A Compiler-Assisted Data Prefetch Controller

Technical Report No: ARCTiC-99-05 May 1999

Steve VanderWiel
David J. Lilja

A Compiler-Assisted Data Prefetch Controller

Steven P. VanderWiel
svw@ece.umn.edu

David J. Lilja
lilja@ece.umn.edu

Department of Electrical and Computer Engineering
University of Minnesota
Minneapolis, MN

Abstract

Data-intensive applications often exhibit memory referencing patterns with little data reuse, resulting in poor cache utilization and run-times that can be dominated by memory delays. Data prefetching has been proposed as a means of hiding the memory access latencies of data referencing patterns that defeat caching strategies. Prefetching techniques that either use special cache logic to issue prefetches or that rely on the processor to issue prefetch requests typically involve some compromise between accuracy and instruction overhead. A data prefetch controller (DPC) is proposed that combines low instruction overhead with the flexibility and accuracy of a compiler-directed prefetch mechanism. At run-time, the processor and prefetch controller each execute separate, but cooperating, instruction streams. Simulations in which both programs are generated from a single application source file using a commercial compiler show that the prefetch controller can significantly improve the cache utilization and execution time of several SPECfp95 benchmarks. Performance comparisons also indicate that the prefetch controller tends to outperform software prefetching techniques and prefetching using a hardware reference prediction table.

1. Introduction

Cache memories have proven to be very effective in narrowing the performance gap between processors and main memory by reducing the average memory access penalty for programs with high reference locality. Because such programs are relatively common, systems have come to rely on complex cache hierarchies to prevent memory bottlenecks. However, data-intensive applications often exhibit memory referencing patterns with little temporal locality, resulting in poor cache utilization and run-times that are increasingly dominated by memory delays [32].

Rather than waiting for a cache miss to trigger a memory fetch, *data prefetching* techniques anticipate cache misses and issue fetches to the memory system in advance of the actual memory reference. To provide an overlap between processing and memory accesses, computation continues while the prefetched block is brought into the cache.

Conventional hardware prefetching techniques, such as multi-word cache blocks [21] and *one-block lookahead* [2,25] techniques, are limited to prefetching unit-stride, sequential access streams. More sophisticated hardware schemes have been proposed [3,7, 8] that enable prefetching for reference streams with arbitrary constant strides. Most notably, a *reference prediction table* (RPT) [3] monitors the processor's reference stream to detect constant stride array references originating from looping structures. The RPT then automatically generates prefetch requests for the given reference stream. Because these hardware prefetch mechanisms must speculate on future access patterns based only on past references, mispredictions are frequent and some prefetch opportunities are lost.

To take advantage of compile-time program information and add flexibility to the scheduling of prefetch requests, *software prefetching* techniques have been proposed [19,25]. Most contemporary microprocessor instruction sets include one or more fetch instructions that are used to specify the address of a data word to be brought into the cache [1,22,33]. The cache responds to the fetch in a manner similar to an ordinary load instruction with the exception that the referenced word is not forwarded to the processor.

Compiler techniques have been developed to insert fetch instructions into code segments that the compiler predicts will benefit from prefetching, such as looping structures that iterate over large arrays [1,13]. While software prefetching has been shown to be effective in hiding memory latency [22], the use of explicit fetch instructions introduces significant instruction overhead. In addition, the need to limit this overhead often causes the compiler to insert fetch instructions that result in *redundant prefetches* for data that is already cached or *unnecessary prefetches* that bring data into the cache that is never referenced by the processor.

Although compiler-controlled prefetching is able to schedule fetch instructions for arbitrary reference patterns, much of the flexibility and accuracy of this approach is lost in the need to limit the associated overhead. To free the processor from this overhead, a compiler-assisted *data prefetch controller* (DPC) is proposed that retains the benefits of using software to generate prefetch requests while off-loading the task of dispatching fetches from the processor. The following section describes this approach in more detail. The simulation and compiler environments used to evaluate the proposed scheme are then presented followed by a discussion of the simulation results and related research. Our conclusions and future work are presented in the final section.

2. The Data Prefetch Controller

The proposed prefetch controller operates in tandem with the processor by issuing prefetch requests to the memory system on behalf of the processor. The DPC coordinates the dispatch of prefetch requests with the processor's current data usage pattern by snooping the processor's address bus. The prefetch controller is then able to predict upcoming processor data requests based on current data references by executing a simple compiler-generated program that allows the DPC to regenerate portions of the processor's referencing pattern. This *prefetch program* is derived from the same application source code files used to create the program executed by the processor. A more detailed description of the prefetch controller and the prefetch program used to drive it is given in the following sections.

2.1 DPC Organization

Because the prefetch controller is partially controlled by events within the processor, it is convenient to place the DPC on the processor die or on a separate die within the same package as the processor. The DPC is capable of executing a simple program stored in its own cache using an instruction set consisting only of integer instructions used to calculate addresses and issue prefetch requests. However, the DPC always remains under the control of the processor.

The processor initiates prefetch controller operations by writing data and control words to a set of memory-mapped registers associated with the DPC. The memory-mapped data registers are used to pass run-time values to the DPC that are necessary to initiate address calculations for a given prefetch stream. When a word is written to the control register the DPC examines the word to determine which prefetch stream is to be initiated. The prefetch controller then jumps to the appropriate section of code and begins issuing prefetches. At this point, the processor resumes its execution of the program.

After a prefetch stream has been initiated, the DPC issues prefetch requests on behalf of the processor to a shared second-level cache. The prefetched blocks are returned to both the first and second-level data caches. Prefetches are issued to the second-level cache to prevent the DPC from contending with the processor for access to the more heavily utilized L1 cache.

Note that if the DPC were to proceed to prefetch all the data needed for a given reference stream, it would likely outpace the processor and pollute the cache with data that was not yet needed. To prevent such an overrun, some mechanism must be provided to throttle the DPC and keep it synchronized with the processor. The method adopted here establishes a producer-consumer relationship between the DPC and the processor. The unit of exchange in this relationship is called a *trigger block*. Trigger blocks are prefetched cache blocks that cause a signal to be sent to the DPC when they have been referenced by the

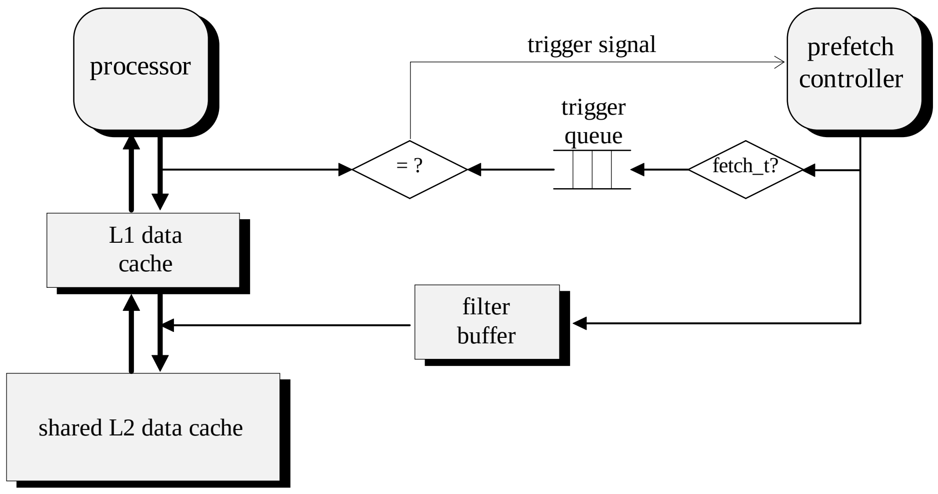


Figure 1. The hardware organization of the data prefetch controller.

processor to indicate that the block has been "consumed." When this trigger signal is received, the DPC is free to issue new prefetch requests.

Although it may be possible to have all prefetched blocks serve as trigger blocks, doing so would result in an unnecessarily large number of trigger signals. An alternate approach is to associate a single trigger block with several other prefetched blocks. When this block is referenced by the processor, it is assumed that all the associated blocks have also been referenced. For loop-based prefetching, it is convenient to have all the prefetches issued for a given loop iteration be associated with a single trigger block. When this block is referenced, the blocks for the next iteration may be prefetched, and so forth. More will be said concerning the selection of trigger blocks in the next section, but for present purposes, it is sufficient to know that the compiler uses a special variant of the fetch instruction, `fetch_t`, to indicate that a cache block is to serve as a trigger block.

To detect when a trigger block has been referenced, a *trigger queue* is used to hold the block addresses of the trigger blocks, as shown in Figure 1. As prefetch addresses are issued from the DPC, those tagged as trigger block addresses are also placed in the trigger queue. When the address at the head of the queue matches the current data reference from the processor, a trigger signal is sent to the DPC and the address at the head of the queue is removed. A new trigger address is then placed at the tail of the queue as new prefetches are issued from the DPC.

Note that the depth of the trigger queue corresponds to the *prefetch distance* [12], δ, maintained by the DPC. That is, when the processor accesses the trigger block issued for iteration i, the resulting trigger signal causes the DPC to issue prefetches for iteration $i + \delta$, where δ is equal to the trigger queue depth. A method of determining an appropriate queue depth for the prefetch controller is presented in Section 4.

A missed trigger block can result in prefetching being disabled for the current prefetch stream. This can occur if the appropriate trigger block is not at the head of the queue when the processor makes a reference to that block. To prevent such an occurrence, the DPC will stall the processor if it is unable to replace the head of the trigger queue with another trigger address immediately after the current queue head has been removed. While this capability is necessary to insure correct operation, the simulation experiments described in Section 3.1.2 found no cases for which it was necessary to stall the processor for this purpose.

The DPC also provides hardware support for detecting and discarding redundant prefetches. Avoiding redundant prefetches in pure software prefetching typically requires the compiler to perform a series of loop transformations such as loop unrolling and loop splitting [1,13]. Although similar techniques may be applied to the code generated for the prefetch program, such compiler analysis is necessarily limited to eliminating only a subset of the redundant prefetches that are actually issued at run-time. In addition, these loop transformations would significantly increase the prefetch program size.

Rather than attempting to avoid redundant prefetches in software, a *filter buffer* is placed between the DPC and the second-level cache to remove redundancies from the prefetch stream. This device is similar to a fully associative address cache that retains the *n* most recently issued prefetch addresses, where *n* is number of entries in the buffer. If a prefetch address is issued by the DPC and found to reside in the filter buffer, the prefetch request is discarded. Otherwise, the request is forwarded to the cache and the address is recorded in the buffer. Such a device is useful for detecting instances of *spatial reuse* [25] which arise when successive executions of a fetch instruction result in prefetch addresses that map to the same cache block. Without the filter buffer, the second prefetch request would be redundantly issued to the second-level cache, potentially stalling the processor unnecessarily due to contention for the cache. The filter buffer can also detect other forms of reuse such as *temporal reuse,* in which a fetch instruction issues requests to the same address during different iterations, and *group reuse*, in which different fetch instructions issue prefetch requests that map to the same cache block.

These reuses result in redundant prefetches only if the cache block retrieved by an earlier prefetch still resides in the processor cache hierarchy when the next prefetch is issued. This is likely to be the case if the two prefetches are issued closely in time but may not be true if they are separated by a greater time interval. The size of the prefetch buffer is therefore kept relatively small to insure that prefetches are discarded only if the same block address has been requested relatively recently. For the benchmark programs used in this study, a prefetch buffer size of 256 elements was found to be sufficient to reduce the number of redundant prefetches to less than 5% of all prefetches issued.

2.2 DPC Programs

As described in the previous section, the processor and prefetch controller each execute separate, but cooperating, programs. Both executables are generated from a single set of application source code files. This common origin enables the prefetch program to be aware of the memory addresses of the main program's global variables and to predict the main program's access patterns. To determine what data should be prefetched by the DPC, the compiler performs *locality analysis* [1,13,31] on the application program loops to find reference streams that are likely to result in cache misses at run-time. Once these reference streams are isolated, the compiler produces code that will allow the prefetch controller to generate the same reference pattern, given a unique *loop index* that specifies which stream is to be generated. In addition, code is added to the main program to communicate data that may be necessary for the DPC to generate the desired prefetch stream, such as array bounds or the starting addresses of dynamically allocated arrays.

Figure 2 illustrates this process by presenting three versions of a vector-matrix computation. The original program is given in Figure 2a. Figure 2b shows how the original program is altered to work with the DPC program given in Figure 2c. Note that the program in Figure 2b is identical to the original program except for the addition of the two assignment statements to the `dpc` structure. These assignments represent stores to the memory-mapped registers of the DPC. The first assignment writes the value of `N` to one of the DPC memory-mapped data registers. The second assignment writes the loop identifier to the control register of the DPC. This causes an interrupt in the DPC which then uses the value stored in the control register to index into an array of functions. The source code representation of the interrupt handling routine is given in Figure 2c as the function `main`.

From Figure 2c, it can be seen that the indexing pattern of the prefetch loop is directly derived from the corresponding loop in the main program. However, the loop body consists only of prefetch directives. These directives are simply source code representations of the two types of fetch instructions and the associated address calculation instructions. The `fetch` directives represent ordinary fetch instructions that are issued to the memory system. The `fetch_t` directive indicates that the address to be prefetched will also serve as a trigger address and should therefore be replicated and placed in the trigger queue on its way to being issued to the filter buffer.

During the initial iterations of the prefetch loop, prefetches are issued without delay until the trigger queue is filled. At this point, the execution of a `fetch_t` instruction will cause the DPC to suspend execution until a trigger signal is generated. The elements of the queue are then shifted and the DPC

```
double A[512], B[512];
double C[512][512];

main( ){
 int N, i, j;

 for ( i=0; i < N; i++)
   for ( j=0; j < N; j++)

     if( C[i][j] >= 0)
        A[i] += B[j]*C[i][j];
     else
        A[i] += -C[i][j];
 ...
}
```

(a) Original Program

```
double A[512], B[512];
double C[512][512];

main( ){
 int N, i, j;

 dpc.msg[0] = N;
 dpc.loop_id = 0;

 for ( i=0; i < N; i++)
   for ( j=0; j < N; j++)

     if( C[i][j] >= 0)
        A[i] += B[j]*C[i][j];
     else
        A[i] += -C[i][j];
 ...
}
```

(b) Main Program

```
extern double A[512], B[512];
extern double C[512][512];

main( ){
 int id;

 while((id = wait4id()) >= 0)
   switch(id){
      case 0 : loop0(); break;
      case 1 : loop1(); break;
      case 2 : loop2(); break;
      ...
   }

 return;
}



void loop0(){
 int N, i, j;

 N = dpc.msg[0];

 for ( i=0; i < N; i++)
   for ( j=0; j < N; j++){

      fetch_t( C[i][j]);
      fetch( A[i]);
      fetch( B[j]);
   }

 return;
}

...
```

(c) Prefetch Program

Figure 2. The original program without any prefetching (a) is split by the compiler into a main program (b) and a prefetch program (c).

issues a new set of prefetches until another `fetch_t` instruction is encountered. Prefetching continues in this way until the prefetch stream is exhausted or a new loop index is issued by the processor.

2.3 Trigger Reference Selection

Note that in Figure 2c, references to the array `C` are used as trigger addresses in the `loop0` function. Some care must be taken when choosing which data structure to use for this purpose, since processor references to this structure dictate the timing of prefetch dispatches for the entire loop. To prevent a missed trigger, the data structure chosen must be referenced by the processor during every loop iteration for which data is prefetched. For this reason, the compiler will first attempt to find an array reference that is not allocated to a register (as `A[i]` could be in the above examples) and is also not controlled by a conditional (as is `B[j]`). Although the first case will typically be detected by locality analysis, the second case requires additional processing of the loop body to find a *must-read* or *must-modify* reference [14]. These reference types are so named because the control flow graph representation of the loop body is such that the references "must" be made during every iteration of the loop.

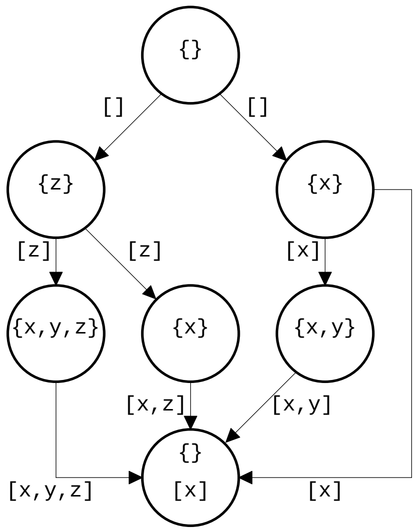
To find the must-read and must-modify references within a loop body, an algorithm based on iterative data-flow analysis [11] can be used. As an example of how this algorithm is applied, consider the loop shown in Figure 3a. In this loop, assume the compiler will attempt to cover the references, `x[i][j]`, `y[i][j]` and `z[j]` with matching prefetch streams. To select one of these references as the trigger reference, the compiler first constructs a control flow graph representation of the loop body, as shown in Figure 3b. Each graph node contains a *gen* set that holds the references occurring within the node that are candidates for the trigger reference. The nodes also contain an *in* set and an *out* set which are initially empty. The algorithm proceeds by iteratively visiting each node of the graph, computing new *in* and *out* sets at each visit. The *in* set of a node is computed as the intersection of the *out* sets of its parent nodes. The *out* set of a node is computed as the union of the node's *in* and *gen* sets.

The algorithm completes when the *out* sets of the graph nodes cease changing with each new visit of the node. At this point, the *out* set of the exit node contains the must-read and must-modify references for the

```
for (i = 0 to nrows){
  for (j = 0 to ncols){

     if ( i < j ){
        if ( z[j] != 0)
           x[i][j] = y[i][j] / z[j]
        else
            x[i][j] = 0
     }
     else{
        if ( x[i][j]  < 0)
            x[i][j] = -y[i][j]
     }
  }
}
```

(a)



(b)

Figure 3. Selecting a prefetch stream to be used to generate trigger signals. The original program (a) is used to construct a control-flow graph of the loop body (b). The *gen* sets (shown in curly brackets), *in* sets (not shown) and *out* sets (shown in square brackets) of the graph are shown as they would appear at the end of the iterative data-flow algorithm. In this example, `x[i][j]` will be used as the trigger reference.

loop body (`x[i][j]` in this example). Of these references, the compiler will choose the one that occurs earliest in the loop as the trigger reference. If the *out* set of the exit node is empty, no prefetching is done for the loop. In practice, the above analysis found no loop in the test benchmark suite for which a suitable trigger reference could not be found.

3. Evaluation Methodology

The FAST simulation package [17] is used to evaluate the performance of each prefetch scheme examined in this study. FAST is an execution-driven simulator that accepts MIPS IV [20] binaries as input. FAST is split into a front-end functional simulator that generates an instruction trace and a back-end timing simulator that reads the instruction trace to calculate timing information.

The FAST back-end process is a cycle-by-cycle simulator that models a four stage, superscalar pipeline. Instructions enter the pipeline in the fetch stage where up to four instructions are read from the simulated instruction cache. For this study, perfect instruction caching is assumed so no memory access delays occur during this stage. Up to four instructions can be dispatched to reservation stations associated with each execution functional unit in a single cycle. Instructions are buffered in reservation stations until their operands are available. The number and type of functional units and their associated latencies are shown in Table 1. With the exception of the load/store unit, the functional units are pipelined so a new instruction may be issued to each functional unit every cycle.

Functional Unit Type	Number of Units	Latency
simple integer ALU	2	1
complex integer ALU	1	5 - 20
simple floating point	1	2
complex floating point	1	20
load / store	1	1 ~ 50

Table 1. The execution functional units of the simulated processor architecture.

3.1.1 Memory Simulation

The load/store functional unit interfaces to a memory simulation module that models access delays for the data cache hierarchy and main memory. The simulated data cache allows two outstanding memory operations so a delayed memory operation does not necessarily stall the next instruction in the load/store unit’s pipeline. The cache hierarchy is comprised of an 8KB, 2-way set associative first-level cache and a 256KB, 4-way set associative second-level cache. Both caches use 32 byte blocks, an LRU replacement policy and a write-back write policy. A cache hit in the primary cache can be serviced in a single cycle while a secondary cache hit has a default latency of six cycles.

Memory accesses take 50 processor cycles to complete. “Partial” cache hits, in which a memory request is made for a cache block that is already en route from memory, can result in delays in the range of 1 to 49 cycles. An additional stall cycle may be introduced by a cache fill that is in progress due to a previous cache miss or prefetch.

3.1.2 Prefetch Simulation

Three prefetching techniques are simulated using the environment described above. Software prefetching is supported within the simulator by implementing the `pref` instruction of the MIPS IV instruction set. When this instruction is issued to the cache simulation module, the resulting prefetch request is treated in a fashion similar to a normal load request except that it never stalls the processor. This models the effect of cache hardware that does not block on a prefetch, as is done in existing systems that support software prefetching.

A 256-entry reference prediction table (RPT) is also simulated [3]. If a memory instruction, m_i, references addresses a_1 and a_2 during successive loop iterations, a stride of $\Delta = (a_2 - a_1)$ is assumed by the RPT. If $\Delta \neq 0$, a prefetch for address $A_3 = a_2 + \Delta$ will be issued, where A_3 is the predicted value of the actual address generated by the processor, a_3. Prefetching continues in this way until $A_n \neq a_n$. To allow for a prefetch distance of more than one loop iteration, prefetch addresses are calculated as $A_n = a_n + (\delta \cdot \Delta)$, where δ is the prefetch distance.

To model the execution of the prefetch controller program, a second functional simulation process is invoked, as shown in Figure 4. This simulation process is driven by the DPC program. As was done with

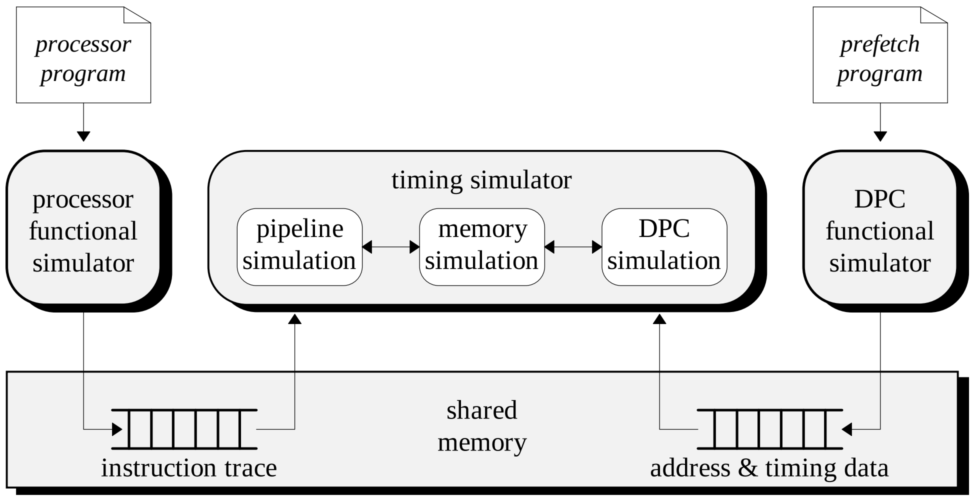


Figure 4. The simulation environment used to evaluate the DPC.

the processor simulator model, perfect instruction caching is assumed. The timing simulation of the prefetch controller also assumes that each DPC instruction completes in a single cycle. This assumption is made to reduce simulation time and is based on the observation that the instructions executed by the DPC consist of simple, integer instructions used to calculate addresses and dispatch prefetches.

3.2 Compiler Support

Although it is possible for a programmer to write separate programs for the processor and the prefetch controller, doing so would add a significant amount of complexity to the programming task. A more desirable approach is to have a compiler generate both binaries from the application source code. To approximate this functionality without developing a custom compiler, the processor program and the DPC program used in this study are each generated by applying a series of source code transformations to the original application source code.

The original source code file is first processed by Silicon Graphics' `whirl` [23] source-to-source transformer which, in addition to applying several source-level optimizations, inserts comments into the output file that specify how prefetching may be done in conforming loop structures. In particular, `whirl` identifies array references within loops that are likely to benefit from prefetching. These references are included in comment statements that are inserted into the output source code and used to guide later source code transformations.

The `whirl` output is then processed by the `janus` [29] tool which is responsible for partitioning the file into separate processor and DPC source code programs. The processor program remains relatively unchanged except for the addition of declarations used to map the DPC registers into user space and assignment statements used to initiate stores to these registers. The prefetch program is generated by scanning the `whirl` output file for prefetch comments and using these comments to generate prefetch loops. Because `whirl` is part of the MIPSpro tool set, the prefetch streams identified by the MIPSpro compiler are made available to the `janus` tool. This common set of compiler-detected prefetch streams allows for a direct comparison between the DPC and software prefetching schemes.

When generating prefetch loops, the basic structure of the loops is retained from the original program, along with the global variable names which are used to resolve addresses between the processor and prefetch programs at compile time. In a separate pass, `janus` selects which loop reference will be used to generate trigger addresses using the algorithm described in Section 2.3. After these source level transformations, the source code is compiled using the MIPSpro `f77` compiler.

To make performance comparisons as meaningful as possible, the MIPSpro compiler is also used to generate the executables for the base and software prefetching cases. Note that the simulation of the reference prediction table is driven by the base case executable since the RPT requires no run-time support from the processor. Programs were compiled using identical compiler options, with the exception of the compiler flag used to control fetch instruction scheduling which was enabled only when generating executables for the software prefetching case.

3.3 Workload Selection

The results presented in the next section are based on simulated runs of several SPECfp95 benchmarks [28]. In each case, the entire benchmark is executed by the simulator, including all library calls, but excluding operating system calls which are trapped by the simulator. The default parameter configurations are scaled down to keep simulation times manageable, as shown in Table 2. Despite these reductions, the benchmark parameters chosen for the simulation runs are large enough to represent non-trivial program runs with realistic memory demands.

Benchmark	Default Parameters		Simulated Parameters		Base Runtime
	size	steps	size	steps	($\times 10^9$ processor cycles)
`tomcatv`	513×513	750	257×257	350	6.92
`swim`	512×512	900	256×256	400	7.23
`hydro2d`	40	200	40	25	14.7
`mgrid`	64×64×64	25	64×64×64	4	9.44
`applu`	33×33×33	300	23×23×23	200	15.1
`turb3d`	64×64×64	111	64×64×64	11	4.46

Table 2. Benchmarks used to evaluate the data prefetch controller.

4. Results

All three prefetching schemes evaluated in this study provide a means of varying the prefetch distance, δ, used to calculate prefetch addresses. The MIPSpro compiler allows a fixed prefetch distance to be specified at compile time via a compiler flag. As described in Section 3.1.2, the base RPT mechanism has been extended to allow for variable prefetch distances. Finally, the depth of the prefetch controller’s trigger queue can be varied to achieve the desired prefetch distance. Although it is possible for both software prefetching and the DPC to adjust the prefetch distance to suit a particular program, and even a particular loop within a program, a constant δ is used for each scheme here.

Figure 5 illustrates the performance effect of varying the prefetch distance for each of the prefetch mechanisms. In these graphs, all simulated benchmark run-times are normalized to the run-time attained

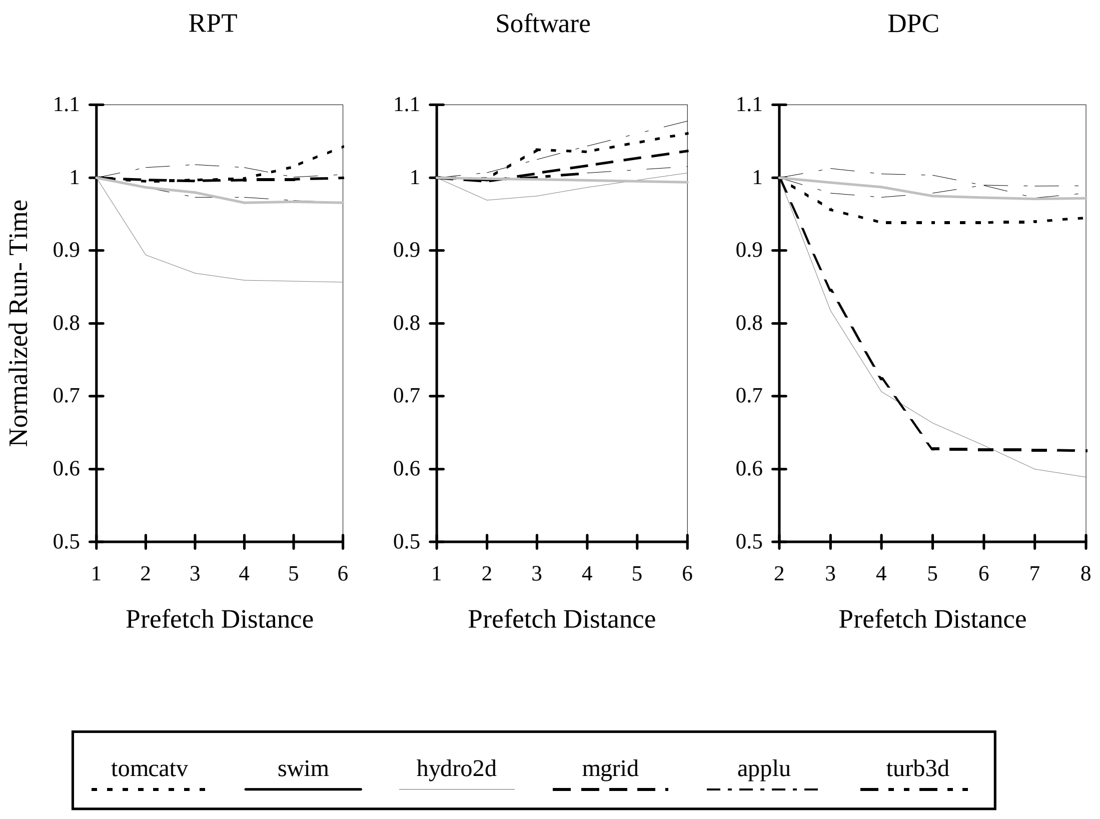


Figure 5. The effect of prefetch distance on execution time for three prefetching mechanisms. Run-times for each graph are normalized to the minimum prefetch distance for each scheme so direct run-time comparisons cannot be made from this figure (see Figure 6).

with the shortest prefetch distance. This corresponds to $\delta = 1$ for software and RPT prefetching. The minimum prefetch distance for the DPC is $\delta = 2$ because the trigger queue always maintains this minimum depth to insure that a new trigger address is available to fill the head position in the queue.

The performance curves for the RPT indicate that a prefetch distance of four will yield the best average run-time for this scheme. For larger prefetch distances, the RPT's accuracy decreases due to a higher frequency of unnecessary prefetches. These unnecessary prefetches occur at the end of a constant stride referencing pattern when the prefetch address calculations cause the RPT to prefetch data past array bounds. This is particularly a problem with reference patterns such as a row-wise traversal of a Fortran array. In this case, the last δ accesses to every row of the array will cause the RPT to "over-shoot" when calculating prefetch addresses because the data the for first δ elements of the next row actually reside at lower addresses.

The run-times of the benchmarks under the software prefetching scheme indicate that, on average, the best performance is achieved with a prefetch distance of two. This relatively low value is a result of two effects. First, the introduction of prefetch instructions into loop bodies increases the execution time of each loop iteration. This additional processing allows more time for prefetched data to be returned from memory, thereby reducing the prefetch distance necessary to bring data into the cache for a later loop iteration. The second, and more significant, effect is a loss of accuracy at higher prefetch distances much like that experienced by the RPT. Although software techniques have been proposed to avoid this effect [13], in practice, these techniques are not always employed either because the loop structure does not allow it or because the associated instruction overhead outweighs the benefit of avoiding some unnecessary or redundant prefetches.

Compared to software or RPT prefetching, relatively little overhead accompanies larger prefetch distances when the DPC is used. As a result, execution time improvements can be seen even for comparatively large prefetch distances, as shown in Figure 5. In particular, the benchmarks `hydro2d` and `mgrid` show marked improvements with larger values of δ. Given that the other benchmarks also tend to improve with longer prefetch distances, a trigger queue depth of six elements is used for the remainder of the performance evaluations.

Given the prefetch distances derived above for each scheme, the resulting benchmark run-times are given in Figure 6 where the run-times have been normalized to the base case given in Table 2. In this graph,

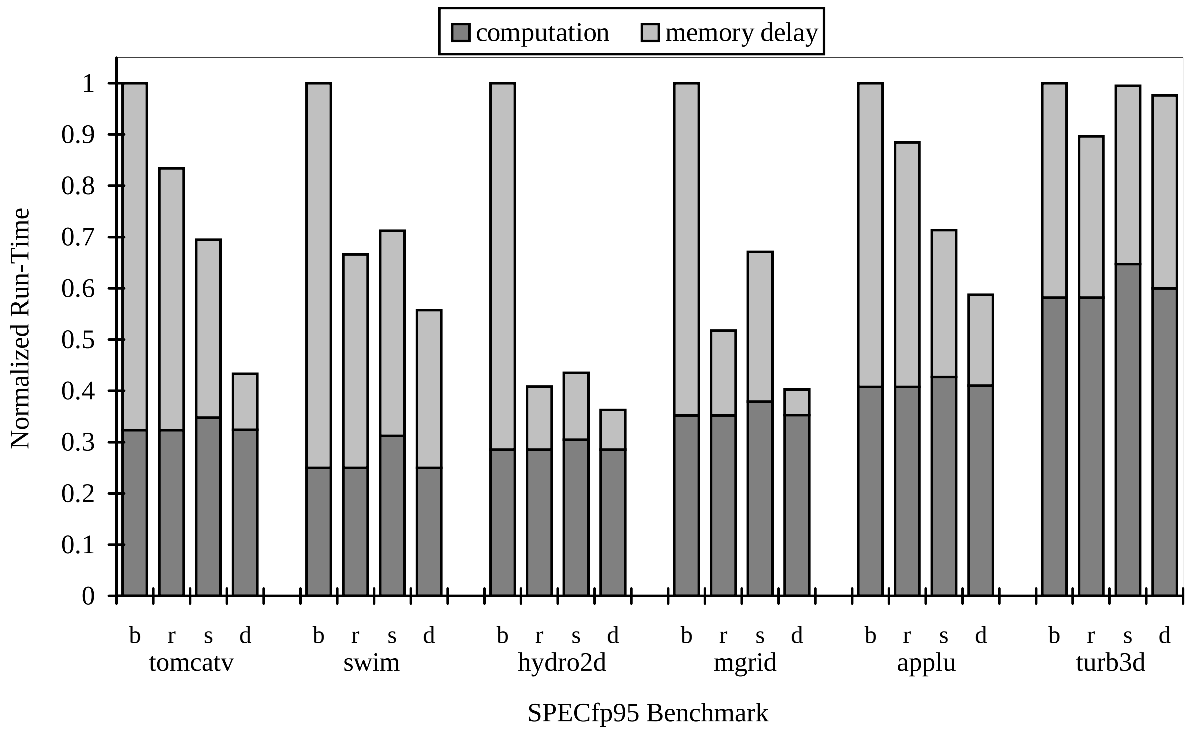


Figure 6. Simulated benchmark run-times normalized to the base case with no prefetching. Run-times are broken down into computation and memory delay components. In this figure, *b* = base, *r* = RPT, *s* = software and *d* = DPC.

run-times are divided into a computational component and a memory delay component. This figure shows that between 42% and 75% of the overall benchmark run-times are spent waiting for memory when prefetching is not used. In general, these delays are significantly reduced when prefetching is employed with the DPC consistently outperforming the other prefetch mechanisms, with one exception. In contrast to the other benchmark programs, `turb3d`'s best run-time was achieved when the RPT was used. Although the RPT is able to detect prefetching opportunities in `turb3d` at run-time, compiler analysis of this program uncovered relatively few of these opportunities as indicated by the poor performance of the two software-driven schemes.

Focusing on the computational component of the run-times, it can be seen from Figure 6 that software prefetching produces noticeable instruction overhead in the processor's execution stream. The amount of computational overhead introduced by the software scheme varies from 4.6% for `applu` to 25% for the `swim` benchmark. In comparison, the processor instructions necessary to control the DPC do not significantly affect the processor's overall computation time. Because the reference prediction table is a pure hardware prefetch mechanism, its computational component is identical to that of the base case.

Although the DPC introduces less instruction overhead than software prefetching, its primary advantage is the superior cache utilization yielded by this approach, as shown in Figure 7a. As this figure shows, miss rates for the DPC tend to be lower than the other prefetch mechanisms, particularly in the second-level cache. In general, the effects of prefetching are more pronounced in the L2 cache because prefetched data are less likely to be prematurely evicted from the L2 due to its larger size and associativity. An example of this last observation is the L1 miss rates for the `swim` benchmark where prefetching actually increases the miss rate by polluting the cache with prefetched data that is rarely used. Note that this same benchmark *does* benefit from prefetching in the second-level cache, however.

To better understand the effect each prefetch mechanism has on run-times and cache miss rates, two metrics of prefetch performance were calculated during each simulation run. *Prefetch accuracy* and *coverage* each measure a different aspect of prefetch performance and are therefore useful in differentiating the prefetch schemes under study.

Prefetch accuracy is defined as

$$\text{accuracy} = \frac{pref_{hit}}{pref_{hit} + pref_{bad}} \times 100$$

where $pref_{\text{hit}}$ is the number of prefetches issued that result in a cache hit and $pref_{\text{bad}}$ is the number of subsequent hits would occur regardless of whether the block was prefetched or not. Also, if a block

Benchmark	L1 Cache Miss Rate (%)				L2 Cache Miss Rate (%)			
	base	RPT	SW	DPC	base	RPT	SW	DPC
`tomcatv`	22.43	18.88	19.32	16.29	22.09	16.44	9.51	0.05
`swim`	74.89	75.52	75.21	79.44	8.97	2.28	2.22	0.12
`hydro2d`	19.56	2.76	4.25	1.08	63.66	23.91	20.87	7.48
`mgrid`	12.12	2.29	7.09	0.64	46.12	17.24	24.42	5.26
`applu`	9.84	7.08	3.29	4.43	62.60	55.21	30.60	21.08
`turb3d`	5.96	4.37	5.13	5.05	27.46	16.78	24.44	21.46

(a)

Benchmark	Prefetch Accuracy (%)			Prefetch Coverage (%)		
	RPT	SW	DPC	RPT	SW	DPC
`tomcatv`	39.80	69.21	83.11	3.97	5.82	8.73
`swim`	30.51	31.48	77.53	6.42	6.64	8.47
`hydro2d`	40.98	52.17	95.38	16.95	15.60	18.84
`mgrid`	16.16	32.66	99.35	10.19	5.09	11.21
`applu`	8.22	51.52	92.52	2.21	6.54	7.22
`turb3d`	4.86	5.71	79.20	2.53	1.02	1.02

(b)

Figure 7. Indicators of cache and prefetch performance.

resides in both cache levels and a hit occurs in the L1, subsequent hits in the L2 are not counted as prefetch hits. Given this definition, prefetch accuracy can be thought of as the fraction of prefetches that result in a cache hit that would not have otherwise occurred for a particular program run.

Figure 7b shows a generally wide separation in prefetch accuracy among the three prefetch mechanisms. Without the aid of compile-time information, the RPT uniformly provides the lowest prefetch accuracy due to the higher degree of speculation inherent in this scheme. Because software prefetching techniques must often sacrifice some accuracy to keep instruction overhead low, the prefetch accuracy measurements for the pure software scheme are lower than the those of the prefetch controller. Finally, the DPC's filter buffer is able to eliminate more redundant prefetches than the other two techniques, yielding a consistently higher prefetch accuracy for the DPC.

While a highly accurate prefetch mechanism is desirable, it is of little use if it prefetches only a small amount of data. For example, the DPC produced a prefetch accuracy of 79.2% for the `turb3d` benchmark, yet the RPT achieved a better run-time for this benchmark with an accuracy rating of only 4.86% because the RPT prefetched more cache blocks that resulted in prefetch hits. Prefetch coverage quantifies this effect by measuring the number of useful prefetches as a fraction of the total processor references. More formally, prefetch coverage is defined as

$$\text{coverage} = \frac{pref_{hit}}{loads + stores} \times 100$$

where *loads* is the total number of read requests issued by the processor, *stores* is the total number of write requests and $pref_{\text{hit}}$ is as defined above. The results of the prefetch coverage measurements are given in Figure 7b where it can be seen that the prefetch controller tends to provide better coverage than the other two schemes. Again, the exception to this observation occurs with the `turb3d` benchmark where the DPC and software prefetching produce equally small coverage values while the RPT shows over twice the degree of coverage. This effect is also shown indirectly in the memory delay component of the execution times given in Figure 6.

4.1.1 Cost/Benefit Analysis

Although the DPC outperforms software prefetching when similar processor and cache configurations are assumed, such comparisons do not account for the additional hardware overhead introduced by the prefetch controller. In reality, the resources freed by using pure software prefetching instead of a DPC would be reallocated to other system components. In the following analysis, it is assumed that these freed resources would be used to increase the first level data cache size.

To better evaluate the advantages of implementing a DPC over a larger cache, it necessary to determine the hardware resources required by each. Although such data is available for existing cache designs, this information is estimated for the DPC by examining the hardware costs associated with a similar device, the ARM 940T [9] processor. The 940T is a general purpose RISC processor with a five-stage pipeline capable of executing a range of integer instructions. Because the 940T is designed primarily for embedded applications, its physical dimensions are kept small with a processor core consisting of 111,000 transistors, occupying 4.8mm^2 using a 0.35µm device fabrication process. The addition of 4KB instruction and data caches increases these dimensions to 802,000 transistors and 15.5mm^2.

The MIPS R10000 [33] processor and its 32KB data cache will be used as the basis for evaluating the relative costs of using a larger cache. Like the 940T, the R10000 uses a 0.35µm fabrication process, allowing for an accurate comparison based on chip area. The data cache of the R10000 occupies approximately 14mm^2 of chip area which is slightly smaller than the 940T. However, it should be noted that the R10000 uses 4 metal layers as compared to the 3 layers used in the 940T and the DPC is likely to be a simpler device than the 940T. Given these observations, it can be safely assumed that the addition of a DPC would consume no more chip area than doubling the L1 data cache size, using current technology.

In Figure 8, the run-times achieved by using software prefetching with a first level cache size of 16KB are compared to the previously reported run-times for software prefetching and the DPC using an 8KB L1 cache. As this data shows, increasing the first-level cache generally has little effect on the overall run-

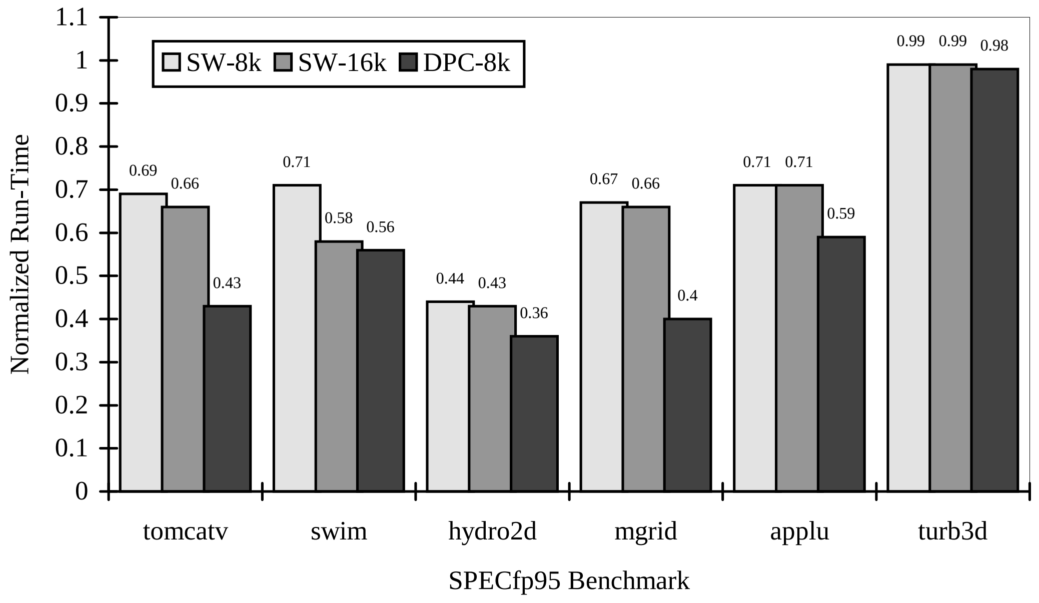


Figure 8. The effect of cache size on software prefetching relative to the use of a DPC.

time of the software prefetching scheme, with the exception of the `swim` benchmark. This benchmark suffers from cache thrashing in an 8 KB cache and therefore benefits more from an increase in cache size than the other benchmarks. In general, however, the DPC-enabled run-times in Figure 8 justify the additional hardware resources estimated above.

5. Related Work

Decoupled architectures [18,26,27] attempt to separate the tasks of accessing and processing data by providing distinct hardware units for each task. Rather than fetch data speculatively, these systems attempt to overlap ordinary loads and stores with other computation. This approach often restricts the amount of time between when the address is made available to the memory system and when the data is needed by the processor.

The DPC's trigger queue shares much in common with a *stream buffer* [10,16]. Stream buffers are FIFO queues that are used to hold prefetched cache blocks. When the address of the block at the head of the queue matches the current processor reference, this block is brought into the cache while the remaining queue entries are shifted up and the next block is prefetched into the tail position. This approach requires that each prefetched block be referenced by the processor before new prefetches can be issued, whereas the trigger queue requires only that the prefetch stream used to generate trigger addresses be fully referenced.

Benchmark	tomcatv	swim	hydro2d	mgrid	applu	turb3d
% Prefetch Streams w/ Multiple Strides	85%	62%	69%	84%	52%	54%

Table 3. The precentage of prefetch streams detected by the MIPSpro complier that consist of more than one stride.

The use of programmable *prefetch engines* has been proposed [4,6,24] as a means of extending the basic idea of a reference prediction table [3] to a programmable device. At run-time, the application program provides starting address, stride and other information that the prefetch engine uses to generates a constant stride prefetch stream. Although these devices require less hardware to implement than the DPC, they are limited to prefetching cache blocks separated by a fixed stride. Analysis of the prefetch addressing patterns produced by programs compiled with the MIPSpro compiler show that the majority of the prefetch streams detected by the compiler do not consist of references that are uniformly separated by a single, fixed stride. As shown in Table 3, between 52% and 85% of the prefetch streams initiated at run-time contain multiple strides. Such reference patterns typically arise from indexing multi-dimensional arrays or arrays that are accessed in a striped or blocked manner.

Shared data caches [15] and hierarchical caches [5,30] have been proposed as a natural store for data that is shared among the nodes of a multiprocessor. In such a system, it may be possible to have one processor to mimic the functionality of a DPC by prefetching data for another processor sharing the same cache, although it is not clear how the actions of the two processors would be coordinated without introducing additional hardware support. It is interesting to note, however, that the run-time improvements achieved by the prefetch controller are greater than typically would be expected from running these programs on a two processor system using traditional parallel processing techniques.

6. Conclusions

Our simulation results suggest that the proposed data prefetch controller can significantly decrease the run-time of data-intensive applications by improving cache utilization. Although the magnitude of the these improvements is dependent on application characteristics, five of the six programs tested ran between 1.7 and 2.7 times faster with the addition of a DPC. Although the reference prediction table outperformed the DPC for one benchmark program, the prefetch controller was found to be a much more accurate prefetch mechanism, resulting in run-times for the remaining benchmarks that were 1.1 to 1.9 times faster than the corresponding run-times achieved with an RPT. This accuracy is partially a result of compile time information that is used to generate the prefetch program executed by the prefetch controller. Although software prefetching uses similar compiler analysis, this approach is hampered by the need to mix prefetching with computation in the same program. By separating these tasks and providing hardware support for removing redundant prefetches, the DPC demonstrated better accuracy and coverage than pure software prefetching. In addition, the instruction overhead introduced into the processor by the prefetch controller was found to be much less than that of a software prefetching scheme. These factors combined to produce run-times using a DPC that were an average of 1.3 times faster than those achieved with software prefetching.

These results show that the DPC is an effective means of reducing the run-times of scientific and engineering applications. Because such programs often exhibit very poor cache utilization, the majority of prefetch schemes concentrate on reducing the memory delays associated with array-based referencing patterns. Extending prefetching techniques to a wider range of applications is an active area of research. The DPC may be particularly useful in providing prefetch support for non-array based codes because more sophisticated prefetching algorithms can be off-loaded to the DPC to handle the comparatively complex referencing patterns found in these applications. Further research will be necessary to establish compiler techniques that will allow the prefetch controller to target a more general class of applications.

7. Acknowledgments

This work was supported in part by National Science Foundation grant no. MIP-9610379 and the Minnesota Supercomputing Institute. Steve VanderWiel was supported in part by an IBM Graduate Fellowship.

References

1. Bernstein, D., C. Doron and A. Freund, “Compiler Techniques for Data Prefetching on the PowerPC,” *Proc. International Conf. on Parallel Architectures and Compilation Techniques*, June 1995, p. 19-16
2. Chan, K.K., et al., “Design of the HP PA 7200 CPU,” Hewlett-Packard Journal, Vol. 47, No. 1, February 1996, p. 25-33.
3. Chen, T-F. and J-L. Baer, “Effective Hardware-Based Data Prefetching for High Performance Processors,” *IEEE Transactions on Computers,* Vol. 44, No. 5, May 1995, p. 609-623.
4. Chen, T-F., “An Effective Programmable DPC for On-chip Caches,” *Proc. International Symposium on Microarchitecture,* Ann Arbor, MI, November 1995, p. 237-242.
5. Cheriton, D.R., H.A. Goosen and P.D. Boyle, “Multi-Level Shared Caching Techniques for Scalability in VMP-MC,” *Proc. 16th International Symposium on High-Performance Computer Architecture (ISCA 89),* Jerusalem, Israel, May 1989, p. 16-24.
6. Chiueh, T., “Sunder: A Programable Hardware Prefetch Architecture for Numerical Loops,” *Proc. Supercomputing ‘94,* Washington D.C., November 1994, p. 488-497.
7. Dahlgren, F., M. Dubois and P. Stenstrom, “Fixed and Adaptive Sequential Prefetching in Shared-memory Multiprocessors,” Proc. of the 1993 International Conference on Parallel Processing, St. Charles, IL, August 1993, p. I-56-63.
8. Fu, J.W.C., J.H. Patel and B.L. Janssens, “Stride Directed Prefetching in Scalar Processors,” Proc. 25th Annual International Symposium on Microarchitecture, Portland, OR, Dec. 1992, p. 102-110.
9. Jaggar, D. *The ARM Architecture Reference Manual*, Prentice Hall, Upper Saddle River, NJ, 1997.
10. Jouppi, N.P., “Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers,” *Proc. 17th International Symposium on Computer Architecture,* Seattle, WA, May 1990, p. 364-373.
11. Kidall, G.A., “A Unified Approach to Global Program Optimization”, *Proc. Symposium on Principles of Programming Languages,* Boston, MA, October 1973, p. 194-206.
12. Klaiber, A.C. and Levy, H.M., “An Architecture for Software-Controlled Data Prefetching,” *Proc. 18th International Symposium on Computer Architecture,* Toronto, Ont., Canada, May 1991, p. 43-53
13. Mowry, T.C., Lam, S. and Gupta, A., “Design and Evaluation of a Compiler Algorithm for Prefetching,” *Proc. Fifth International Conf. on Architectural Support for Programming Languages and Operating Systems,* Boston, MA, Sept. 1992, p. 62-73.
14. Muchnick, S.S., *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco, CA, 1997.
15. Nayfeh, B.A, K. Olukotun and J.P. Singh, “The Impact of Shared-Cache Clustering in Small-Scale Shared-Memory Multiprocessor,” *Proc. 22nd International Symposium on High-Performance Computer Architecture,* San Jose, CA., February 1996, p. 74-84.
16. Palacharla, S. and R.E. Kessler, “Evaluating Stream Buffers as a Secondary Cache Replacement,” *Proc. 21st International Symposium on Computer Architecture*, April 1994.
17. Plender, C.G., *FAST-MIPS: A MIPS ISA Simulation Environment*, Master’s Project, University of Minnesota Department of Electrical and Computer Engineering, June, 1997.

18. Pleszkun A.R and E.S. Davidson, “Structured Memory Access Architecture,” *Proc. International Conference on Parallel Processing*, Bellaire, MI, August 1983, p. 461-471.

19. Porterfield, A.K., *Software Methods for Improvement of Cache Performance on Supercomputer Applications*, Ph.D. Thesis, Rice University, May 1989.

20. Price, C., *The MIPS IV Instruction Set*, MIPS Technologies, Inc., Mountain View, CA, 1996.

21. Przybylski, S., “The Performance Impact of Block Sizes and Fetch Strategies,” *Proc. of the17th Annual International Symposium on Computer Architecture*, Seattle, WA, May 1990, p. 160-169.

22. Santhanam, V., E.H. Gornish and W.C. Hsu, “Data Prefetching on the HP PA-8000,” *Proc. 24th Annual International Symposium on Computer Architecture*, Denver, CO, June 1997.

23. Silicon Graphics, Inc., *MIPSpro Compiling and Performance Tuning Guide*, Mountain View, CA. 1997.

24. Skeppstedt, J. and M. Dubois, “Hybrid Compiler/Hardware Prefetching for Multiprocessors Using Low-Overhead Cache Miss Traps,” *Proc. International Conference on Parallel Processing*, Bloomington, IL, August 1997, p. 298-305.

25. Smith, A.J., “Cache Memories,” Computing Surveys, Vol. 14, No. 3, Sept. 1982, p. 473-530.

26. Smith, J.E., “Decoupled Access/Execute Computer Architectures,” *Proc. Nineth Annual Symposium on Computer Architecture*, Austin, TX, April 1982, p. 112-119.

27. Smith, J.E., “Dynamic Instruction Scheduling and the Astronautics ZS-1,” *IEEE Computer*, Vol. 22, No. 7, July 1989, p, 21 - 35.

28. SPEC; Standard Performance Evaluation Corporation, *http://www.specbench.org*, 1998

29. VanderWiel, S.P., *Masking Memory Latency with a Compiler-Assisted Data Prefetch Controller*, Ph.D. Thesis, University of Minnesota Department of Electrical and Computer Engineering, September, 1998.

30. Wilson, A.W., “Hierarchical Cache / Bus Architecture for Shared Memory Multiprocessors,” *Proc. 14th International Symposium on High-Performance Computer Architecture (ISCA 87)*, Pittsburgh, PA., June 1987, p. 244-252.

31. Wolf, M.E. and M.S. Lam, “A Data Locality Optimizing Algorithm,” *Proc. SIGPLAN ‘91 Conference on Programming Language Design and Implementation*, June 1991, p. 30 - 44.

32. Wulf, W. A. and S. A. McKee, “Hitting the Memory Wall: Implications of the Obvious,” *ACM Computer Architecture News*, Vol. 23, No. 1, March 1995, p. 20 - 24.

33. Yeager, K.C., “The MIPS R10000 Superscalar Microprocessor,” *IEEE Micro*, Vol. 16, No. 2, April 1996, p. 28 - 41.