Paul J. Conrad
CS 610, Winter 2004
Dr. Keith Schubert

Unrolling Loops For Scheduling On a Two Processor Machine

Unrolling of loops within a computer program is perhaps one of the most important forms of code optimizations that the program code can benefit from. In this article, we will take a look at a function called **daxpy** (**d**ouble **p**recision of **a** times **x[I]** plus **y[I]**), which is used quite frequently in soling systems of linear equations. In figure 1, we have a MIPS32 code that is not in the best of forms. This code is equivalent in C/C++ style code as:

```
for (int i=0; i<n; i++ ) {
        y[i] = a * x[i] + y[i];
}
```

Through out this tutorial, the register $s1 contains the base address for y[], and the register $s2 is the base address for x[]. We have register $t1 holding the constant value of a, and register $t2 contains the end address of the array y[].

```
L1:   LD    $r1, 4[$s1]     // Load low word of y[i] into $r1                S1
      LD    $r2, 0[$s1]     // Load high word of y[i] into $r2               S2
      ADD   $s1, $s1, #8    // Increment y[] index by 8 bytes                S3
      MTHI  $r2             // Move high word into HI of [HI:LO]             S4
      MTLO  $r1             // Move low word into LO of [HI:LO]              S5
      LD    $r3, 0[$s2]     // Load x[i] into $r3                            S6
      ADD   $s2, $s2, #4    // Increment x[] index by 4 bytes                S7
      LD    $r4, $t1 // Load constant a into $r4                            S8
      MADD  $r3, $r4 // Multiply $r3 by $r4 and add to [HI:LO]              S9
      MFLO  $r1             // Move LO of result to $r1                      S10
      MFHI  $r2             // Move HI of result to $r2                      S11
      ST    $r1, -4[$s1]    // Save $r1 where we orginially got it           S12
      ST    $r2, -8[$s1]    // Save $r2 where we originally got it           S13
      BNE   $s1, $t2, L1    // Branch if Not Equal To last index of y[]      S14
```

Figure 1. daxpy code example

The example code, assuming that each instruction takes one cycle to execute, takes 14 cycles or 7 cycles per word, for each iteration of the loop. At first glance this code looks rather tight knit and optimized to run well. It is not, because we have a WAR (**W**rite **A**fter **R**ead) dependencies between Statement 2 and Statement 3, Statement 6 and Statement 7, and Statement 8 and Statement 9. If we were to run this code as-is on a two processor machine, it would look like:

	Pipeline #1			Pipeline #2		Cycle #
L1:	LD	$r1, 4[$s1]		LD	$r2, 0[$s1]	1
	ADD	$s1, $s1, #8		*stall*		2
	MTHI	$r2		MTLO	$r1	3
	LD	$r3, 0[$s2]		ADD	$s2, $s2, #4	4
	stall			LD	$r4, $t1	5
	MADD	$r3, $r4	*stall*			6
	MFLO	$r1		MFHI	$r2	7
	ST	$r1, -4[$s1]		ST	$r2, -8[$s1]	8
	BNE	$s1, $t2, L1				9

Table 1. daxpy code on two processor pipeline

In table 1, the code now takes 9 cycles or 4.5 cycles per word for each iteration of the loop. This is a speedup of 35.7%, even though we have three points in the code that we stall out at. We can resolve these stalls and make the code better by unrolling the loop and reordering some of the instructions. In order to unroll the loop, we have to (1) make a copy of the loop body, (2) rename the registers in the new copy so we do not have a conflict with the original loop body registers (we do not want to add anymore dependencies or headaches). Our new loop unrolled with just one copy and RAW dependencies removed by instruction reordering looks like:

```
L1:   LD    $r1, 4[$s1]     // Load low word of y[i] into $r1                    S1
      LD    $r2, 0[$s1]     // Load high word of y[i] into $r2                   S2
      LD    $r5, 12[$s1]    // Load low word of y[i+1] into $r5                  S3
      LD    $r6, 8[$s1]     // Load high word of y[i+1] into $r6                 S4
      MTHI  $r2             // Move y[i] high word into HI of [HI:LO]            S5
      MTLO  $r1             // Move y[i] low word into LO of [HI:LO]             S6
      LD    $r3, 0[$s2]     // Load x[i] into $r3                                S7
      LD    $r4, $t1 // Load constant a into $r4                                S8
      LD    $r7, 4[$s2]     // Get x[i+1] and place into $r7                     S9
      MADD  $r3, $r4 // Multiply $r3 by $r4 and add to [HI:LO]                  S10
      MFLO  $r1             // Move LO of result to $r1                          S11
      MFHI  $r2             // Move HI of result to $r2                          S12
      MTHI  $r6             // Move y[i+1] high word into HI of [HI:LO]          S13
      MTLO  $r5             // Move y[i+1]low word into LO of [HI:LO]            S14
      ADD   $s1, $s1, #16   // Increment y[] index by 16 bytes                   S15
      MADD  $r7, $r4 // Multiply $r7 by $r4 and add to [HI:LO]                  S16
      ADD   $s2, $s2, #8    // Increment x[] index by 8 bytes                    S17
      MFLO  $r5             // Move LO of result to $r5                          S18
      MFHI  $r6             // Move HI of result to $r6                          S19
      ST    $r1, -12[$s1]   // Save $r1 where we orginially got it               S20
      ST    $r2, -16[$s1]   // Save $r2 where we originally got it               S21
      ST    $r5, -4[$s1]    // Save $r5 where we orginially got it               S22
      ST    $r6, -8[$s1]    // Save $r6 where we originally got it               S23
      BNE   $s1, $t2, L1    // Branch if Not Equal To last index of y[]          S24
```

Figure 2 daxpy code example loop unrolled once

In figure 2, the loop body now has two iterations of the loop (the original, plus an extra loop unrolled in it). The registers that have been renamed in the new copy of the unrolled loop are $r5, $r6, and $r7. The index registers are doubled in their increments (would be tripled if we add another loop iteration to the loop body). The code now has no data dependencies and can run in 24 cycles on a single processor with 6 cycles per word. In table 2, we have the code pipelined on a two processor pipeline and the results are excellent since we no longer have any stalls to be concerned with.

	Pipeline #1		Pipeline #2		Cycle #
L1:	LD	$r1, 4[$s1]	LD	$r2, 0[$s1]	1
	LD	$r5, 12[$s1]	LD	$r6, 8[$s1]	2
	MTHI	$r2	MTLO	$r1	3
	LD	$r3, 0[$s2]	LD	$r4, $t1	4
	LD	$r7, 4[$s2]	MADD	$r3, $r4	5
	MFLO	$r1	MFHI	$r2	6
	MTHI	$r6	MTLO	$r5	7
	ADD	$s1, $s1, #16	MADD	$r7, $r4	8
	ADD	$s2, $s2, #8	MFLO	$r5	9
	MFHI	$r6	ST	$r1, -12[$s1]	10
	ST	$r2, -16[$s1]	ST	$r5, -4[$s1]	11
	ST	$r6, -8[$s1]	BNE	$s1, $t2, L1	12

Table 2. unrolled daxpy code on two processor pipeline

The code in table 2, does not contain any stalls anywheres since there was careful consideration of reordering instructions that caused stalls in figure 1 and table 1. The programmer and/or the optimizing complier must ensure that the alogrithm and program-correctness is not jeopardized by instruction re-ordering. The instructions that were re-ordered were the loop index variables since they could be incremented wherever we felt like incrementing them. The store instructions can write back to the original locations via effective address calculations. The code, as shown in table 2, executes each iteration of the loop in 12 cycles or 3 cycles per word. According to Amdahl's Law, this is a 57.1% speedup of the code.