tag or valid bits are provided. Since the current version of the simulator cannot model the execution of instructions down wrong-paths, all simulations that allow runahead to continue past dependent branches always stay on the correct path of execution. Therefore the simulation results should be considered best case.

Figure 5. Simulated Memory Hierarchy



2.2 Simulation benchmarks

The benchmarks used in the simulations include the compress, go, and perl benchmarks from the Spec95 suite, as well as version 1.1 of the mpeg-2 decoder written by the MPEG Software Simulation Group. The compress benchmark used the normal test input (bigtest.in). Go used a play level of 10 and a board size of 13. Perl used the scrabbl.in and scrabbl.pl input files. Finally, mpeg_play used a 490KB mpeg2 file as input (hakinen2.mpg). All benchmarks were compiled with gcc using the -O optimization flag. All simulations were allowed to run for the first 500M instructions before they were terminated. The data caches and branch prediction were cold started for all simulations, and no attempt was made to model the effects of context switches. Graphs of the simulation results are shown in the Appendix.

2.3 Data cache simulations

Data cache simulations were run for the mpeg_play, compress, go, and perl benchmarks, and a number of statistics were gathered. The simulations were run with varying L1 dcache sizes and runahead policies. The L1 dcache size was varied from 1 to 8KB, while the runahead policies were: no-runahead, runahead until a dependent branch is fetched, and runahead past any dependent branches until the runahead-initiating miss is serviced. The DMAQ and L2 dcache sizes were fixed at 8 entries and 1MB respectively. All cache line sizes were fixed at 32 bytes.

The first set of simulation results, shown in Figures 6 through 21 show the CPI and data cache miss statistics of the simulated processor for the benchmarks in question. Note that since the simulated processor executes all instructions in a single cycle and has a perfect level one instruction cache, the data memory's contribution to the processor CPI (MCPI) is simply one less than the overall CPI.

The first CPI plot, shown in Figure 6, is for the compress benchmark. Note that compress benefits very little from runahead that stops at dependent branches for all L1 dcache sizes greater than 1KB. In fact, the CPI actually increases slightly over the non-runahead case for an 8KB L1 dcache. When the processor is permitted to run past dependent branches, the CPI improvement is noticeable over the entire range of L1 dcache sizes studied. Note that over most of the range of dcache sizes, the more aggressive runahead scheme results in greater than 25% improvement in MCPI for compress. This can be seen quite clearly in Figure 7. The less aggressive runahead scheme actually hurts processor performance with the 8KB L1 dcache. The L1 dcache miss rates for compress, as well as the miss rate improvements over non-runahead, can be seen in Figures 8 and 9.

Runahead appears to benefit the d-stream performance of mpeg_play more than compress. This can be seen in Figures 10 through 13. Both runahead schemes improve the CPI of the processor, with the more aggressive scheme beating out the scheme that halts runahead at dependent branches. The more aggressive scheme improves MCPI by about 25% over most of the L1 dcache size range, while the less aggressive scheme improves MCPI by about 20%. The L1 dcache miss rates and miss rate improvements over non-runahead, shown in Figures 12 and 13, clearly show the benefits of runahead for the mpeg_play benchmark. The more aggressive prefetching scheme improves the dcache miss rate by over 20%, while the less aggressive scheme improves the miss rate by about 17%.

Finally, both the perl and go benchmarks benefit significantly from runahead. This can be seen in Figures 14 through 21. The MCPI improvement for perl (22-49% for the aggressive scheme, and 15-35% for the less aggressive scheme) tops that of both mpeg_play and compress. The MCPI improvement for go (42-58% and 22-32%)is even better than that for perl. The dcache miss rates for both benchmarks benefit in a similar fashion.

2.4 Branch Prediction Simulations

The simulator used in the previous section was modified such that it could be used to gauge the ability of runahead to improve branch prediction accuracy. The simulated processor was augmented with a dynamic branch prediction unit consisting of a direct mapped array of two-bit counters corresponding to the algorithm first described in [6]. No tag bits were provided for the counters, so there is some degree of aliasing in the predictor array. A 64 bit runahead branch shift register was provided to hold the outcomes of conditional branches encountered while the processor is in runahead mode. The L1 dcache size is fixed at 4KB for all of the branch prediction simulations. The number of two-bit counters used were 64, 128, 256, and 512. Also, all runahead episodes are allowed to continue past dependent branches. Finally, note that all of the branch misprediction statistics presented are for branches encountered while the processor is in non-runahead mode.

As was mentioned in an earlier section, the current simulator is unable to explore wrong-paths while the processor is in runahead mode. This means that all of the predicted outcomes added to the shift register are always correct, even for branches whose outcome couldn't be computed during runahead due to a dependence upon an invalid register. In a real processor, the dynamic branch predictors would be used to "guess" the value that should be shifted into the runahead shift register for runahead branches whose outcome cannot be resolved. Therefore all of the results presented in this section should be viewed as a upper limit to the branch performance gains that may be obtained using the runahead technique. A more advanced simulator that will address these concerns is in development.

Finally, note that jump instructions can be a problem since a register holding a jump target address can become invalid during runahead. The current simulator ignores this possibility by assuming that jump targets can always be properly computed. An actual runahead processor would typically use the predicted target address supplied by a subroutine return prediction stack if a jump instruction sourced an invalid register during runahead. The next version of the simulator will address this issue.

The first set of branch prediction simulation results are show in Figures 22 and 23. These Figures are for the compress benchmark. Three misprediction rates are shown in Figure 22: no runahead, runahead using the shift register where possible, and runahead without a shift register (using only the dynamic predictor array, which is updated during both runahead and non-runahead). The branch misprediction rates for compress do not vary noticeably over the entire x-range. This is a result of the small number (95) of static conditional branches in the compress benchmark [7]. Also, since these simulations stopped after 500M instructions, the branch misprediction rate is slightly higher for the two-bit counters than that reported elsewhere [7]. This is a result of the lower training time per conditional branch in the predictor array. Figure 23 indicates that the processor with a runahead branch shift register is able to reduce the branch misprediction rate by 41% for compress. Eliminating the shift register and relying only on the ability of runahead to train the two-bit counters results in a 20% reduction in the misprediction rate.

The simulation results for mpeg_play are shown in Figures 24 and 25. The effect of the size the dynamic predictor array is apparent here in the decreasing misprediction rate as the number of two-bit counters is increased. This is a result of the relatively large number of static conditional branches in the mpeg_play benchmark (5,598 for mpeg_play from Spec95) [7]. Runahead is able to improve the branch misprediction rate somewhat, as is shown in Figure 25, with an improvement of about 7-11% for the processor with the shift register, and an improvement of about 2-4% for the processor with only the dynamic predictors.

3 Future Work

There are several possible improvements to the baseline runahead scheme, which could reduce hardware cost or improve performance.

3.1 Reducing the cost of the runahead register file

It may be possible for a runahead processor to rely entirely on its forwarding paths to supply runahead instructions with register values that are computed in runahead. This would allow a low-cost runahead processor to eliminate the runahead register file (RA-RF). Another approach would be to provide runahead registers for only a subset of the non-runahead register file. This might be very effective, especially with compiler support. It is also possible to use a very small fully associative cache as a RA-RF.

3.2 Reducing the amount of branch prediction hardware

It may be the case that a runahead processor with a runahead branch shift register and a static branch prediction scheme can achieve acceptable branch prediction performance compared to a non-runahead processor equipped with a dynamic branch prediction unit.

3.3 Provide a runahead store data cache

Some performance is lost since runahead stores cannot modify the level one data cache. Subsequent runahead load instructions cannot access the missing data, which results in unnecessary register invalidations. While the processor can duplicate the architected register file with a reasonable amount of hardware, it is impractical to provide a duplicate copy of the level one data cache.

A multi-way or fully associative runahead store data cache could be used to capture store data in runahead, which could then be used by subsequent loads in the same runahead episode. This runahead cache could probably be quite small yet still deliver a reasonable improvement in performance. Note that this approach is analogous to the victim cache described in [8].

3.4 Resume runahead at a more optimal point in the instruction stream

One of the problems with the baseline runahead scheme is that runahead always starts at the PC of a load or store that produced a data cache miss. If a runahead sequence generates one or more prefetches that haven't been serviced by the time the processor restarts execution at the runahead-initiating load or store, then it might be more productive for the processor to continue runahead at the last instruction that it didn't execute at the end of the previous runahead episode, rather than restarting at the PC of any prefetches that have not yet been serviced. In some situations it may make sense to resume runahead at a point in-between the last non-runahead instruction and the point at which runahead last stopped. The following factors influence the choice of where to restart runahead:

- If the last runahead episode ended with relatively few registers valid, then the processor would probably be able to generate relatively few prefetches by continuing where it left off. A better strategy may be to restart at an intermediate point in the instruction stream.
- The processor might generate useless prefetches if runahead continues down an incorrect runahead execution path due to an incorrect prediction of a runahead dependent branch. Therefore it may be the case that the processor should resume runahead at an earlier instruction for best performance. The correct execution path at the point which it started down the wrong path is a likely place to resume runahead in this case.

3.5 Skip over unnecessary instructions while in runahead mode

Runahead performance might be improved by having the processor skip over runahead instructions that aren't needed to keep the processor on the correct path or to fetch data. A hardware scheme to detect unneeded computational instructions may be impractical to implement. Even if the compiler marked these instructions it would still be difficult to quickly skip the istream over them, and at any rate doing so would still decrease istream performance via inefficient fetch utilization.

However a compiler may be able to efficiently perform this task if the instruction set is augmented. The only hardware support required is the addition of a special unconditional branch instruction to the processor ISA. This runahead branch, or ra-branch, would only be executed by the processor if it was in runahead mode. While the processor is not in runahead mode, ra-branches would be treated as NOPs. These branches would be used to "skip" the processor over instructions that are not needed to compute load/store addresses or branch outcomes while it is in runahead mode, thus minimizing the number of runahead instructions executed per generated prefetch. The compiler would have to use the ra-branch instruction sparingly to avoid adding too many NOPs to the code.

Another approach is to use a variant of the informing memory technique from [9]. The basic idea is to switch from a non-runahead thread to a runahead thread on a cache miss. The runahead thread would be a mirror image of the non-runahead thread, except that computational instructions that aren't needed to fetch the data stream are left out. This eliminates the need for the ra-branch instruction at the cost of (possibly quite significant) code bloat and the resulting decrease in istream performance.

3.6 Use the runahead branch shift register as a history register in a two-level branching scheme

By using the runahead branch shift register as a history register it may be possible to mix past and future outcomes in the predictor table. This might result in improved performance by using the predictor array more efficiently.

3.7 Use runahead to program a modified stream buffer

It may be possible to use runahead instructions to load a modified stream buffer [8] with prefetching information. This prefetching information could be augmented with other information from a correlating data access predictor. The predictor could be used to correlate past, present, and future (runahead) access information in an attempt to predict stream buffer access strides, etc. Note that using a correlating predictor for data prefetching was first examined in [5].

3.8 Fine-Grained Runahead

The basic idea behind runahead is to exploit processor cycles that are usually wasted in the form of stalls. The baseline runahead scheme can only issue (a sequence of) runahead instructions on level one data cache misses, which are relatively rare compared to other stall sources in a simple processor with a very high clock rate. For this reason we refer to the baseline runahead scheme as coarse-grained runahead. If the processor is modified such that it can issue runahead instructions on more common stall events, such as those caused by read-after-write (RAW) dependencies between instructions, it may be possible to significantly improve runahead performance by replacing what are normally useless pipeline bubbles with runahead instructions. W ine-grained runahead. Note that this scheme is conceptually similar to fine-grained multithreading, with the primary difference being that the second (runahead) thread of execution is a thread whose use of the processor helps rather than hinders the cache and branch prediction performance of the primary non-runahead thread of execution.

It is relatively easy to convert the first bubble added to a single-issue pipeline during a RAW stall event into a runahead prefetching instruction. This first pipeline bubble is simply a runahead copy of the instruction that is stalled. Since this instruction has a runahead dependency by definition, it will probably not be able to accomplish any useful runahead work. If the stall condition lasts for more than once cycle, then it is possible for one or more of the non-runahead instructions that may be trapped behind the stalled instruction to issue runahead copies of themselves in lieu of additional bubbles. These fine-grained runahead instructions may be able to generate runahead prefetches and compute branch “trial-run” outcomes. This will require extra hardware in the early stages of the pipeline.

This scheme is similar to out-of-order execution, however it does not require large blocks of logic (such as reorder buffers) that eat up precious chip real estate and can impact the clock cycle time. An in-order issue superscalar processor could potentially issue several fine-grained runahead instructions per RAW stall.

The end result of all of this is that a processor that incorporates both fine and coarse-grained runahead can attempt to productively exploit many processor cycles that would otherwise be wasted. There are many details and complications to this fine-grained runahead scheme which we will examine in much greater detail in the future.

4 Conclusions

In this paper we first discussed a new method of improving several aspects of processor performance, and subsequently presented some early simulation results. Although the simulation results that have been presented are encouraging, they are best viewed as preliminary since the simulator that was used could not explore wrong paths of execution or model instruction cache and pipeline effects. Similarly, simulation results from only four benchmarks were available at the time that this paper was written. Several possible extensions to the baseline runahead scheme were discussed and will be investigated in greater detail in the future.

5 References

[1] Personal communication with Peter L. Bird, June 1996.

[2] James E. Pierce, "Cache Behavior in the Presence of Speculative Execution - The Benefits of Misprediction," Ph.D. Thesis, The University of Michigan, 1995.

[3] 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.

[4] James Dundas and Todd Basso, "An Overview of the PUMA FXU," The University of Michigan, March 26, 1996.

[5] Tien-Fu Chen and Jean-Loup Baer, "Effective Hardware-Based Data Prefetching for High-Performance Processors," IEEE Transactions on Computers, vol. 44, num. 5, May 1995.

[6] Alan Jay Smith and Johnny K. F. Lee, "Branch Prediction Strategies and Branch Target Buffer Design," IEEE Computer, vol. 17, no. 1, January 1984.

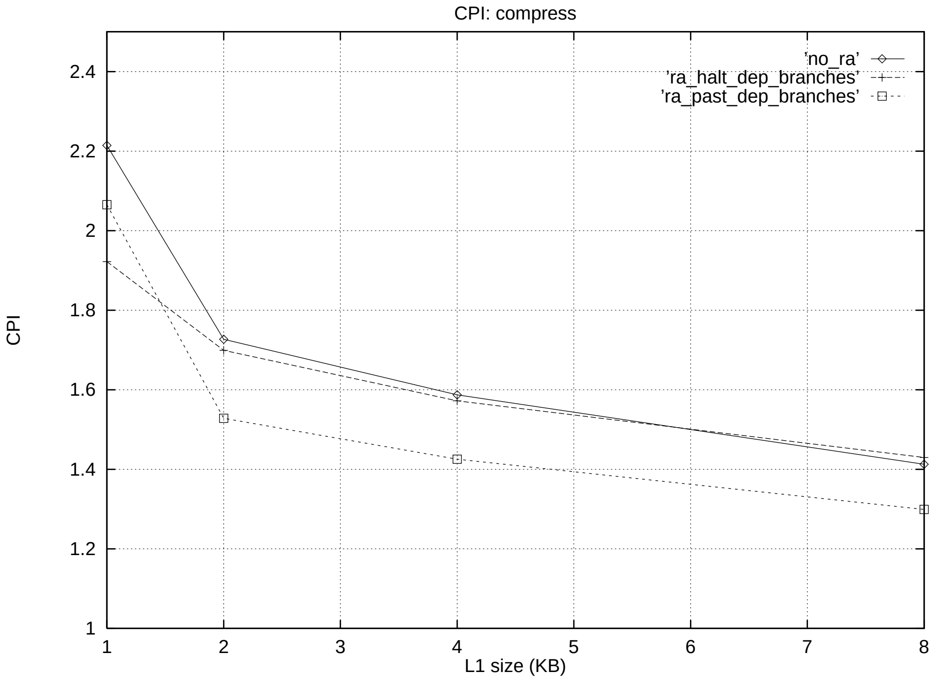
[7] Personal communication with Chih-Chieh Lee, July 1996.

[8] Norman Jouppi, "Improving Direct-Mapped Cache Performance by the addition of a small Fully-Associative Cache and Prefetch Buffers," Proceedings of the 17th Annual International Symposium on Computer Architecture, 1990.

[9] Mark Horowitz, Margaret Martonosi, Todd Mowry, and Michael Smith, "Informing Loads: Enabling Software to Observe and React to Memory Behavior," Stanford University CSL Technical Report CSL-TR-95-673, July 1995.

6 Appendix

Figure 6. CPI for compress

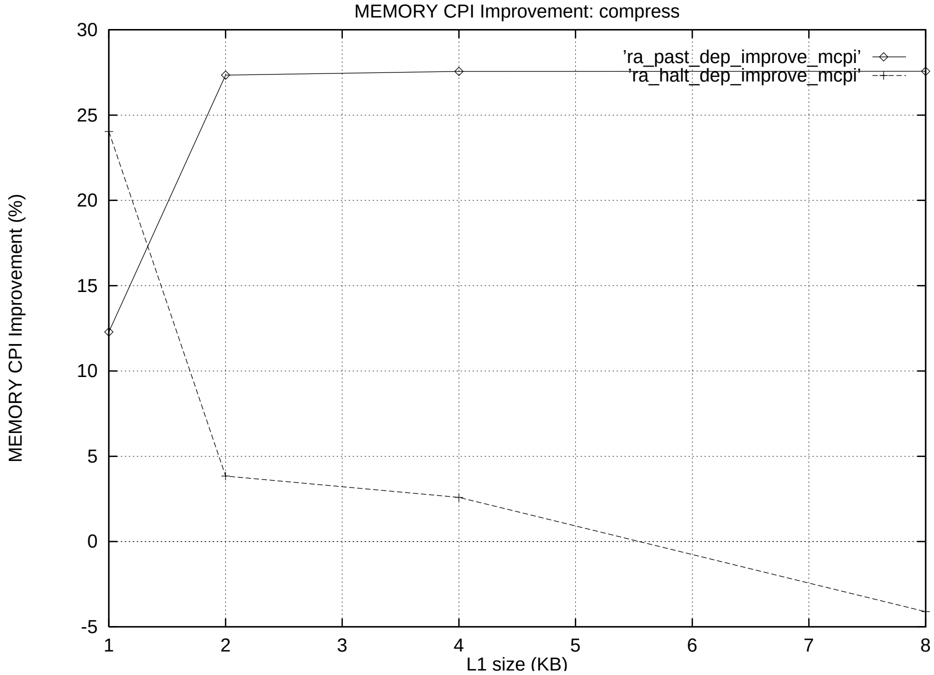


no_ra: Non-runahead processor that stalls on L1 data cache misses.

ra_halt_dep_branches: Runahead processor that halts runahead at branches dependent upon invalid registers.

ra_past_dep_branches: Runahead processor that continues runahead past dependent branches (predicted with 100% accuracy).

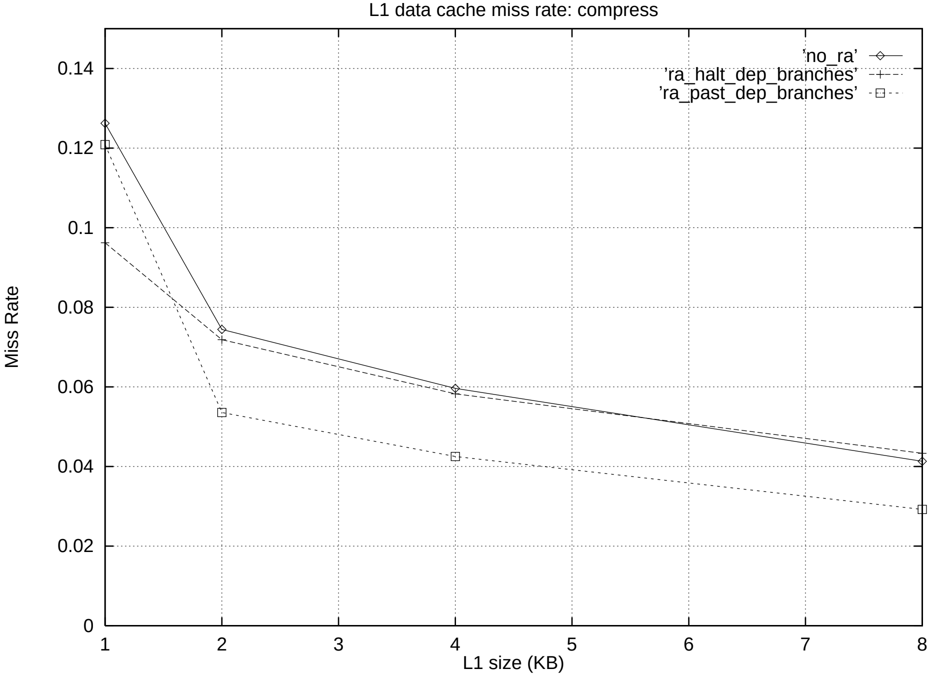
Figure 7. MCPI improvement for compress



ra_past_dep_improve_mcpi: Runahead processor that continues runahead past dependent branches (predicted with 100% accuracy).

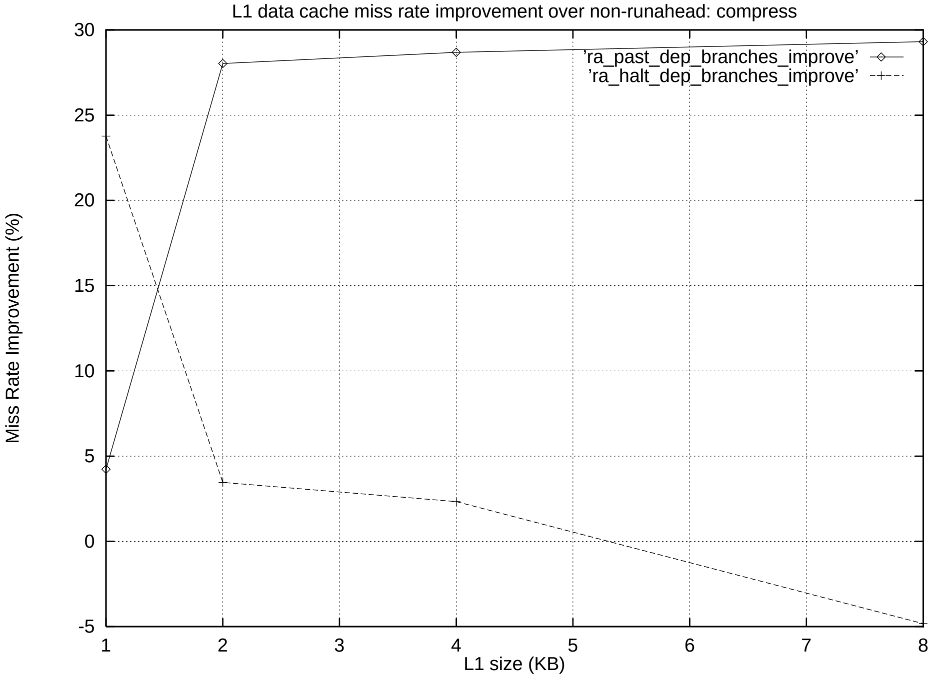
ra_halt_dep_improve_mcpi: Runahead processor that halts runahead at branches dependent upon invalid registers.

Figure 8. L1 data cache miss rates for compress



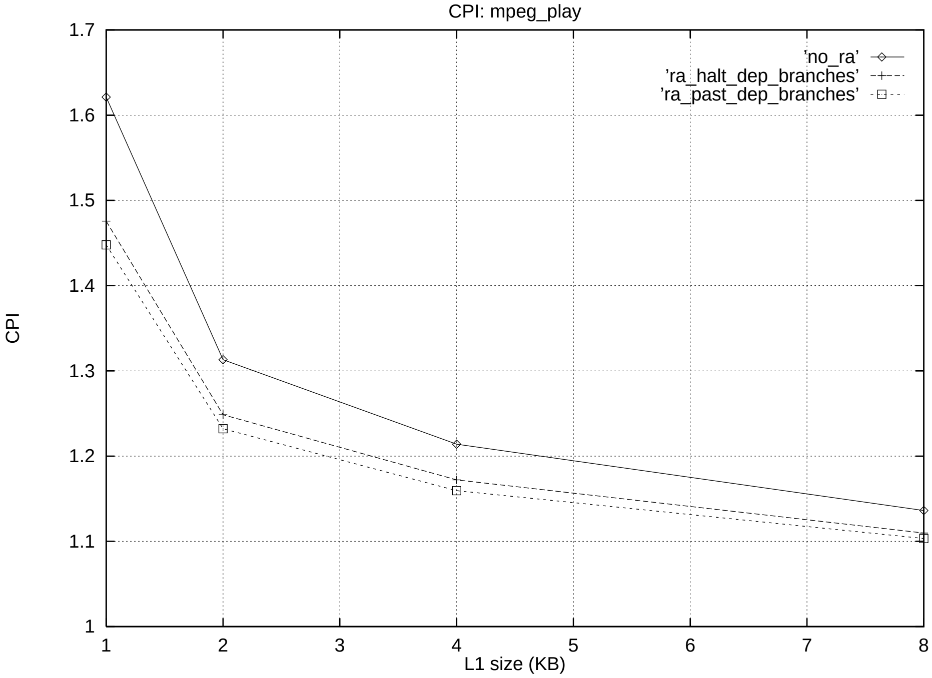
no_ra:	Non-runahead processor that stalls on L1 data cache misses.
ra_halt_dep_branches:	Runahead processor that halts runahead at branches dependent upon invalid registers.
ra_past_dep_branches:	Runahead processor that continues runahead past dependent branches (predicted with 100% accuracy).

Figure 9. L1 data cache miss rate improvement for compress



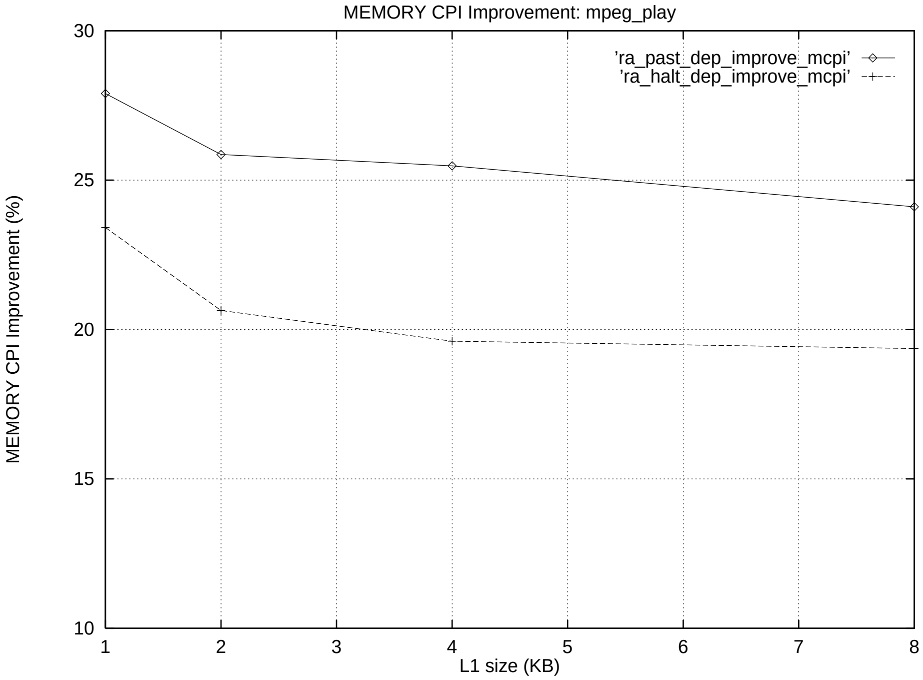
ra_past_dep_branches_improve:	Runahead processor that continues runahead past dependent branches (predicted with 100% accuracy).
ra_halt_dep_branches_improve:	Runahead processor that halts runahead at branches dependent upon invalid registers.

Figure 10. CPI for mpeg_play



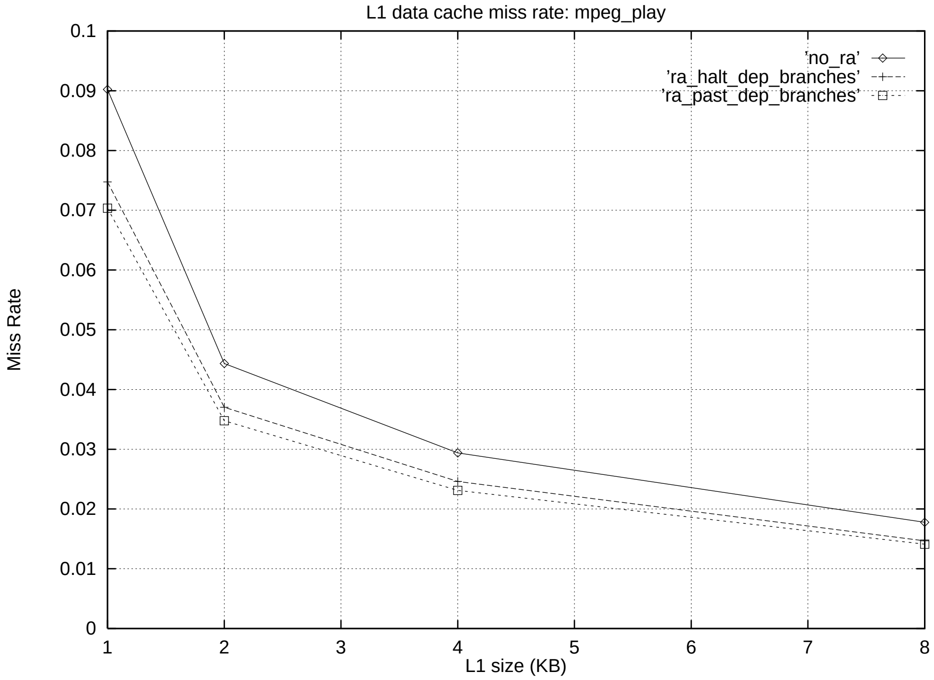
no_ra:	Non-runahead processor that stalls on L1 data cache misses.
ra_halt_dep_branches:	Runahead processor that halts runahead at branches dependent upon invalid registers.
ra_past_dep_branches:	Runahead processor that continues runahead past dependent branches (predicted with 100% accuracy).

Figure 11. MCPI improvement for mpeg_play



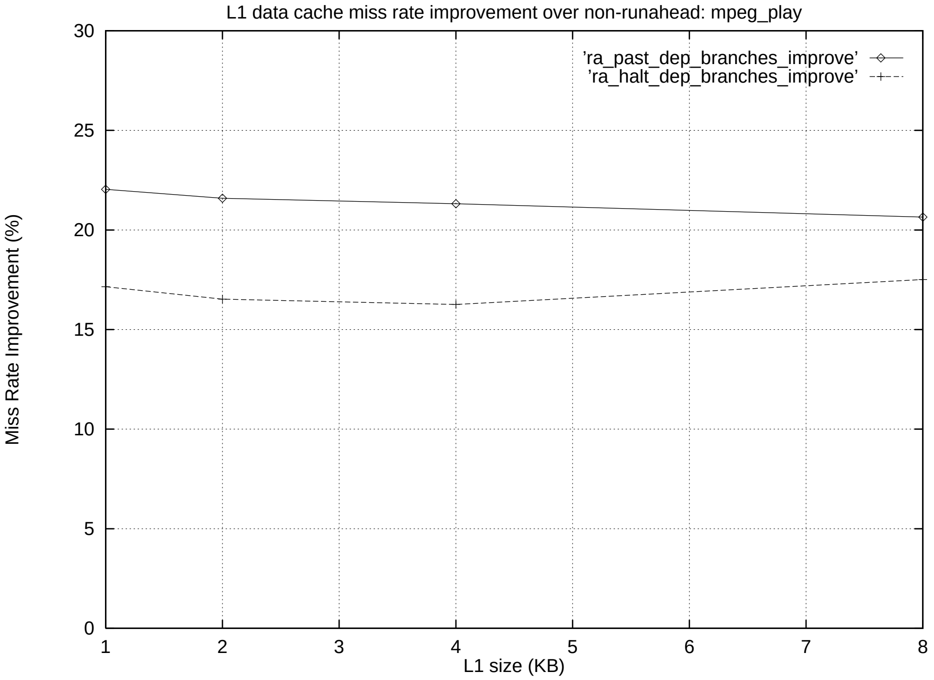
ra_past_dep_improve_mcpi:	Runahead processor that continues runahead past dependent branches (predicted with 100% accuracy).
ra_halt_dep_impro e_mcpi:	Runahead processor that halts runahead at branches dependent upon invalid registers.

Figure 12. L1 dcache miss rate for mpeg_play



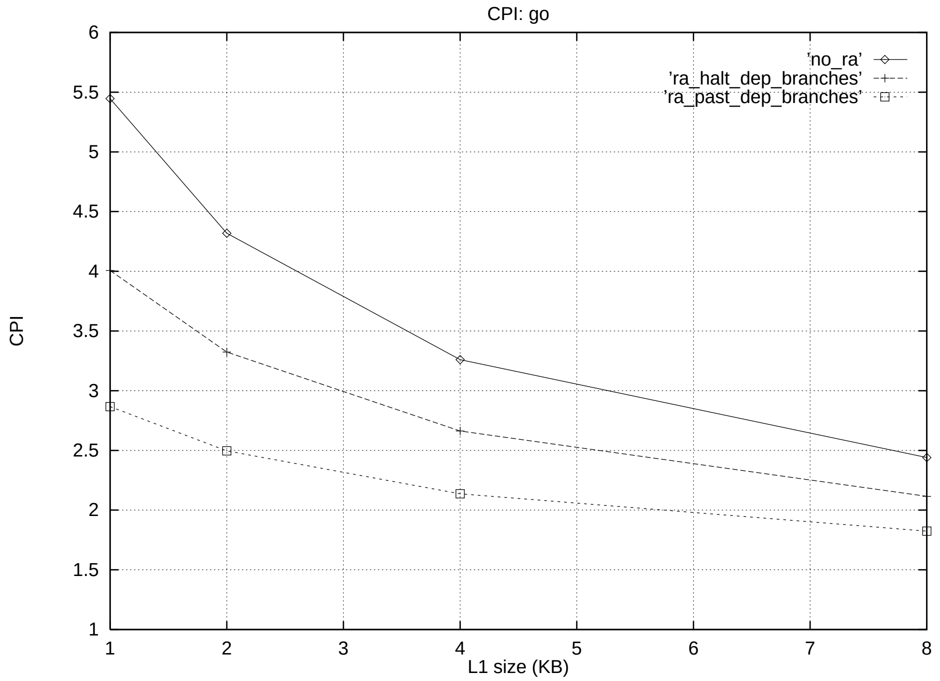
no_ra:	Non-runahead processor that stalls on L1 data cache misses.
ra_halt_dep_branches:	Runahead processor that halts runahead at branches dependent upon invalid registers.
ra_past_dep_branches:	Runahead processor that continues runahead past dependent branches (predicted with 100% accuracy).

Figure 13. L1 dcache miss rate improvement for mpeg_play



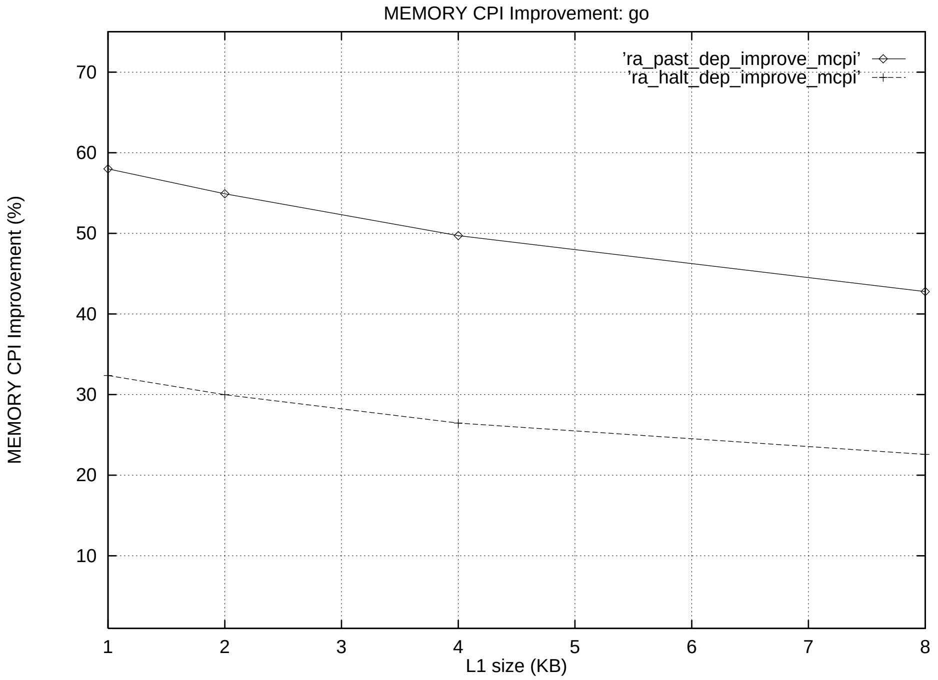
ra_past_dep_branches_improve:	Runahead processor that continues runahead past dependent branches (predicted with 100% accuracy).
ra_halt_dep_branches_improve:	Runahead processor that halts runahead at branches dependent upon invalid registers.

Figure 14. CPI for go



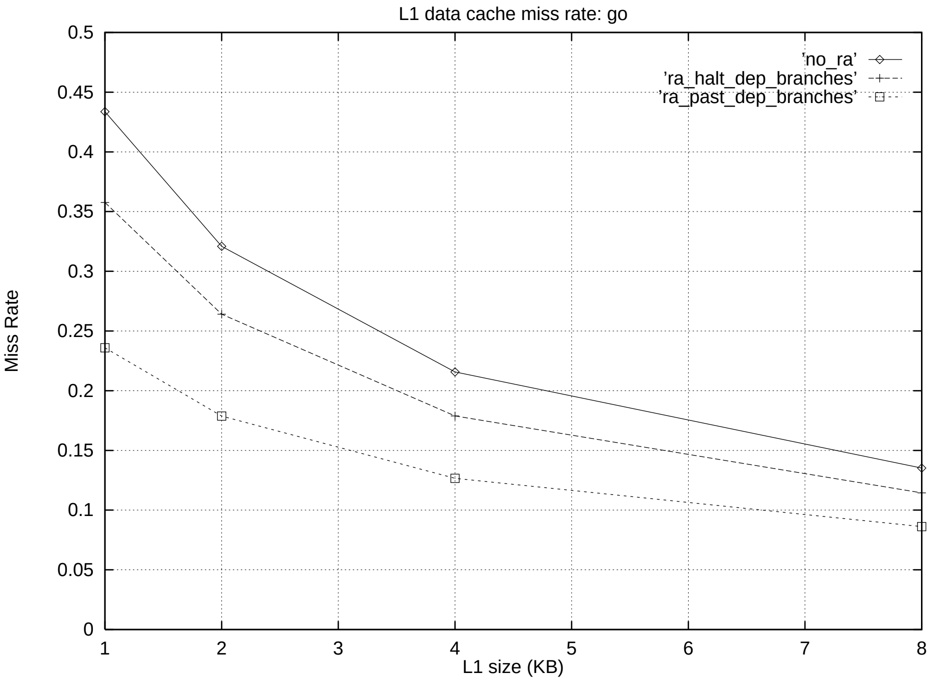
no_ra:	Non-runahead processor that stalls on L1 data cache misses.
ra_halt_dep_branches:	Runahead processor that halts runahead at branches dependent upon invalid registers.
ra_past_dep_branches:	Runahead processor that continues runahead past dependent branches (predicted with 100% accuracy).

Figure 15. MCPI improvement for go



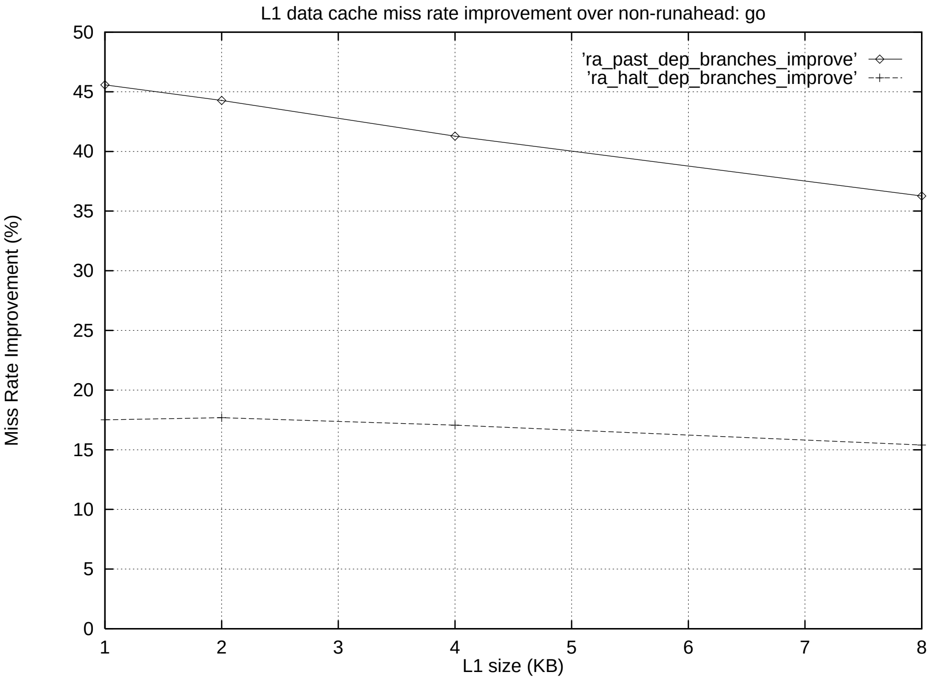
ra_past_dep_improve_mcpi:	Runahead processor that continues runahead past dependent branches (predicted with 100% accuracy).
ra_halt_dep_improve_mcpi:	Runahead processor that halts runahead at branches dependent upon invalid registers.

Figure 16. L1 dcache miss rate for go



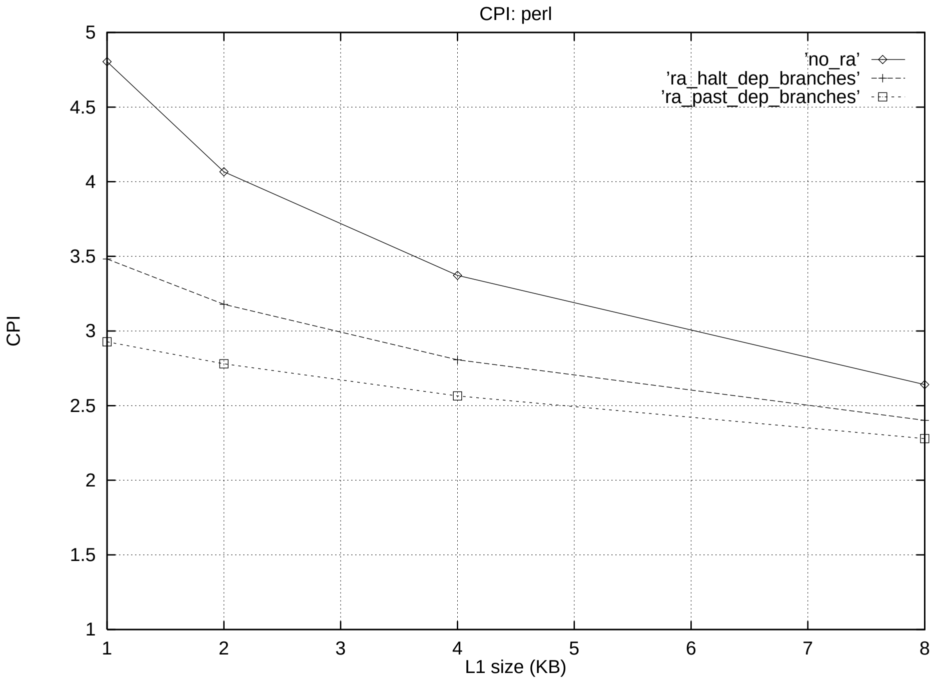
no_ra:	Non-runahead processor that stalls on L1 data cache misses.
ra_halt_dep_branches:	Runahead processor that halts runahead at branches dependent upon invalid registers.
ra_past_dep_branches:	Runahead processor that continues runahead past dependent branches (predicted with 100% accuracy).

Figure 17. L1 dcache miss rate improvement for go



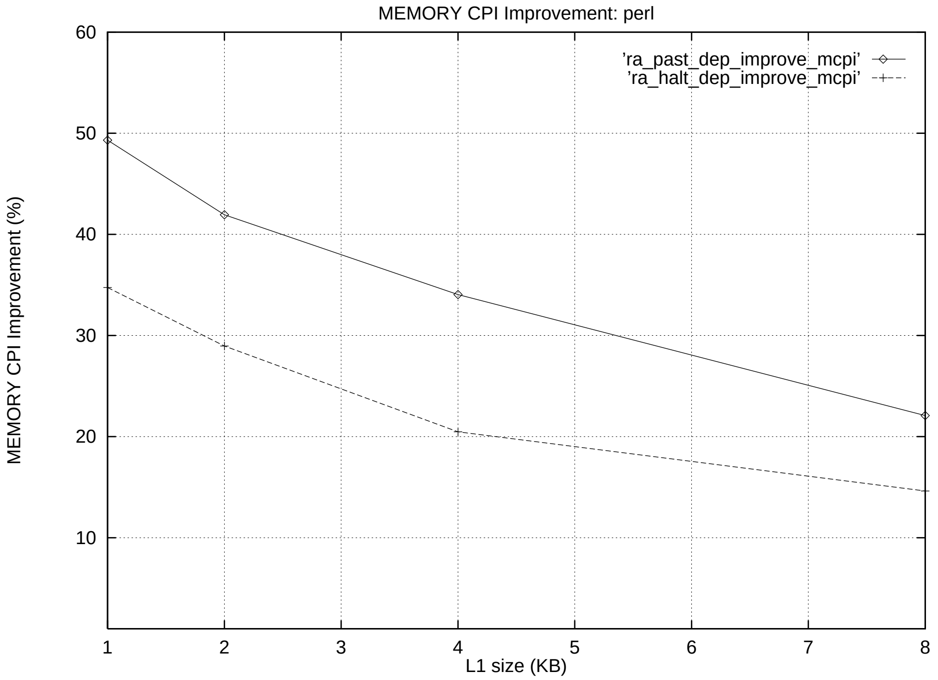
ra_past_dep_branches_improve:	Runahead processor that continues runahead past dependent branches (predicted with 100% accuracy).
ra_halt_dep_branches_improve:	Runahead processor that halts runahead at branches dependent upon invalid registers.

Figure 18. CPI for perl



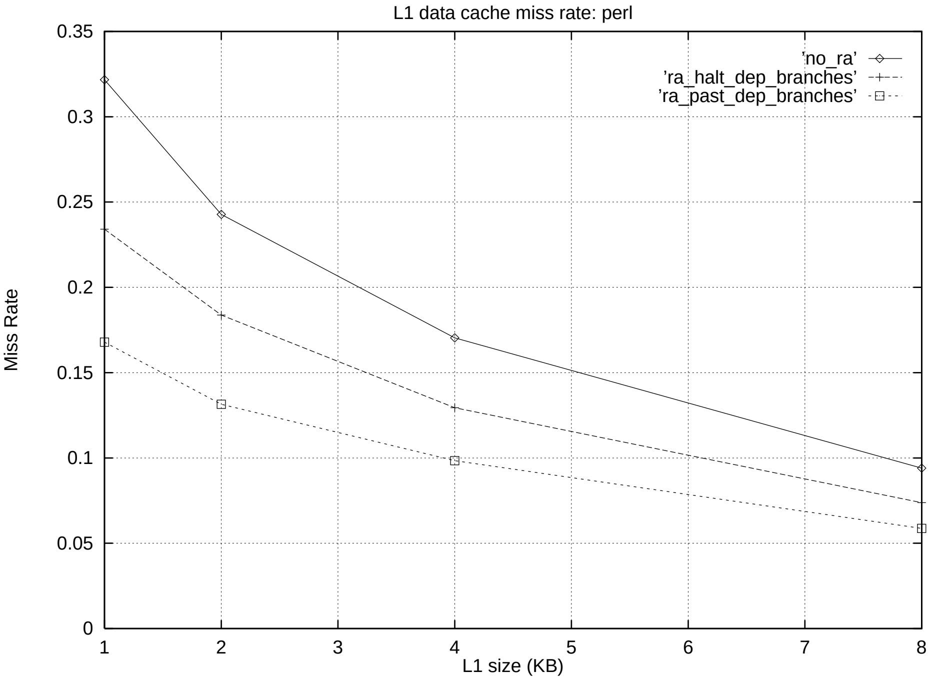
no_ra:	Non-runahead processor that stalls on L1 data cache misses.
ra_halt_dep_branches:	Runahead processor that halts runahead at branches dependent upon invalid registers.
ra_past_dep_branches:	Runahead processor that continues runahead past dependent branches (predicted with 100% accuracy).

Figure 19. MCPI improvement for perl



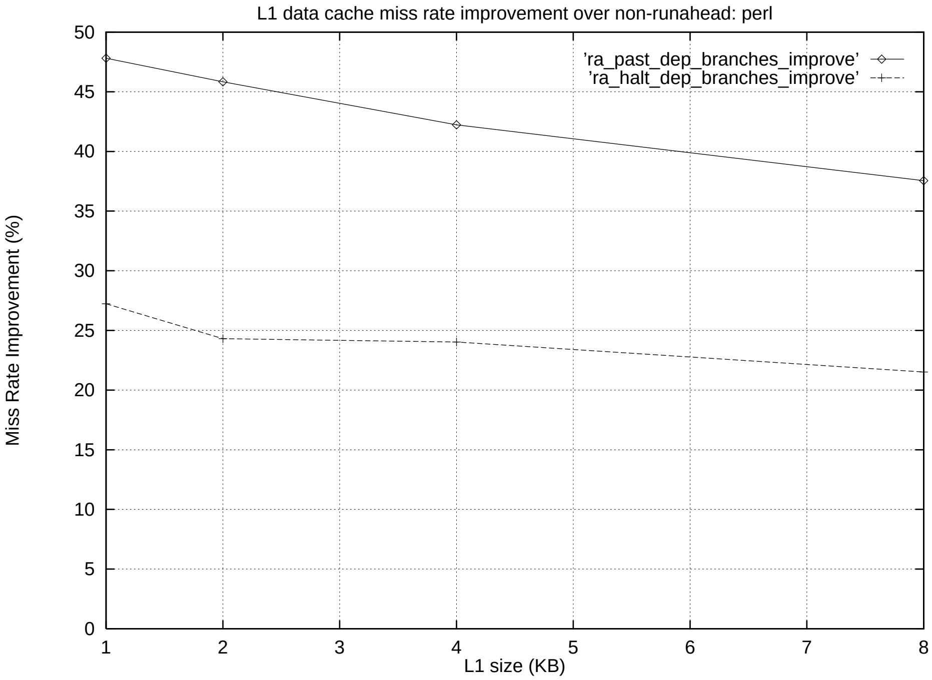
ra_past_dep_improve_mcpi:	Runahead processor that continues runahead past dependent branches (predicted with 100% accuracy).
ra_halt_dep_improve_mcpi:	Runahead processor that halts runahead at branches dependent upon invalid registers.

Figure 20.   L1 dcache miss rate for perl



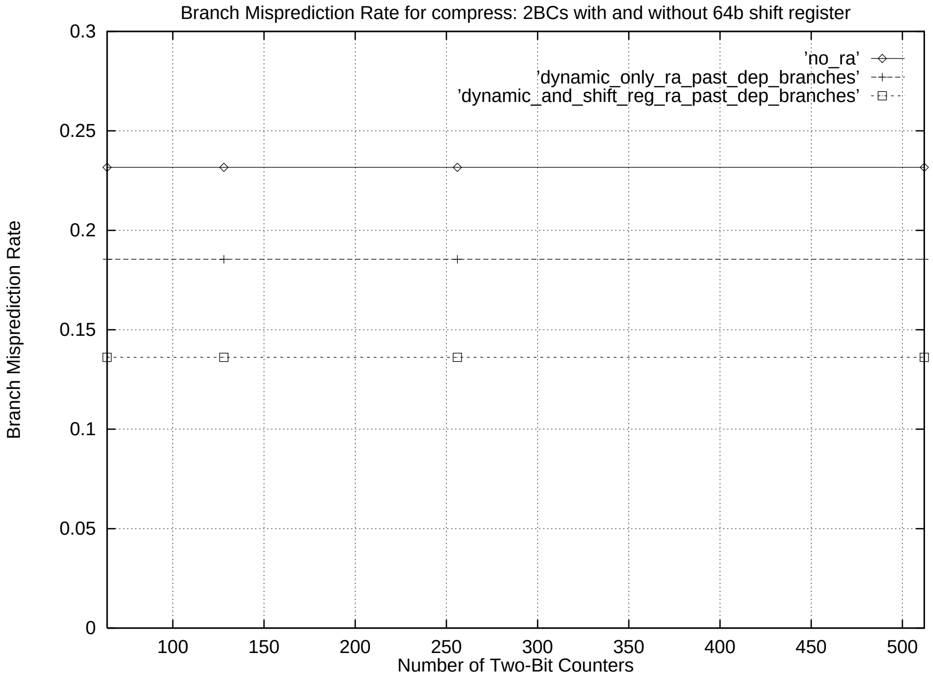
no_ra:	Non-runahead processor that stalls on L1 data cache misses.
ra_halt_dep_branches:	Runahead processor that halts runahead at branches dependent upon invalid registers.
ra_past_dep_branches:	Runahead processor that continues runahead past dependent branches (predicted with 100% accuracy).

Figure 21.   L1 dcache miss rate improvement for perl



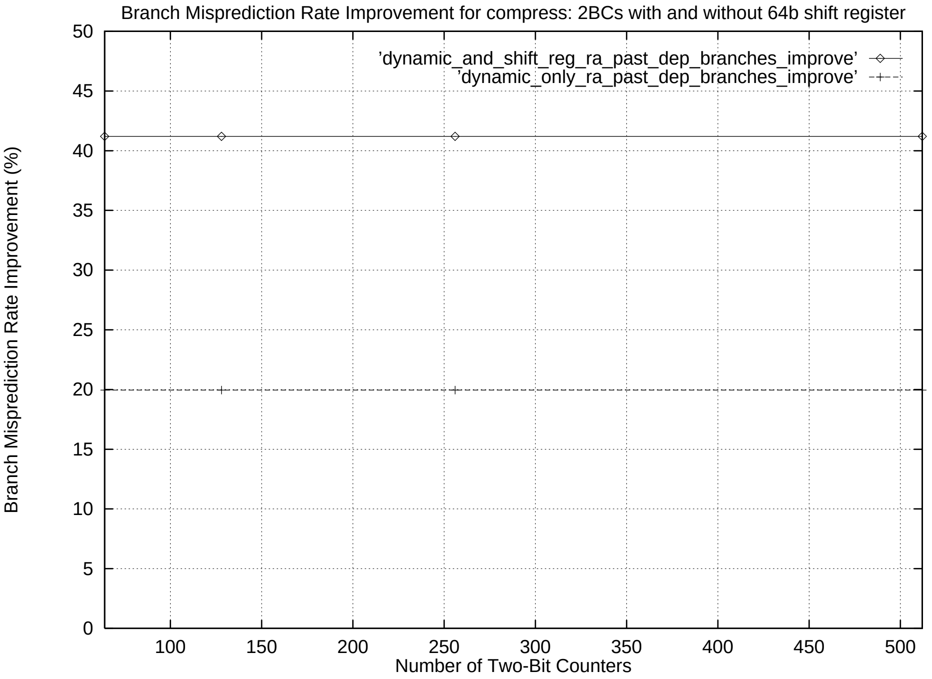
ra_past_dep_branches_improve:	Runahead processor that continues runahead past dependent branches (predicted with 100% accuracy).
ra_halt_dep_branches_improve:	Runahead processor that halts runahead at branches dependent upon invalid registers.

Figure 22. Branch misprediction rate for compress with and without register and runahead



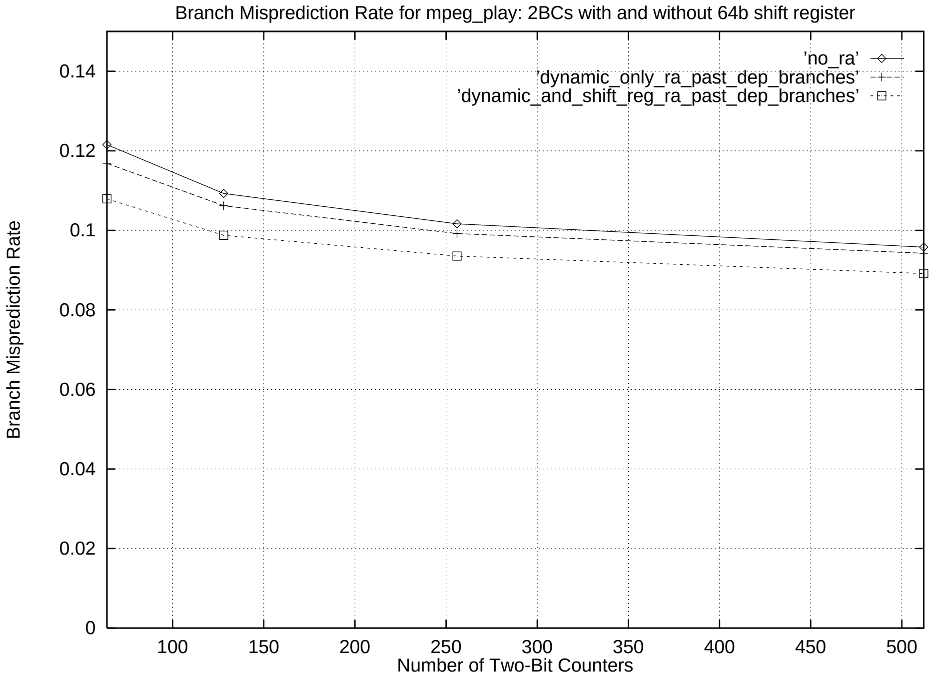
no_ra:	Non-runahead processor that stalls on L1 data cache misses.
dynamic_only_ra_past_dep_branches:	Runahead processor that only has two-bit counters for prediction.
dynamic_and_shift_reg_ra_past_dep_branches:	Runahead processor that uses two-bit counters and a shift register for prediction.

Figure 23. Branch misprediction rate improvement for compress with and without register



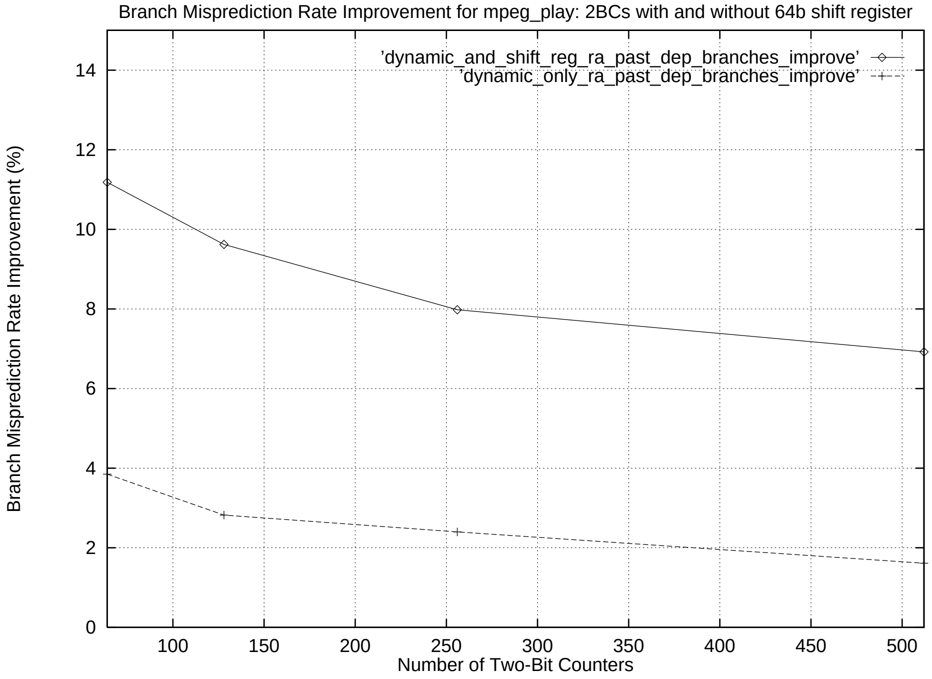
dynamic_and_shift_reg_ra_past_dep_branches_improve:	Runahead processor that uses both the shift register and the two-bit counters to predict non-runahead branches (runahead branches are predicted perfectly).
dynamic_only_ra_past_dep_branches_improve:	Runahead processor that does not have a shift register. Runahead is only used to train the two-bit counters (runahead branches are predicted perfectly).

Figure 24. Branch misprediction rate for mpeg_play with and without register and runahead



no_ra:	Non-runahead processor that stalls on L1 data cache misses.
dynamic_only_ra_past_dep_branches:	Runahead processor that only has two-bit counters for prediction.
dynamic_and_shift_reg_ra_past_dep_branches:	Runahead processor that uses two-bit counters and a shift register for prediction.

Figure 25. Branch misprediction rate improvement for mpeg_play with and without register



dynamic_and_shift_reg_ra_past_dep_branches_improve:	Runahead processor that uses both the shift register and the two-bit counters to predict non-runahead branches (runahead branches are predicted perfectly).
dynamic_only_ra_past_dep_branches_improve:	Runahead processor that does not have a shift register. Runahead is only used to train the two-bit counters (runahead branches are predicted perfectly).

Using Stall Cycles to Improve Microprocessor Performance

James D. Dundas and Trevor N. Mudge

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122

September 1996

email: {dundas, tnm}@eecs.umich.edu