Improving Instruction-level Parallelism by Loop Unrolling and Dynamic Memory Disambiguation

JACK W. DAVIDSON and SANJAY JINTURKAR
{jwd,sj3e}@virginia.edu
Department of Computer Science, Thornton Hall
University of Virginia
Charlottesville, VA 22903 U. S. A.

Abstract

Exploitation of instruction-level parallelism is an effective mechanism for improving the performance of modern super-scalar/VLIW processors. Various software techniques can be applied to increase instruction-level parallelism. This paper describes and evaluates a software technique, dynamic memory disambiguation, that permits loops containing loads and stores to be scheduled more aggressively, thereby exposing more instruction-level parallelism. The results of our evaluation show that when dynamic memory disambiguation is applied in conjunction with loop unrolling, register renaming, and static memory disambiguation, the ILP of memory-intensive benchmarks can be increased by as much as 300 percent over loops where dynamic memory disambiguation is not performed. Our measurements also indicate that for the programs that benefit the most from these optimizations, the register usage does not exceed the number of registers on most high-performance processors.

Keywords: loop unrolling, dynamic memory disambiguation, instruction-level parallelism, register renaming.

1: Introduction

Modern high-performance processors include a variety of hardware mechanisms to support the overlapped execution of independent instructions. Complicated instruction pipelines, multiple data paths, and multiple functional units are a few examples of such mechanisms. The potential to overlap execution of instructions is often referred to as instruction-level parallelism or ILP. As hardware mechanisms for exploiting ILP have become more prevalent, software techniques for increasing the available ILP in programs have become increasingly important.

One such code improvement technique is *loop unrolling* (LU) [9, 18] which in conjunction with *register renaming* (RR) [3, 14] can increase ILP. LU replicates the original loop body multiple times, adjusts the loop termination code and eliminates redundant branch instructions. The resulting larger basic block increases the probability that the instruction scheduler can reorder instructions to exploit ILP. However, the scheduler's effectiveness is limited by artificial dependencies created by LU's naive reuse of registers and other data dependencies between instructions.

Application of RR can eliminate the artificial dependencies. The resulting loop has more ILP exposed than the original, rolled loop. The determination of data dependencies requires some analysis by the compiler. Dependencies involving registers can be determined by symbolic comparison, which is a relatively simple process. But dependencies which involve memory references are not easy to resolve. Two memory references, which are symbolically dissimilar, may still access the same memory location. On the other hand, two memory references, which are symbolically the same, may access different memory locations. Determining whether the two memory references access the same memory location or not is known as the *aliasing* problem.

In the absence of precise information, a compiler must assume that all the memory references are aliases for the same memory location. Such a conservative approach limits the compiler's ability to reorganize the instructions in the program to increase ILP. In this paper, we discuss a technique, called *dynamic memory disambiguation* (DMD), which disambiguates memory references in loops at execution time. Our research indicates that when DMD is used in conjunction with LU and RR, ILP in benchmark loops can be increased by as much as three times.

This research is a continuation of our efforts to evaluate the effectiveness of techniques which determine critical pieces of information regarding data alignment and aliasing at execution time. This technique has proven effective at reducing the memory bandwidth requirements of memory-intensive programs [2, 8]. We extend this approach to enhance the exploitation of ILP in loops. Note that the application of DMD allows the exploitation of ILP even when there are multiple call sites and aliasing exists at some but not all of the call sites. Common situations such as these are difficult for interprocedural analysis to handle.

The following are definitions of some terms frequently used throughout this paper.

Unrolled loop: A loop unrolled *n* times consists of (*n + 1*) versions of the loop body of the original rolled loop.

Aggressive loop: A unrolled loop where potential aliasing of memory references is ignored by the instruction scheduler.

Safe loop: An unrolled loop where potential aliasing of memory references is not ignored by the instruction scheduler.

2: Related Work

There are a number of compiler techniques for increasing the ILP in a program. One such technique is LU. Weiss discusses LU from the perspective of automatic scheduling by the compiler [18]. This study also evaluates the effect of LU on instruction buffer size and register pressure within the loop for Livermore loops [15].

Register renaming is used to eliminate artificial dependencies. Kuck discusses techniques such as scalar expansion and variable renaming that can eliminate anti and output dependencies [13]. Techniques to eliminate dependencies were implemented in the Bulldog and Cydra-5 compilers [16]. Mahlke discusses the effect on performance of renaming registers in an unrolled loop [14]. To minimize conflicts and increase ILP, all register uses in the unrolled loop are assigned unique registers. In our approach, we rename registers only if it will lead to an improved instruction schedule.

Bernstein, Cohen, and Maydan evaluate the effect of DMD on software pipelining, loop invariant code motion, and redundant load elimination [6]. In their approach, the difference between the array reference expressions of the memory references which are to be disambiguated is computed. If the absolute value of the difference is greater than the data access size, then the references are not aliased. The resultant expressions are inserted as checks. Another approach by Huang proposes that the cost of dynamic disambiguation be hidden using speculative instructions and predicted execution [11].

In this paper, we evaluate the impact of DMD on ILP, when it is applied in conjunction with LU, RR and SMD. Unlike earlier approaches, we apply DMD to array references in unrolled loops only, so that benefits are maximized. To minimize the cost of conditional branches inserted by DMD, all the checks generated by it are inserted outside the unrolled innermost loop. The construction of checks is done after all the traditional optimizations have been performed on the code, so that the optimizer has a better idea of the probable benefits achieved by the application of DMD. Our results indicate that when DMD is applied in conjunction with LU, RR and SMD, the ILP of memory-intensive benchmarks can be increased by as much as 300 percent over loops where DMD is not performed.

3: Basic Issues

To illustrate the basic issues involved in aggressively exposing ILP in loops, we present an example using a simple, hypothetical machine. On this machine, the latency of a memory load and a conditional branch is two cycles. All other instructions have a latency of one cycle. The cycle width of the machine is two. The example is presented using register transfer lists (RTLs) to describe instructions [4, 5]. In the examples, `M[addr]` denotes a memory reference, while `r[n]` is a register reference.

The following C code adds the contents of array `b` to array `a`.

```
for (i = 0; i < n; i++)
   a[i] += b[i];
```

The addresses of the memory references are parameters to the function containing the loop. The machine instructions for the above code are given below. Each iteration of this loop takes six cycles to execute.

```
// r[11]:address of a, r[10]: address of b
// r[4]: address of a + (n * 4)
L16:
    r[2]=M[r[11]];r[3]=M[r[10]];
    r[10]=r[10]+4;nop;
    r[2]=r[2]+r[3];nop;
    M[r[11]]=r[2];r[11]=r[11]+4;
    PC=r[11]<r[4]->L16;nop;
    nop;nop
```

When the loop is unrolled once, the following code is obtained.

```
// r[11]:address of a, r[10]: address of b
// r[4]: address of a + (n * 4)
L16:
    r[2]=M[r[11]];r[3]=M[r[10]];
    nop;nop
    r[2]=r[2]+r[3];nop;
    M[r[11]]=r[2];nop;
```

```
    r[2]=M[r[11]+4];r[3]=M[r[10]+4];
    r[10]=r[10]+8;nop;
    r[2]=r[2]+r[3];nop;
    M[r[11]+4]=r[2];r[11]=r[11]+8;
    PC=r[11]<r[4]->L16;nop;
    nop;nop;
```

Here, each iteration takes 5 cycles. This is a 20 percent performance increase over the rolled loop. Now RR and static memory disambiguation (SMD) can be applied to the loop. SMD can determine if memory references `M[r[11]]` and `M[r[11]+4]` are aliases for the same memory location. Compile-time analysis indicates that they are not aliases. This is because symbolic comparison shows that the memory locations accessed by these two references are separated by a distance of four bytes. This indicates that scheduling the load `M[r[11]+4]` before the store `M[r[11]]`will not change the semantics of the code. Using this information, the scheduler produces the following code.

```
L16:
    r[2]=M[r[11]];r[3]=M[r[10]];
    r[6]=M[r[11]+4];nop;
    r[2]=r[2]+r[3];nop;
    M[r[11]]=r[2];nop;
    r[7]=M[r[10]+4];r[10]=r[10]+8;
    nop;nop;
    r[6]=r[6]+r[7];nop;
    M[r[11]+4]=r[6];r[11]=r[11]+8;
    PC=r[11]<r[4]->L16;nop;
    nop;nop;
```

This code also requires 5 cycles per iteration of the loop. Thus, there a no improvement. A closer examination indicates that the instruction schedule can be improved further if the load of `M[r[10]+4]`is scheduled before the store of `M[r[11]]`. However, that is not possible since the load of `M[r[10]+4]` may be an alias for the store of `M[r[11]]`. Since the contents of the registers `r[10]` and `r[11]` are parameters to the function enclosing the loop, the relationship between the contents of the two registers cannot be determined by intra-procedural analysis.

To resolve this problem, DMD is applied. DMD generates a new copy of the loop called the aggressive loop. In the aggressive loop, the scheduler ignores the potential aliasing between the memory references `M[r[10]+4]`and `M[r[11]]`. Consequently, the scheduler is able to place the load of `M[r[10]+4]` before the store of `M[r[11]]`. In the safe copy, the code remains the same as that after the application of SMD. The compiler inserts checks to select the appropriate copy at run time. Code to select the appropriate loop to execute and the two loops are shown below.

```
// r[11]:address of a, r[10]: address of b
// r[12]: n * 4, r[4]: address of a + (n * 4)
// check if a + n < b
r[13]=r[11]+r[12];nop;
PC=r[13]<r[10]->L16;nop
nop;nop;
// check if b + n >= a
r[13]=r[10]+r[12];
PC=r[13]>=r[11]->L24;nop;
nop;nop;
// L16 begins aggressive loop
L16:
    r[2]=M[r[11]];r[3]=M[r[10]];
    r[6]=M[r[11]+4];r[7]=M[r[10]+4];
    r[2]=r[2]+r[3];r[10]=r[10]+8;
    M[r[11]]=r[2];r[6]=r[6]+r[7];
    M[r[11]+4]=r[6];r[11]=r[11]+8;
    PC=r[11]<r[4]->L16;nop;
    nop;nop
    ...
// L24 begins the safe loop
L24:
```

At execution time, if the aggressive copy of the loop is executed, then 3.5 cycles per iteration are required, which is a 70 percent increase over the original code. Thus DMD, in conjunction with LU and RR can significantly increase ILP.

4: Algorithms and Implementation

In this section, we discuss the issues involved in implementing automatic LU, RR, and DMD. In this paper, only the high-level algorithm to perform DMD is presented. Algorithms to perform LU, RR and instruction scheduling are presented in other reports [9, 10].

A portion of the high-level algorithm to implement LU, RR and DMD is contained in Figure 1. RR and DMD are applied late in the optimization process, because they are applied to unrolled loops only, and LU is applied after all the traditional optimizations have been performed on the program. After a loop is unrolled, DMD is applied to determine if there are any memory references in the loop that are aliases for the same memory location. Figure 2 contains the DMD algorithm. First, the number of load and store partitions are computed. This determines the number of checks required to perform DMD. To insert the alias checks, the minimum and the maximum addresses accessed by the memory references in each partition are needed. Depending on the stride of each partition, the address expression to compute the minimum (maximum) address is trivially available. The minimum (maximum) address expression, along with stride and iteration count of the loop, can be used to construct the address expression to compute the maximum (minimum) address. Once the address expressions have been calculated, *InsertAliasChecks,* is called to insert the checks. At run time, each check compares the minimum and the maximum

```
1   // Main routine to implement LU and DMD
2   proc UnrollAndDisambiguate(CurrFunction) is
3       // Consider each loop in the current function.
4       ∀LOOP ∈ CurrFunction.Loop do
5           LOOP.InductionVars ← FindInductionVars(LOOP)
6           // Classifies memory references into different partitions if a unique identifier is found to distinguish
7           // a set of such references. For example, all references to an array A passed as a parameter will have a loop
8           // invariant register as their partition identifier.
9           ClassifyMemoryReferencesIntoPartitions(LOOP)
10          // Calculate relative offsets of the memory references belonging to same partition from the induction variable.
11          CalculateRelativeOffsets(LOOP)
12          // Unroll the loop if it fits in the cache.
13          CurrFunction.Loop ← {UnRollLoopIfProfitable(LOOP)} ∪ CurrFunction.Loop
14          EliminateInductionVariables(LOOP)
15          // Do alias analysis and apply DMD if necessary. Indicate the upper limit for the number of partitions, for
16          // which disambiguation should be done. If there are more than the limit, then DMD is not done.
17          DynamicMemoryDisambiguation(LOOP, NoOfStorePart, NoOfTotalPart)
18          // Do instruction scheduling. Apply register renaming if necessary.
19          DoInstructionScheduling(LOOP, CycleWidth)
20          enddo
21      endproc
```

Figure 1: Main loop to perform Loop Unrolling and Dynamic Memory Disambiguation.

```
1   proc DynamicMemoryDisambiguation(LOOP, S, T) is .
2       if (LOOP.Unrolled)) then
3           StorePart ← ComputeTotalPartitions(LOOP, StorePart, LoadPart)
4           // There should be at least one Load partition
5           if (((LoadPart + StorePart) > 0) ∧ ((LoadPart+StorePart) <= T) ∧ (StorePart <= S) ∧ (LoadPart > 0)) then
6               // Insert iteration count of the loop into the preheader of the loop. Store the iterations in register IterReg.
7               InsertCode(LOOP, LOOP.iterations, IterReg)
8               PartSet1 ← LOOP.Partitions
9               PartSet2 ← ∅
10              ∀ P ∈ LOOP.Partitions.StorePart do
11                  ∀ Q ∈ (LOOP.Partitions.LoadPart ∨ LOOP.Partitions.StorePart) do
12                      // If the partition identifiers for P and Q are same, then SMD can be applied. Otherwise, DMD has
13                      // to be applied. To minimize the number of checks which need to be inserted, we consider the addresses
14                      // of all the memory references in partitions P and Q. Thus we consider all Stores in P and
15                      // all Loads(Stores) in Q.
16                      If ((P.Ident ≠ Q.Ident) ∧ (¬(ChecksInserted(P.Ident, Q.ident))))
17                          InsertAliasChecks(P, Q, LOOP.Partitions)
18                          endif
19                      endfor
20                  endfor
21              endif
22          endif
23      endproc
```

Figure 2: Routine to perform Dynamic Memory Disambiguation

addresses accessed by memory references in one partition containing stores with the maximum and minimum addresses accessed by memory references in a second partition. The second partition may contain either loads or stores. For instance, if there is one partition with only stores, a second partition with only loads, and a third partition with both loads and stores, then there will be a set of three checks. Note that whether memory references within a partition are aliases for the same location or not can be determined by the application of SMD. The maximum number of checks which can be inserted is a parameter to the algorithm.

After DMD has been applied, instruction scheduling is performed. During scheduling, if the size of the ready set is less than the cycle width (i.e., the maximum number of instructions which can be issued in a cycle), then, an attempt to rename registers is made. The process renames registers to eliminate anti- and output data dependencies.

During the renaming process, the routine ensures that precise information about memory reference aliasing is available. If the routine determines that there are potential aliasing problems for memory reference and the copy of the loop is the safe copy, then it does not perform any RR. But if the copy of the loop is an aggressive copy, then it does perform RR which permits the code to be scheduled better. Thus, RR is performed only if it will be useful. Our approach, while delivering benefits, does not increase the register utilization unnecessarily.

5: Experimental Results

5.1: Framework

We have implemented the above algorithms in the portable C compiler *vpcc-vpo* [4, 5]. The compiler was retargeted to a hypothetical VLIW machine. The compiler employs the same instruction set as that of the MIPS R4000 [12] architecture family with the instruction latencies given in Table 1. The latency of the instructions are comparable to those on current high performance superscalar/VLIW processors. In addition, the machine has unlimited supply of all functional units except the branch unit. There is only one branch unit available. The register allocator and the instruction scheduler have 25 integer registers and 15 floating-point registers available to them. This excludes the registers reserved for the assembler, any special purpose registers, and registers required for stack frame maintenance. Not constraining the number of functional units allows us to completely exploit the ILP exposed by the compiler transformations.

Instruction	Latency	Instruction	Latency
Memory load	3	Single prec. ALU	4
Memory store	1	Double prec. ALU	4
Integer ALU	1	Single prec. mul	5
Integer mul	5	Double prec mul.	6
Integer div	6	Single prec div.	7
Cond. Branch	2 (delay slot executed)	Double prec. div.	10

Table 1: Instruction Latencies

In this study, we concentrate on the measurement of ILP. ILP is measured using the formula proposed by Wall [17].

ILP = Total latency / Total cycles

We chose this measure because it gives an idea of the resources required to fully exploit the available parallelism in a piece of code. Unlike speedup, which is a relative measurement, this formula states the parallelism independent of any baseline measurement.

	Benchmark	Description
SYNTHETIC	arraymerge	Merges two sorted arrays
	bubblesort, quicksort, shellsort	Sorting algorithm
	puzzle	Benchmark to test recursion
	queens	The eight queens problem
	sieve	Sieve of eratosthenes
USER	cache	Cache simulation
	encode	Stores encoded vpo's RTL files
	sa-tsp	Travelling salesperson problem
NUMERICAL	add	Array addition
	copy	Array copy
	linpack	Floating-point benchmark
	ll1, ll5, ll11, ll12	Livermore kernels 1, 5, 11, 12
	s152, s176, s254	Loop kernels from Callahan-Dongarra-Levine test suite
UNIX UTILITIES	cal	Prints out a calender
	diff	Prints out diff. between two files
	grep	Searches for a string in a file
	nroff	A standard document formatter
	od	Prints out the octal dump of a file
	sort	Sorting utility

Table 2: Description of benchmarks

To measure ILP, we used the architecture measurement tool *Ease* to instrument the code and measure the dynamic instruction counts and latency of the code [7]. In all the experiments, the unroll factor was three and the cycle width was four unless otherwise stated. The measurements reported in this study were performed on the set of benchmarks listed in Table 2. The benchmarks are divided into four categories according to the nature of the benchmark

5.2: Results

In this section, we present the results of our study. The study was conducted in three parts. In the first part, we measured the effect of LU, RR, SMD and DMD on loops in each benchmark. In the second part, we measured the register usage for numerical benchmarks when LU, RR, SMD and DMD are successively applied. In the third part, we investigated the effect of unroll factor on ILP.

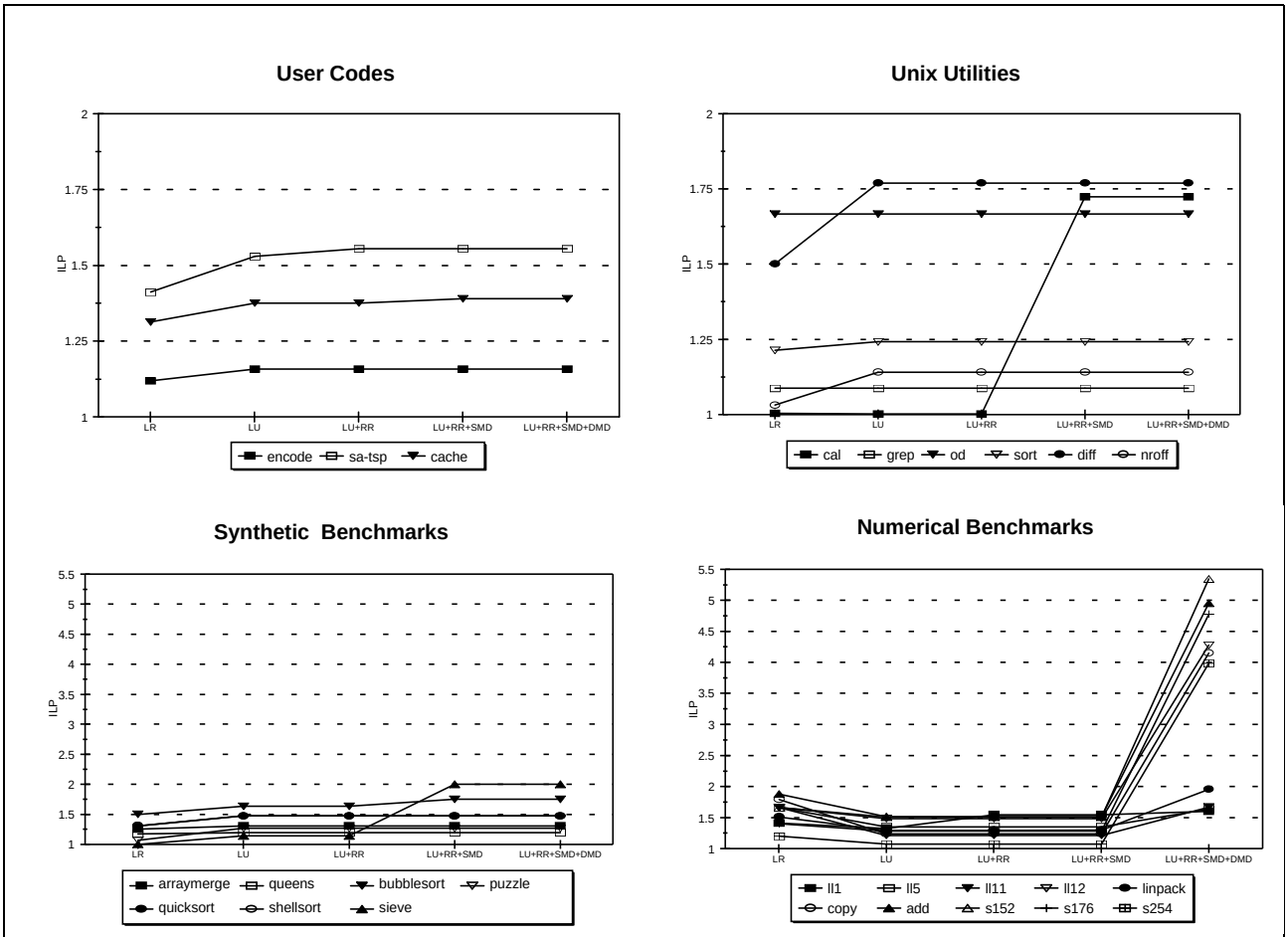

Figure 3: ILP of loops in various benchmarks in each category on the hypothetical VLIW machine.

ILP of loops in benchmarks: For each benchmark, we present the combined ILP for all the innermost counting loops in each benchmark. To calculate the combined ILP, we sum the cycle and latency counts of all the innermost counting loops and apply Wall's formula.

Figure 3 contains graphs† which show the combined ILP of all the innermost counting loops for various benchmarks. For user codes, all the benchmarks benefit from unrolling loops. The application of RR and SMD improves ILP for the benchmarks *cache* and *sa-tsp*, but by a small amount. The loops in these benchmarks contain multiple basic blocks and function calls. Since it is not safe to reorder memory references across function calls in absence of inter-procedural analysis, aggressive scheduling cannot be done and therefore, the benefits accrued are limited. For Unix utilities, the benchmarks *sort*, *diff* and *nroff* benefit from LU. However, only *cal* improves due to SMD. This benchmark has a loop which contains writes. By applying SMD, these writes are done in parallel resulting in a significant increase in ILP. For the synthetic benchmarks, the application of SMD improves the ILP of benchmarks *bubblesort* and *sieve*. In *bubblesort*, the application of RR in conjunction with SMD allows parallel execution of multiple high-latency memory loads. In *sieve*, SMD allows the parallel execution of multiple writes to the memory. For the numerical benchmarks, the application of LU to rolled loops decreases ILP because the compiler replaces multiple increments of the induction variable by a single increment. Application of RR in conjunction with SMD marginally improves the ILP. But when DMD is also applied, the ILP increases significantly to as high as 5.2. In all these benchmarks, the contents of arrays are being

†LR refers to rolled loops.

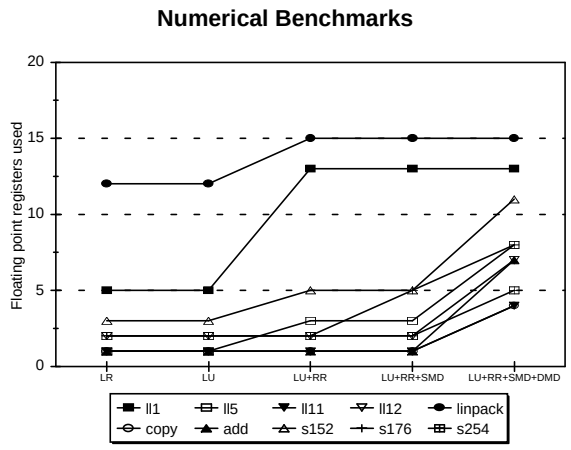

Figure 4: Floating-point register usage for Numerical Benchmarks on the hypothetical VLIW machine.

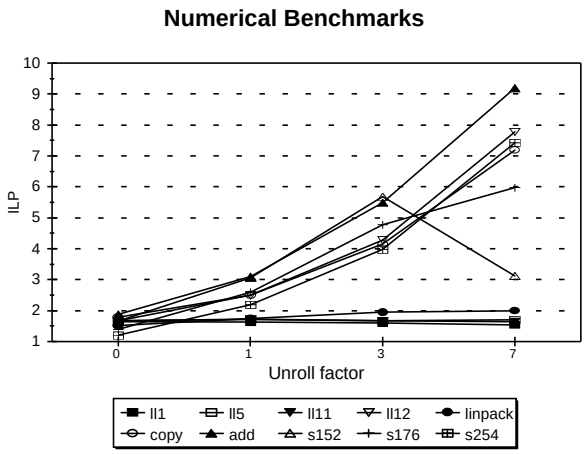

Figure 5: Effect of Unroll Factor on loops in Numerical Benchmarks on the hypothetical VLIW machine.

modified. The addresses for these arrays are being passed as parameters to the function. In absence of precise information about aliasing between the memory references, no reordering of the load instructions is possible. But when DMD is applied, the loads in these benchmarks are reordered, which allows parallel execution. As a side effect of RR, the common subexpression eliminator (CSE) [1] does a better job and is able to eliminate multiple loads from the same location in an unrolled loop, which, originally belonged to the different iterations of the rolled loop. This occurs in the benchmark *ll12*.

Register usage: In this section, the register usage when LU, RR, SMD and DMD are successively applied to the numerical benchmarks is presented. All the numerical benchmarks operate on floating-point numbers. The integer register usage increases marginally when DMD is applied because integer registers are needed to compute the addresses of the memory references so that checks can be inserted that determine whether the safe or aggressive loop is to be executed. The increase, however, is minimal, and is not shown here, but can be found in another report [10].

Figure 4 shows the usage of floating-point registers. The application of DMD increases the usage of registers for almost all the benchmarks. This is because the application of DMD facilitates RR in these benchmarks, which in turn enables the scheduling of high latency load instructions in parallel. From Figure 4, it is apparent that 15 registers are enough to support the application of LU, RR, and DMD for an unroll factor of three. While higher unroll factors could increase register usage, typical RISC machines have at least 32 floating-point registers.

Effect of the unroll factor: In this section, the effects of the unroll factor on the ILP of loops is presented. We measured the effect of unroll factors of 0, 1, 3 and 7 on the ILP of the loops in numerical benchmarks, the results of which are shown in Figure 5. Other results are available in the detailed technical report [10]. The cycle width is kept constant at 8, so that only the effect of changing the unroll factor is measured. Also, RR, SMD and DMD have been applied in each case.

Ideally, increasing the unroll factor should increase the ILP, but that is not always the case. Lack of registers causes a decrease in ILP for benchmark *s152* when the unroll factor increases from 3 to 7. In this benchmark, there are not enough registers to perform register renaming. Consequently, a number of loads are executed in a sequential fashion, rather than being executed in parallel. On the other hand, all other benchmarks in the category take full advantage of the available loads and schedule them in parallel, which increases ILP sharply. From this figure, it is apparent that increasing the unroll factor from 3 to 7 increases the ILP of all but one numerical benchmarks perceptibly. The number of registers available are sufficient for most numerical benchmarks to allow the application of RR and DMD.

6: Summary

With increasing frequency, emerging high-performance processors include mechanisms for executing independent instructions in parallel. The effectiveness of these features depends, to a large extent, on the amount of instruction-level parallelism in a program. This paper has described and evaluated a software technique, dynamic memory disambiguation, that permits loops containing write memory references to be scheduled more aggressively, thereby exposing more instruction-level parallelism. Our measurements show that when DMD is applied in conjunction with loop unrolling, register renaming and static memory disambiguation, the ILP of memory-intensive benchmarks can be increased by as much as 300 percent over loops where only loop unrolling, register renaming, and static memory disambiguation has been performed. Like many other optimizations, loop unrolling, register renaming, and dynamic memory disambiguation use register resources. Our measurements also indicate that for the programs that benefit the most from these optimizations, the register usage does not increase appreciably and does not exceed the number of registers found on most high-performance processors. We conclude that dynamic memory disambiguation can be a valuable and viable transformation that can significantly enhance the instruction-level parallelism in loops where compile-time analysis cannot determine if there is any aliasing.

7: Acknowledgements

This work was supported in part by National Science Foundation grants CCR-9214904 and MIP-9307626. We would also like to thank Mark Bailey and Bruce Childers for their input and feedback.

References

[1]] Aho A., Sethi, R., and Ullman, J. D., *Compilers Principles, Techniques and Tools*, Addison-Wesley, Reading, MA, 1986.

[2] Alexander, M. J., Bailey, M. W., Childers, B. R., Davidson, J. W., and Jinturkar, S., "Memory Bandwidth Optimizations for Wide-Bus Machines", *Proceedings of the 25th Hawaii International Conference on System Sciences*, Mauii, HA, January 1993, pp. 466-475.

[3] Bacon, D. F., Graham, S. L., and Sharp, O. J., "Compiler Transformations for High-Performance Computing", *ACM Computing Surveys*, **26**(4), Dec. 1994, pp. 345-420.

[4] Benitez, M. E. and Davidson, J. W., "The Advantages of Machine-Dependant Global Optimization", *Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, GA, June 1988, pp. 329-338.

[5] Benitez, M. E., "Register Allocation and Phase Interactions in Retargetable Optimizing Compilers", PhD Dissertation, University of Virginia, Charlottesville, April 1994.

[6] Bernstein, D., D. Cohen, D. E. Maydan, "Dynamic Memory Disambiguation for Array References", Proceedings of the 27th Annual International Symposium on Microarchitecture, San Jose, CA, Dec. 1994, pp. 105-112.

[7] Davidson, J. W. and Whalley, D. B., "Ease: An Environment for Architecture Study and Experimentation", *Proceedings of the 1990 ACM Sigmetrics Conference on Measurement and Modelling of Computer Systems*, Boulder, CO, May 1990, pp. 259-260.

[8] Davidson, J. W. and Jinturkar, S., "Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses", *Proceedings of SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994, pp 186-195.

[9] Davidson, J. W. and Jinturkar, S., "An Aggressive Approach to Loop Unrolling", Technical Report CS-95-26, Department of Computer Science, University of Virginia, Charlottesville, June 1995.

[10] Davidson, J. W. and Jinturkar, S., "Improving Instruction-level Parallelism by Loop Unrolling and Dynamic memory Disambiguation", Technical Report CS-95-13, Department of Computer Science, University of Virginia, Charlottesville, February 1995.

[11] Huang, A., Slavenburg, G., and Shen J., "Speculative disambiguation: A Compilation Technique for Dynamic Memory Disambiguation", Proceedings of the 21st International Symposium on Computer Architecture, Chicago, IL, April 1994, pp 200-210.

[12] Kane, G., "MIPS RISC Architecture", Prentice-Hall, Englewood Cliffs, NJ, 1992.

[13] Kuck, D. J., R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations", *Proceedings of the ACM SIGPLAN '81 Symposium on Principles of Programming Languages*, Jan 1981, pp. 207-281.

[14] Mahlke, S. A., Chen, W. Y., Gyllenhaal, J. C. and Hwu, W. W., "Compiler Code Transformations for Superscalar-Based High-Performance Systems", *Proceedings of Supercomputing '92*, Portland, OR, Nov. 1992, pp. 808-817.

[15] McMohan, F. H, *The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range*, Lawrence Livermore National Laboratory, Livermore, CA, 1986.

[16] Rau, B. R., Yen, D. W. L., and Towle, R. A., "The Cydra Departmental Supercomputer", *IEEE Computer*, January 1989, pp. 12-35.

[17] Wall, D. W., "Limits of Instruction-Level Parallelism", *WRL Research Report*, 93/6, Digital Equipment Corporation, Palo Alto, CA, 1993.

[18] Weiss, S,. and Smith, J. E., "A Study of Scalar Compilation Techniques for Pipelined Supercomputers", *Proceedings of Second International Conference on Architectural Support for Programming Languages and Operating Systems*", Palo Alto, CA, Oct. 1987, pp. 105-109.