TOLERATING FIRST LEVEL MEMORY ACCESS LATENCY IN HIGH-PERFORMANCE SYSTEMS

William Y. Chen Scott A. Mahlke Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois
Urbana, Illinois 61801

Abstract - - *In order to improve performance, future parallel systems will continue to increase the processing power of each node in a system. As node processors, though, can execute more instructions concurrently, they become more sensitive to the first level memory access latency. This paper presents a set of hardware and software techniques, collectively referred to as* register preloading*, to effectively tolerate long first level memory access latency. The techniques include speculative execution, loop unrolling, dynamic memory disambiguation, and strip-mining. Results show that register preloading provides excellent tolerance to first level memory access latency up to 16 cycles for an issue 4 node processor.*

INTRODUCTION

The objective of designing a high-performance system is to speed up the execution of application programs. An important approach to achieve this objective is to exploit program parallelism at both the instruction level and the multiprocessor level. For example, the Alliant FX/2800 system [1] supports parallel execution among its 28 Intel i860 node processors each of which is capable of completing two instructions per clock cycle. The Intel Touchstone system and the Thinking Machines CM5 system [2] provide additional examples where program parallelism is exploited at both the instruction level and the multiprocessor level. The potential performance gain of this approach, however, can be severely reduced by long data access latencies. In addition, as the number of instructions a node processor can execute per cycle increases, the sensitivity to first level memory access latency also increases [3]. This paper proposes a method, referred to as *register preloading*, to tolerate the data access latency of the first level memory, often designed as the top level cache, in high-performance parallel systems.

There are at least three major factors that contribute to long access latencies to the first level memory in high-performance systems. First, due to difficulties of maintaining coherence among on-chip caches in multiprocessor systems, node processors in multiprocessor systems often bypass their on-chip caches when accessing shared data. In this case, the first level memory is the off-chip cache as far as the shared data is concerned. Since the processor clock period is typically much shorter than the access time of a large off-chip coherent cache, this design decision often results in long first level memory access latency in multiprocessor systems.

Second, to achieve more predictable data memory access timing, high-performance systems often come without data cache or they bypass cache when accessing large data sets. For example, the Multiflow TRACE system [4] and the Cydrome Cydra-5 system [5] directly access an interleaved main memory for their data. In this case, the first level memory is the main memory whose access latency is usually more than ten processor clock cycles.

The third factor arises when a shared data cache is used to provide a large coherent first level memory to node processors in a shared memory multiprocessor system. The motivation is to avoid replicating data into private caches so that better utilization of cache storage can be achieved. Also, the cache coherence problem is eliminated by sharing the cache. An example is the Alliant FX/2800 system [1] where all 28 processors access a shared data cache as their first level memory. The latency of this first level memory is approximately 14 cycles and varies due to bank conflicts. To achieve high performance in such systems, parallel programs must be able to tolerate more than 10 cycles of first level memory access latency.

Register preloading is a suite of hardware and software methods to collectively create opportunities to execute many useful instructions between a memory load and the instructions that use the fetched data. Speculative execution allows instructions to be scheduled before their preceding conditional branches. Dynamic memory disambiguation allows memory loads to be scheduled before their preceding memory stores even in the presence of inconclusive data dependence analysis results. Loop unrolling increases the scope for code motion within a loop body.[1] Strip-mining is used to create sequential inner loops from parallel DOALL and DOACROSS loops so that these inner loops can be further transformed with register preloading. By creating a large code motion scope and eliminating constraints due to conditional branches and inconclusive data dependence analysis results, these methods jointly enable the instruction scheduler to schedule many useful instructions between the load instructions and their dependents.

To demonstrate the usefulness of register preloading, we have implemented its compiler transformations and conducted simulation studies on several parallel matrix kernels. Two important considerations are addressed. First, how sensitive is the node processor performance to the first level

[1] Note that software pipelining can be used to achieve the same objective.

memory latency without register preloading? Second, how much of the first level memory latency can register preloading actually tolerate for different node processor configurations?

Related Work

Compiler assisted methods to reduce the penalty of traversing the memory hierarchy have been developed in recent years. Data prefetching [6] [7] [8] is used to cope with the problem of long latency for moving data from the second level memory into the first level. This is accomplished by requesting the transfer of a piece of data from the second level memory to the first level memory when the data is expected to be used in the future. Loop restructuring [9] [10] can be used to improve local memory performance by grouping references to the same memory location together to improve the utilization of the first level memory. These techniques do not address the problem of long latency to access data that is available in the first level memory. However, by combining these techniques with register preloading, one can tolerate access latencies of both the first level and the second level memories. Redundant data access elimination [11] [12] detects and reuses a copy of a variable if it is available in a processor register. By reducing the needs to access the first level memory, this technique complements register preloading to further reduce the negative effects of long first level memory access latencies.

REGISTER PRELOADING

Register preloading consists of a hardware support for speculative execution and a suite of compiler transformations to facilitate effective instruction scheduling of load instructions and their dependents. The goal of register preloading is to transform parallel program structures so that long first level memory access latencies can be fully overlapped with useful computation. In this section, the procedure for register preloading is described in the context of DO, DOALL and DOACROSS loops in parallel FORTRAN programs.

Speculative Execution Support

The key hardware feature for register preloading is speculative execution support. Speculative execution refers to executing an instruction before it is certain that its execution is required. In terms of register preloading, this occurs when a load instruction is moved above a preceding conditional branch or moved to a previous loop iteration. The minimal hardware feature required to support register preloading is a set of non-trapping load instructions whose exception conditions will be ignored by the processor [4]. When a non-trapping load instruction is executed and an exception occurs, the processor simply ignores the exception and writes a garbage value into the destination register. The compiler will ensure that the garbage value will not be used incorrectly by the subsequent instructions.

Note that few commercial microprocessor architectures provide non-trapping load instructions. Therefore, these architectures require an extension to their instruction sets to fully support register preloading. The usefulness of speculative execution in register preloading will be further discussed in the next subsection.

Register Preload By One Iteration

The basic idea of register preloading is to initiate memory accesses far enough in advance so that the data is available in a processor register at its time of use in order to reduce idle execution cycles. For loops, this can be realized by having the current iteration to load data for future iterations. The number of iterations that the memory accesses should be initiated before the use of their data is a function of the memory access latency and the size of each iteration.

A simple example suffices to illustrate the basic loop transformation for register preloading one iteration ahead. In the following example, the original code segment loads and uses one element of the B array in every iteration. Assume that in order to tolerate a moderate memory latency, the access to each B element should be moved to one iteration before it is used. The result of the register preloading transformation is shown in code segment b) below. The compiler creates a temporary variable, *temp1*, that is assigned to a processor register. The first element of B is loaded before entering the loop. During each iteration, the B element used by the next iteration is preloaded into *temp1*.

a) Original code segment

```
do J = 1 , N
    A(J) = B(J) * K
endo
```

b) One iteration register preloading

```
     temp1 = B(1)
do J = 1 , N
     A(J) = temp1 * K
     temp1 = B(J+1)
endo
```

Note that one extra memory load to $B(N+1)$ is executed after the transformation. Since array B may be declared with only N elements in the original program, this extra memory access may generate an illegal address and cause an access violation. However, this access violation is ignored by a processor with speculative execution support. Note also that a garbage value will be assigned to *temp1* when the execution of the loop completes. However, this garbage value will not be used incorrectly by the subsequent computation because *temp1* is introduced by the compiler for the purpose of preloading the B elements only. The compiler can easily ensure that *temp1* is never used before defined after the execution exits the loop.

At this point, one may argue that speculative execution is not essential for the above example. One can simply reduce the loop bound by 1 and execute the last iteration after exiting the loop. However, this approach would require additional conditional statements to test for the special case where N is equal to 1 unless N is a compile-time constant. The complexity increases if the memory latency requires the memory data to be preloaded more than one iteration ahead. Therefore, speculative execution support reduces the compile-time complexity and run-time overhead of register preloading for simple DO loops.

Speculative execution support becomes more essential for loops with conditional statements. This is illustrated in the following example. The original loop is shown in code segment a) where each loop iteration loads an element of array B only if its index is greater than 0. Loading an element of array B in a previous iteration would either require moving the conditional statement along with it, or speculative execution support. The former incurs high compile-time com-

plexity in general. With speculative execution, an element in array B can be preloaded before its use is certain. The result of the transformation is shown as code segment b). [2]

a) Original code segment

```
do J = 1 , N
    if (C(J) > 0)
        A(C(J)) = B(C(J)) * K
endo
```

b) Preloading with speculative support

```
i1:     temp = B(C(1))
    do J = 1 , N
        if (C(J) > 0)
            A(C(J)) = temp * K
i2:     temp = B(C(J+1))
    endo
```

Register Preload By Multiple Iterations

The simple method presented in the previous subsection cannot preload a memory data more than one iteration before its use. The problem is that the preload instruction for the next iteration cannot be moved beyond the use of the temporary variable in the current iteration. This problem is illustrated in code segment a) of the following example. Note that the preload immediately follows the use of *temp1*; the anti dependence prevents the preload from being moved any further.

a) One-iteration register preload

```
    temp1 = B(1)
do J = 1 , N
    A(J) = temp1 * K
    temp1 = B(J+1)
endo
```

One plausible solution is to introduce D additional temporary variables when the preload is done D iterations ahead. A data is preloaded into *tempD* in the current iteration, moved from one temporary variable to the next in each successive iteration, until it is used D iterations later. The result of this code transformation for $D = 2$ is shown in code segment b). Unfortunately, this solution does not improve the situation. The problem is that *temp2* loaded in the current iteration is used as the source operand of the register copying instruction $i2$ in the next iteration. Again, the anti dependence prevents the preload from being moved any further. This solution introduces extra overhead instructions, $i1$ and $i2$, without enlarging the distance between the load and the use for the B array elements.

b) Two-iteration register preload without unrolling

```
        temp1 = B(1)
i1:     temp2 = B(2)
    do J = 1 , N
        A(J) = temp1 * K
i2:     temp1 = temp2
        temp2 = B(J+2)
    endo
```

[2] Note that in our implementation, $C(J)$ would be further preloaded. However, $C(J)$ is not preloaded in the example to keep the example simple.

Loop unrolling, though, may be utilized by the compiler to preload D iterations ahead. In general, a loop must be unrolled D-1 times to allow such a preload. Code segment c) shown below illustrates this solution. The loop is first transformed perform one-iteration register preload before the loop body is unrolled. After unrolling, the temporary variables are renamed and the index used in the register preload is adjusted to preload the B elements two iterations ahead. Since the B elements are now preloaded two iterations ahead, the elements used in the first two iterations are preloaded before the execution enters the loop. Note that the preload instructions and their dependents are now separated by two iterations of the original loop.

c) Preloading with unrolling

```
    temp1 = B(1)
    temp2 = B(2)
do J = 1 , N , 2
    A(J) = temp1
    temp1 = B(J+2)
    A(J+1) = temp2
    temp2 = B(J+3)
endo
```

The desired number of unrolls for a loop is a function of the memory access latency and the size of the original loop body. However, the total number of unrolls allowed is limited by the number of processor registers available to hold the preload data. It is important to keep the register pressure under control to avoid excessive spilling after the transformation. We will experimentally evaluate the increase in register pressure in the experimental evaluation section.

Dynamic Memory Disambiguation

There are situations when compile time dependence analysis cannot resolve whether two memory instructions access the same location. A common example is an array indexed by another array (e.g., A(B(I))). Typically, parallelizing compilers cannot parallelize loops with uncertain dependences. These serial loops can have dramatic effect upon the program performance. Thus, it is crucial to efficiently execute these loops. Register preloading can decrease the execution time of serial loops by reducing the number of processor idle cycles waiting for a first level memory access.

Dynamic memory disambiguation [13] was originally proposed by Nicolau for parallelization of code with inconclusive dependence analysis results. We have utilized Nicolau's method to allow a load to be moved above a store that may reference the same memory location. This enables register preloading even in the presence of inconclusive dependence analysis results.

Dynamic memory disambiguation relies on run time checking to guarantee correct program execution. For the case of register preloading, the address of the preload variable must be checked against all ambiguous store addresses before the value is used in a computation. If the two addresses match, the store value must be copied into the preload destination register to reflect the memory content change. For example, in the following unrolled code segment, the A array is indexed by an element of the B array.

```
do J = 1 , N, 2
    A(J) = A(B(J)) * K
    A(J+1) = A(B(J+1)) * K
endo
```

It is uncertain whether the uses of array A are independent of either defines of array A. Register preloading without dynamic memory disambiguation cannot move a preload above a previous store. However, with dynamic memory disambiguation, preloads can be moved much further in advance if care is taken to repair any values that are wrongly preloaded. The previous example with register preloading applied is shown below.

```
addr1 = Addr(A(B(1)))
temp1 = A(B(1))
addr2 = Addr(A(B(2)))
temp2 = A(B(2))
do J = 1 , N , 2
    A(J) = temp1 * K
    if (Addr(A(J)) == addr2) temp2 = A(J)
    addr1 = Addr(A(B(J+2)))
    temp1 = A(B(J+2))
    A(J+1) = temp2 * K
    if (Addr(A(J+1)) == addr1) temp1 = A(J+1)
    addr2 = Addr(A(B(J+3)))
    temp2 = A(B(J+3))
endo
```

It is noted that storing the addresses of the load instructions requires additional registers. With more unrolling, the number of comparison and branch instructions can become quite large. The conditional branches also introduce additional control dependences which otherwise would not exist. These problems, however, can be reduced with hardware support for predicated execution [14] [5]. The address comparisons can be performed as soon as both the load and store addresses are known. The compare result is stored in a predicate register (one boolean bit). The value reassignments are then conditionally executed based on the value of the predicate register. Predicated execution reduces register live ranges by performing the address compares early. Also, the conditional execution of value reassignments removes the additional control dependences.

Generating Sequential Inner Loops From Parallel Loops

Our discussion on register preloading has been so far based on sequential inner loops where multiprocessor parallelism is exploited in outer loops. The register preloading transformations cannot be directly applied to parallel loops because each iteration of a parallel loop cannot be guaranteed to execute on any particular processor. However, register preloading requires the iteration that preloads a datum and the one that uses the datum to be executed by the same processor. This is achieved by creating sequential inner loops within each parallel loop and applying register preloading to the sequential inner loops.

Sequential inner loops can be created for parallel loops in the DOALL form in a straightforward manner. In the following DOALL construct, the loop can be strip-mined (or sectioned) into an outer loop with p iterations and an inner loop with n iterations. [3]

a) Original DOALL loop

```
doall J = 1 , N
    Statements
endo
```

b) With sequential inner loop

```
doall J = 1 , N , n
    do I = J , J+n-1
        Statements
    endo
endo
```

Register preloading is directly applicable to the resulting sequential inner loop. Note that n must be large enough to take advantage register preloading. However, if n is too close to N, there may be insufficient multiprocessor level parallelism to keep all node processors busy or to maintain load balance. A good heuristic derives the appropriate n value by considering the memory access latency, loop body size, the estimated N value, and the number of node processors in the system.

Parallel loops in the DOACROSS form present more difficulties for register preloading. The following DOACROSS loop generated by the KAP/CEDAR parallelizer [15] from the FLOW52 program of the PERFECT club suite [16] presents the problem. [4]

a) Original DOACROSS loop

```
doacross J = 1 , N
    await(0,1)
    A(m-J) = A(J)
    advance(0)
    await(1,1)
    B(m-J) = B(J)
    advance(1)
endo
```

b) With sequential inner loops

```
doacross I = 1 , N , n
    await(0,1)
    do J = I , I+n-1
        A(m-J) = A(J)
    endo
    advance(0)
    await(1,1)
    do J = I , I+n-1
        B(m-J) = B(J)
    endo
    advance(1)
endo
```

The parallelism for this DOACROSS loop results from overlapping of the computation for array B in the current iteration with the computation of array A in the next iteration. The await dependence distance is set to 1 to account for the worst case.

In order to create sequential inner loops, we apply strip-mining followed by a variation of loop distribution to the DOACROSS loop [17]. The result of the transformation is shown as code segment b) above. Note that the loop distribution can only be applied when there are no backward dependences between statements to be distributed into different inner loops. With the DO loops inside the DOACROSS loop, register preloading can be applied to the inner DO loops directly.

Although it is easy to visualize why normal DO and DOALL loops obtain speedup with register preloading, it may not be clear why a DOACROSS loop can obtain the same benefit. To illustrate this point, Figure 1 presents

[3] If ($p \times n \neq N$), the extra iterations can be peeled off as a separate sequential loop.

[4] The array index calculations are simplified for this example.

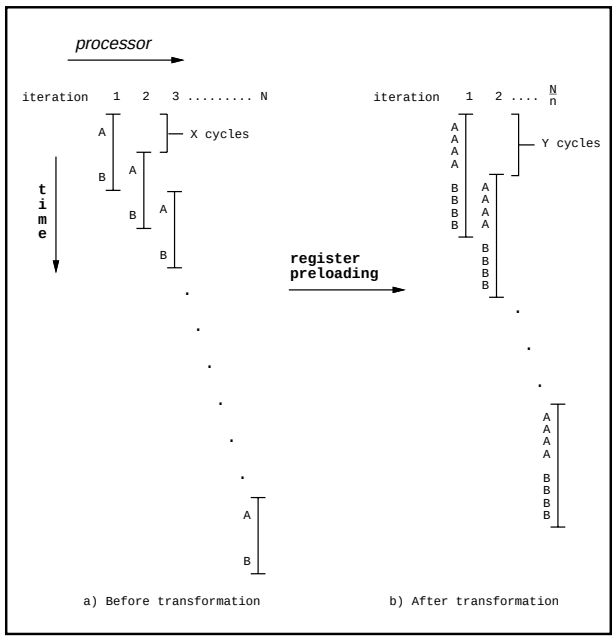


Figure 1: An example DOACROSS loop execution.

the execution of the above DOACROSS loops with and without register preloading. The multiprocessor execution timing before and after code transformations and register preloading are shown. Note that there are N iterations of the DOACROSS loop for the trace of Figure 1a, and there are N/n iterations for the trace of Figure 1b.

Assume that the successive initiations of iterations of the original DOACROSS loop are separated by x cycles. Further assume that the delay between successive initiations becomes y cycles after creating sequential inner loops. If N is large, the execution time of the DOACROSS loop can be approximated by $x \times N$ before the transformation and $y \times N/n$ after the transformation. By applying register preloading to each sequential inner loop, it is necessary to make $y < n \times x$, thereby reducing the execution time for the DOACROSS loop.

EXPERIMENTAL EVALUATION

In this section, the effectiveness of register preloading is analyzed for a set of kernel benchmarks. The ability of scalar and superscalar node processors to tolerate varying first level memory access latency with several degrees of register preloading support is compared.

Methodology

Register preloading has been implemented in the IMPACT compiler developed at the University of Illinois. The IMPACT compiler is geared towards high-performance scalar and superscalar systems. The benchmarks used in this study consist of the 4 numeric matrix kernels shown in Table 1. The performance of a node processor is evaluated with several levels of register preload support, neither compiler nor architectural support, compiler support only, and both compiler and architectural support. Execution driven simulation is used to derive the timing for various node processor architectures.

Table 1: Benchmarks.

Benchmark	Description
deter	matrix determinant
inverse	matrix inverse
matrix	matrix multiplication
sparse	solve sparse linear system

Table 2: Instruction latencies.

INT function	*latency*	*FP function*	*latency*
ALU	1	ALU	2
multiply	3	multiply	3
divide	10	divide	10
branch	2	conversion	2
load	varies	load (1 word)	varies
store	1	store	1

The performance of each node processor configuration is reported as a normalized execution time relative to a base architecture. The base node architecture is a pipelined scalar processor. The instruction set is a RISC assembly language similar to the MIPS R2000 instruction set [18]. The underlying processor has CRAY-1 style interlocking [19] and deterministic instruction latencies (Table 2). The load latency is varied from 2 to 16 cycles for the experiments. A 100% hit rate is assumed for the first level memory system.

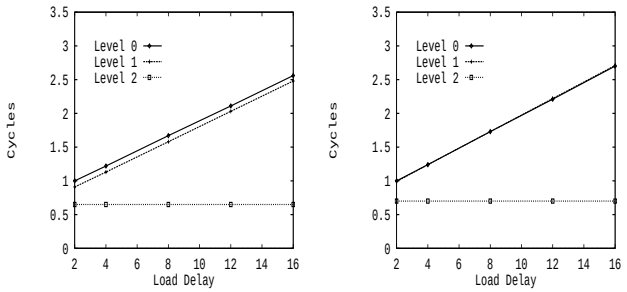
Three node processor architecture configurations are considered: a single issue scalar processor (base), a 4 issue superscalar processor, and an 8 issue superscalar processor. For the superscalar node processors, uniform function units are assumed. Therefore, each superscalar processor can issue any combination of N (4 or 8) instructions each cycle. For those systems which have architectural support for preloading, the node processors are assumed to have a non-trapping preload instruction.

Results

The normalized execution times for each benchmark and node processor architecture configuration are shown in Figures 2 – 4. In each figure, the execution times with three levels of register preloading support are given. Level 0 consists of no register preloading support. Level 1 consists of only the compiler support (strip-mining, loop unrolling, and dynamic memory disambiguation). Level 2 includes both compiler support for register preloading and architectural support for speculative execution of preload instructions.

The figures show that all node processors can effectively tolerate up to a 16 cycle load delay with both compiler and architectural support. Only negligible performance loss is observed for issue 1 and 4 node processors when the latency increases from 2 to 16 cycles. For issue 8 node processors,

a) deter b) inverse



c) matrix d) sparse

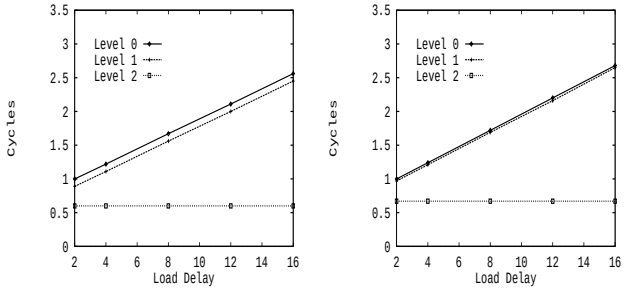
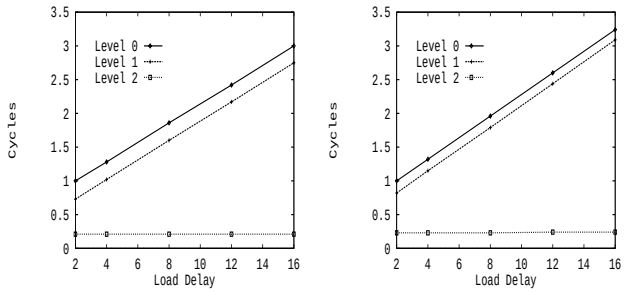


Figure 2: Normalized execution time for issue 1.

a) deter b) inverse



c) matrix d) sparse

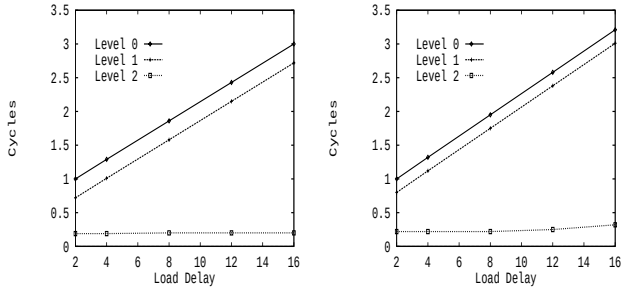


Figure 3: Normalized execution time for issue 4.

small performance losses are observed when the load delay is increased beyond 12 cycles. Without both compiler and architecture support, the performance of all node processor architecture configurations suffers dramatically. For example, the execution time of *deter* with an issue 1 node processor (Figure 2a) increases about 2.5 times when the load delay is increased from 2 to 16 cycles. Similarly, the execution time of *sparse* with an issue 4 node processor (Figure 3d) increases over 3 times when the load delay is varied from 2 to 16 cycles. The importance of register preloading support for a node processor to tolerate high memory latency is clearly shown for this set of benchmarks.

The capacity of a node processor to tolerate memory latency is not effectively increased with only compiler support. Strip-mining, loop unrolling, and dynamic memory disambiguation increase the performance of all benchmarks, however the relative decrease of performance as the load delay increases is not noticeably changed. This can easily be observed for all benchmarks by comparing the slopes of the Level 0 performance and the Level 1 performance in Figures 2 – 4. Without the ability to speculatively execute preloads, the compiler transformations do not increase a node processor's ability to tolerate high memory latency.

The performance of higher issue rate node processors is affected more dramatically by increased load delay. For example, consider *inverse* with level 0 preloading support. For an issue 1 node processor (Figure 2b), the execution time is increased 2.7 times when the load latency is increased from 2 to 16 cycles. However, for an issue 8 node processor (Figure 4b), the execution time is increased 3.3 times when the load latency is increased from 2 to 16 cycles. This behavior could be anticipated, though. As a processor is able to execute more instructions concurrently, it becomes more sensitive to data access latency, and therefore suffers more performance penalty as the data access latency is increased.

It can be seen from the figures that register preloading also increases the performance of each node processor for small load latencies, e.g., 2 cycles. A maximal increase of 10 times is observed for *matrix* with an issue 8 node processor (Figure 4c). This increase is due to several reasons. First, speculative execution and dynamic memory disambiguation increase the movement freedom of load instructions. The load instructions can often be executed in empty cycles created by other long latency computations. Therefore a more efficient schedule of the loop is obtained. Second, loop unrolling provides more instructions to schedule within the loop body. In this manner, a more compact schedule for the loop can be obtained.

A side effect of register preloading is the increased physical register requirement for a program. Additional temporary registers are added with loop unrolling and dynamic memory disambiguation. Register live ranges are increased by preloading in previous loop iterations and above conditional branches. The maximal register usage across issue rates and load latencies with varying levels of register preload support is quantified in Figure 5.

For level 1 support, address calculation can be performed early with the compiler transformations. Thus, the life time of the integer registers that hold effective addresses are extended, which results in a significant increase in integer register usage. The limited increase in floating point register usage reflects the fact that some loads cannot be moved due to the lack of speculative execution support. This lim-

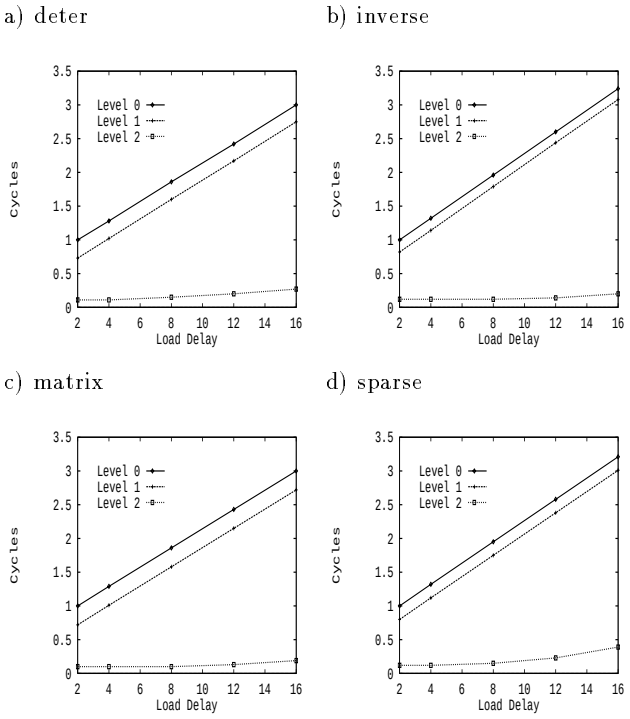


Figure 4: Normalized execution time for issue 8.

itation is overcome by the level 2 support, which allows many more loads to be performed early. While the performance improvement is impressive as shown in Figures 2-4, the cost is the increase usage of floating point registers to serve as the destination of these loads. For the benchmarks, as many as 52 floating point registers are needed to effectively tolerate up to 16 cycles of first level memory access latency.

CONCLUSION

Register preloading is a combination of hardware and compiler techniques to create opportunities to overlap the access of the first level memory with useful computation. Speculative execution removes control dependences between load instructions and preceding branch instructions. Dynamic memory disambiguation removes memory dependences between memory accesses in the presence of inconclusive dependence analysis. A larger number of potential independent instructions is provided in a loop body with loop unrolling. Strip-mining is used to create sequential inner loops from parallel DOALL and DOACROSS loops so that these inner loops can be further transformed with register preloading. The combination of these techniques enables each node processor to execute useful instructions between a memory load and the instructions that use the fetched data. As a result, systems are able to effectively tolerate large first level memory access latency.

The performance of systems with varying node architecture configurations is evaluated with several degrees of register preloading support. Without any support for register preloading, the performance of a node processor suffers

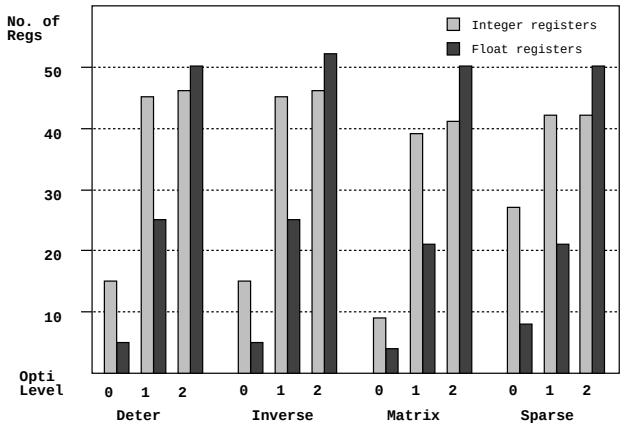


Figure 5: Maximum register usage across register preload support levels.

dramatically as the memory latency is increased. Larger performance losses are observed in systems with node processors that can execute more instructions each cycle. With only compiler support, the performance of a node processor is increased, however the ability to tolerate high memory latency is not significantly affected. With both compiler and architectural support for preloading, the performance of all node processor architectures is relatively constant across all load latencies evaluated.

In order to improve performance, future parallel systems will continue to increase the processing power of each node in a system. As node processors, though, can execute more instructions concurrently, they become more sensitive to the first level memory access latency. As shown in this paper, increases in the memory latency can result in significant performance losses. Register preloading can be effective in hiding moderate latencies for the architectures evaluated. However, parallel systems based on more powerful node processors will require additional methods to tolerate much longer first level memory access latency. Currently, we are investigating more sophisticated approaches to this problem.

ACKNOWLEDGEMENTS

The authors would like to acknowledge Nancy Warter and all members of the IMPACT research group for their comments and suggestions. Special thanks to the anonymous referees whose comments and suggestions helped to improve the quality of this paper significantly. This research has been supported by the Joint Services Engineering Programs (JSEP) under Contract N00014-90-J-1270, Dr. Lee Hoevel at NCR, the AMD 29K Advanced Processor Development Division, Matsushita Electric Industrial Corporation Ltd., Hewlett-Packard, and the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS).

REFERENCES

[1] Alliant Computers Systems Corporation, *Alliant*

FX/C-2800 Programmer's Handbook, 1990.

[2] G. Hodson, "Touchstone delta: Supercomputing at 32 gigaflops," *Microcomputer Solutions*, pp. 16–17, Sept. 1991.

[3] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proc. 18th Ann. Int'l Symp. Computer Architecture*, (Toronto, Canada), pp. 266–275, June 1991.

[4] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," in *Proc. Second Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems.*, (Palo Alto, CA), pp. 180–192, Oct. 1987.

[5] B. R. Rau, D. W. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, pp. 12–35, Jan. 1989.

[6] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum, "Compiler-directed data prefetching in multiprocessor with memory hierarchies," in *Proc. Int'l Conf. on Supercomputing*, (Amsterdam, The Netherlands), pp. 354–368, June 1990.

[7] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *Proc. Fourth Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems.*, pp. 40–52, Apr. 1991.

[8] T. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors," *J. Parallel and Distributed Computing*, vol. 12, pp. 87–106, 1991.

[9] M. Wolfe, "Iteration space tiling for memory hierarchies," in *Proc. of the 4th SIAM Conference*, 1989.

[10] D. Gannon, W. Jalby, and K. Gallivan, "Strategies for cache and local memory management by global program transformation," *J. Parallel and Distributed Computing*, vol. 5, pp. 344–358, 1988.

[11] D. Callahan, S. Carr, and K. Kennedy, "Improving register allocation for subscripted variables," in *Proc. ACM SIGPLAN '90 Symp. Compiler Construction*, pp. 53–65, 1990.

[12] E. D. Granston and A. V. Veidenbaum, "Detecting redundant accesses to array data," in *Proceeding of Supercomputing '91*, pp. 854–865, Nov. 1991.

[13] A. Nicolau, "Run-time disambiguation: coping with statically unpredictable dependencies," *IEEE Trans. Computers*, vol. 38, pp. 663–678, May 1989.

[14] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pp. 177–189, January 1983.

[15] Kuck & Associates, Inc., *KAP User's Guide*. Champaign, IL., Nov. 1988.

[16] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. . Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orzag, F. Seidl, O. J. nson, G. Swanson, R. Goodrum, and J. Martin, "The PERFECT club benchmarks: Effective performance evaluation of supercomputers," Tech. Rep. CSRD-827, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, May 1989.

[17] D. A. Padua, *Multiprocessors: Discussion of Some Theoretical and Practical Problems*. PhD thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, Nov. 1979. Center for Supercomputing Research and Development Report No. UIUCDCS-R-79-990.

[18] G. Kane, *MIPS RISC Architecture*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

[19] R. M. Russell, "The cray-1 computer system," *Communications of the ACM*, vol. 21, pp. 63–72, Jan. 1978.