Loop Transformation Techniques To Aid In Loop Unrolling And Multithreading

Litong Song, Yuhua Zhang and Krishna Kavi
The University of North Texas

Abstract

In modern computer systems loops present a great deal of opportunities for increasing Instruction Level and Thread Level Parallelism. Loop unrolling is a technique used to obtain greater ILP while independent loop iterations are assigned to different threads to obtain greater TLP. However, techniques are needed to avoid unnecessary checks to assure that only the correct number of iterations are executed. In this paper we evaluate simple loop transformation techniques that can improve the performance by eliminating some unnecessary conditional instructions checking for iteration bounds. We present information on the number of instructions eliminated as well as on the improved branch prediction rates and execution performance improvements. Our techniques are applicable to most modern architecture including superscalar, multithreaded, VLIW or EPIC systems.

Key words. ILP, TLP, Loop Level Parallelism, Branch Prediction, Code Transformation.

1. Introduction

Instruction level parallelism (ILP) is needed to fully utilize the available functional units in Superscalar and VLIW architectures. Loops can provide such ILP, since loops can be unrolled to hide dependencies among instructions as well as stalls resulting form longer latencies encountered by some instructions. However, since the number of iterations is unknown at compile time, the unrolled loop must include test instructions to assure that a loop is executed correctly.

Thread level parallelism (TLP) is needed to fully utilize hardware contexts available in multithreaded architectures. Loop iterations can be allocated to different threads to facilitate higher levels of thread parallelism. The goal of such parallelization efforts is to maximize resource utilization, minimize overhead and data dependencies among threads. One technique to achieve these goals is to spawn a fixed number of threads at a time, and new threads are created only after previously spawned threads complete. This is the approach we take in our multithreaded system called Scheduled Dataflow (SDF) [1, 5-8].

The literature abounds with loop optimization techniques particularly for multiprocessor systems (see [4] for a survey of some such techniques). Loop unrolling techniques for modern computer architectures are well understood and reported in textbooks such as [3].

2. Multithreading and Thread Level Parallelism

A multithreaded system contains multiple "loci of control" (or threads) within a single program; the processor is shared by these multiple threads leading to higher utilization. A detailed survey of multithreading at programming level, system level and architecture level can be found in [9]. Lately, there is an increasing interest in providing hardware support for multithreading.

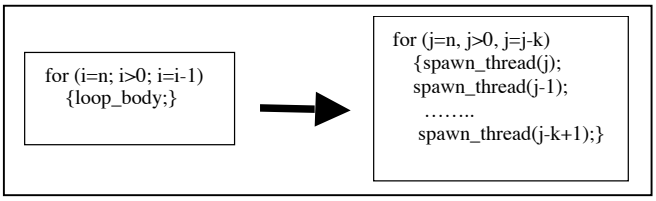
Our Scheduled Dataflow (SDF) [1, 5-8] architecture differs from other multithreaded architectures in two ways: i) our programming paradigm is based on non-blocking threads,, and ii) complete decoupling of all memory accesses from execution pipeline. Data is pre-loaded into an enabled thread's register context prior to its scheduling on the execution pipeline. After a thread completes execution, the results are post-stored from its registers into memory. We use two separate processing elements: the Synchronization Processor (SP) performs pre-load and post-store while the Execution Processor (EP) performs actual computations of a thread. EP does not access memory (operates only on data in registers) while SP does Load and Store to move data between memory and registers. Unlike Superscalar, our architecture performs no (dynamic) out-of-order execution and thus eliminates the need for complex instruction issue and retiring hardware.

Previously we reported results comparing our SDF architecture with superscalar and VLIW systems [1, 5-8]. We also reported how SDF scales better than other systems as we increase the number of functional units and the number of register contexts [1]. This behavior can also be observed from the data presented in this paper in later sections. In our research we are continuing to explore innovative compiler optimizations to improve the performance of SDF even further. In this paper, we present how loop level parallelism can be translated into thread level parallelism, while optimizing resource utilization and minimizing overhead due to thread creation and context switches. We will use our SDF to explore how a simple loop transformation can aid in these efforts.

2.1. SDF Threads for Loops.

In our system, we create threads using FALLOC instruction, which allocates a frame for the thread. The

created thread will receive its inputs in the frame and will be enabled only after receiving all the required inputs. Upon receiving all inputs, the thread is allocated a register set and its input data is transferred from the frame memory (and other data from I-structures) into its registers -- this phase is called the pre-load phase, and is executed by the SP. Then the thread is scheduled for execution on the EP; since all the data is in registers, EP will not access memory. The results of a thread (upon completion of execution) are stored in the frames of awaiting threads (or stored in shared I-structures) – the SP also performs the post-storing. Due to the non-blocking nature of SDF, our threads are very fine grained. When considering loops, we generate one or more threads for each iteration. However, spawning a thread requires the availability of a frame and a register set as described above. So we spawn only a limited number of threads corresponding to a few iterations of a loop at a time (depending on the number of hardware contexts). After the spawned threads complete, we create another set of threads, and repeat this process until all loop iterations are completed. This can be viewed as transforming a loop as shown below, assuming that k threads are spawned at a time to execute k loop iterations in parallel.

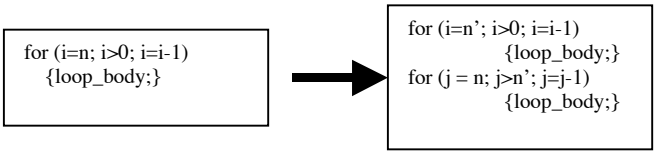


Such transformation works well if the loop is executed an even multiple of k times. If not, we need to use additional tests as shown below.

```
for (j=n, j>0, j=j-k){
   if (j≥k) {spawn_thread(j);… spawn_thread(j-k+1);}
   else if (j≥k-1) {spawn_thread(j);… spawn_thread(j-k+2);}
   else if ….
   else if (j≥2) {spawn_thread(j); spawn_thread(j-1);}
   else  {spawn_thread(j);}
    }
```

The non-blocking nature of SDF incurs overhead when servicing conditional branches, leading to substantial overheads in parallelizing loops. We can transform the original loop into two new loops as shown below, where n' is an even multiple of k.

Now we can spawn k threads at a time to parallelize the first loop and execute the second loop sequentially by creating only one thread for each of the remaining iterations. In general, we need to compute the upper and lower bounds for the two loops after transformation and these values depend on whether the original loop is counting up or counting down, and the step value. Note that this transformation is done at compile time and incurs no extra hardware.



2.2. Experimental Results.

We used matrix multiplication and a picture zooming code segment published in [10] to explore the improvements resulting from our loop transformation. For our experiments we spawned 5 threads at a time to execute 5 iterations of a loop in parallel (that is, k =5), and varied the number of times a loop is actually executed. Table 1 shows the data for matrix multiplication.

Table 1: Data For Matrix Multiplication (1SP-1EP)

Data Size		Execution Cycles	Reduction	Total Instructions	Reduction
53*53	old	4,867,345	25.5%	8,112,999	27.4%
	new	3,626,615		5,893,200	
104*104	old	31,505,289	39.9%	51,788,231	44.2%
	new	18,938,761		28,916,551	
151*151	old	77,057,483	27.2%	123,963,633	24.7%
	new	56,128,581		93,393,532	

As can be seen from the table, the loop transformation results in substantial performance improvements (labeled "new"). The table also shows the reductions achieved in the total number of instructions[1] executed when the loop transformation is used. The overall reduction in execution cycles is greater than the reduction in total number of instructions showing the impact of eliminating branch instructions.

In SDF we can vary the number of functional units (SP and EP's) to achieve higher levels of performance. Table 2 shows the performance gains obtained using the proposed loop transformation with varying number of SPs and EPs. This table shows that the impact of the loop transformation is greater for more functional units. As an aside, the table also shows the scalability of our SDF architecture.

In the next two tables (Tables 3 and 4) we present the data obtained for another benchmark, a picture zooming code segment [10]. The reductions are less significant for this application since the loop bodies contained conditional instructions unrelated to the thread level parallelism being addressed by our loop transformation – these conditional instructions cannot be eliminated by our

[1] Note that the execution cycles are less than the number of instructions because of the use of two processing units, EP and SP. The numbers in show that we achieve approximately 1.6 instructions per cycles (IPC) with two processing elements.

transformation. Table 4 shows that the reductions improve as we add more functional units.

3. Loop Unrolling and Instruction Level Parallelism (ILP)

Loop unrolling relies on replicating the loop body, increasing the size of the straight-line code that can provide higher levels of ILP. Thus unrolling is useful in a variety of processors, including simple pipelines, statically scheduled Superscalar and VLIW systems. Consider the following loop.

```
for (i=n; i>0; i=i-1)   x[i] = x[i] +s;
```

The code segment can be translated into MIPS like assembly language shown below.

```
Loop:   LD      F0, 0(R1)     ; Load Array element into F0
        ADDD    F4, F0, F2    ; add scalar value F2
        SD      F4, 0(R1)     ; store updated value
        DADDUI  R1, R1, #-8   ;decrement pointer
        BNE     R1, R2, Loop  ; branch if not done
```

The dependencies between the load (LD) and the add (ADDD), between the add and the store (SD) and between the decrement (DADDUI) and the conditional branch (BNE) instruction, prevent the hardware from exploiting much, if any ILP. In order to provide greater ILP, we can unroll the loop such that each new iteration actually corresponds to several iterations of the original loop. Below we show unrolling of 4 iterations (both in its original form and after reordering, to eliminate stalls due to instruction latencies).

```
Loop:  LD      F0, 0(R1)          Loop:  LD   F0, 0(R1)
       ADDD    F4, F0, F2                LD   F6, -8(R1)
       SD      F4, 8(R1)                 LD   F10, -16(R1)
       LD      F6, -8(R1)                LD   F14, -24(R1)
       ADDD    F8, F6, F2                ADDD F4, F0, F2
       SD      F8, -8(R1)                ADDD F8, F6, F2
       LD      F10, -16(R1)              ADDD F12, F10, F2
       ADDD    F12, F10, F2              ADDD F16, F14, F2
       SD      F12, -16(R1)              SD   F4, 8(R1)
       LD      F14, -24(R1)              SD   F8, -8(R1)
       ADDD    F16, F14, F2              SD   F12, -16(R1)
       SD      F16, -24(R1)              SD   F16, -24(R1)
       DADDUI  R1, R1, #-32              DADDUI R1, R1, #-32
       BNE     R1, R2, Loop              BNE  R1, R2, Loop

       Before Reordering                 After Reordering
```

We assumed that the loop is executed a multiple of 4 times, hence we needed to test only at the end every 4 original loop iterations. If the number of times a loop is executed is unknown at compile time, it may not be possible to either unroll the loop, re-order loop instruction or avoid conditional instructions (and associated increments/decrement of pointer index register such as R1 in above code example). It would be more difficult to re-order instructions across these branch instructions (unless speculation is used).

```
for (i=n; i>0; i=i-1)
   x[i] = x[i] +s;
```

```
for (i=n'; i>0; i=i-1)
     x[i] = x[i] +s;
for (j = n; j>n'; j- j-1)
     x[j] = x[j] + s;
```

However, the loop can be transformed into two loops (similar to the transformation shown in section 2.1) at compile time as shown.

Here n' is the largest multiple of 4 but smaller than n. The first loop will be executed a multiple of 4 times, and thus can be unrolled four times as before. We may choose not to unroll the second loop. Once again, we need to compute the upper and lower bounds for the two loops after transformation and these values depend on whether the original loop is counting up or counting down, and the step value. Although this code transformation appears obvious, we wanted to explore the actual performance improvements that can be obtained on superscalar and VLIW architectures.

3.1. Experimental Results For Superscalar systems.

For this experiment, we used Simplescalar tool [2] that simulates superscalar architecture using MIPS instructions. We used matrix multiplication and a picture zooming program segment [10] as our benchmarks.

We unrolled loops in both these applications 5 times, but varied the total number of times a loop is executed. The execution cycles, total number of instructions executed, number of branch instructions executed, branch mis-prediction rates, and instruction cache miss rates are used to compare the performance improvements obtained when the code transformation is used (new) with original code (old). Table 5 shows this data for matrix multiplication. As can be seen from the data, although the execution cycles have not shown impressive reductions, the number of branch instructions and the branch mis-prediction rates have shown dramatic improvements. With modern deeply pipelined systems (unlike that simulated by Simplescalar tool), the higher accuracy of branch prediction, and fewer branch instructions can lead to significant performance gains. It should be noted that the improvement depends not only on the configuration but also on how close the number of iterations are to a multiple of 5 (our degree of unrolling).

We also repeated our experiment with different number of functional units, by changing the number of Floating point and Integer units. This data for matrix multiplication is shown in Table 6. Once again, the loop transformation has shown some reduction in the total number of execution cycles for all configurations. This data is presented to emphasize that superscalars do not scale as well as SDF with more multiple functional units.

In the next experiment, we used the picture zooming program segment [10]. Table 7 shows the improvements obtaining using the proposed loop transformation (similar to Table 5). The improvements are not significant because one of the loops of zoom includes many conditional branches. This can be seen by the smaller reduction in the branch instructions shown in Table 7.

3.2. Experiments with VLIW architecture.

In Very Long Instruction Word (VLIW) or Explicitly Parallel Instruction Computer (EPIC) architectures, each instruction packet contains multiple, independently executable instructions. Each of these instructions is executed on available hardware or functional units. Loop unrolling is used to produce longer sequences of straight code whereby independent instructions can be grouped together in VLIW instruction packages. For this experiment we used the Trimaran[2] infrastructure that includes a VLIW architecture simulator and an optimizing compiler. In Table 8, we utilized a 2-wide VLIW machine (comprising one FP and one Integer instruction) for matrix multiplication. We unrolled the innermost loop 5 times. As can be seen from the table, the loop transformation (labeled new) shows higher reductions in execution cycles and total number of branch instructions executed as compared data for superscalar (shown in Tables 5 and 6). Moreover, the improvements are even better for wider VLIW (8-wide, comprising 4 FP and 4 Integer instructions) since more instruction slots can be filled since the unrolling of the first loop requires no conditional instructions.

Table 9 shows the results for zoom. As with the other architectures the performance improvements for zoom are less significant. However, the total number of branch instructions eliminated by the loop transformation is still significant. This in turn can minimize the number of predicated instructions that are needed in VLIW and EPIC architectures to speculatively execute instructions along the branch paths to achieve higher ILP.

4. Conclusions

In our ongoing research we have been exploring computer architectural innovations and compiler optimizations to achieve higher performance levels while minimizing the hardware complexity. We have proposed and evaluated a multithreaded architecture that supports fine-grained parallelism efficiently. In our search for new compiler optimizations for our multithreaded architecture (known as the Scheduled Dataflow, SDF), we are discovering some optimizations that are unique to decoupled and multithreaded architectures, and some techniques that are applicable to a wider variety of models including VLIW and Superscalar systems.

In this paper we have reported on a compile time loop transformation that can be used to increase both instruction level parallelism (ILP) and thread level parallelism (TLP) while eliminating significant number of conditional instructions. We have reported the resulting improvements in execution cycles, reduction of branch instructions executed, improvements in branch prediction accuracy, and reduction of (instruction) cache miss rates on SDF, Superscalar and VLIW architectures.

5. References.

[1] J. Arul. Design, Implementation and Evaluation of the Scheduled Dataflow Architecture, PhD Dissertation, Department of Electrical and Computer Engineering, The University of Alabama in Huntsville, Aug. 2001.

[2] D. Burger and T. M. Austin, “The SimpleScalar Tool Set Version 2.0”, Tech Rep. #1342, Department of Computer Science, University of Wisconsin, Madison, WI.

[3] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach – 3nd Ed.*, Morgan Kaufmann Publisher, 2003.

[4] A.R. Hurson, J.T. Lim, K.M. Kavi and B. Lee "Parallelization of DOALL and DOACROSS loops - a survey", *Advances in Computers*, Vol. 45, pp 54-105, (Edited by M. Zerkowitz), Academic Press 1997.

[5] K.M. Kavi, R. Giorgi and J. Arul. "Scheduled Dataflow: Execution paradigm, architecture and performance evaluation", *IEEE Transactions on Computer*, pp 834-846, Aug. 2001.

[6] K.M. Kavi, J. Arul and R. Giorgi. “Performance Evaluation of a Non-Blocking Multithreaded Architecture for Embedded, Real-Time and DSP Applications”, *ISCA PDCS-2001*, Dallas Texas, August 8-11, 2001, pp 365-371.

[7] K.M. Kavi, J. Arul and R. Giorgi. “Execution and cache performance of the Scheduled Dataflow Architecture”, *Journal of Universal Computer Science*, Special Issue on Multithreaded and Chip Multiprocessors, Oct. 2000, Vol. 6, No. 10, pp. 948-967.

[8] K.M. Kavi, R. Giorgi and J. Arul. “Comparing execution performance of Scheduled Dataflow Architecture with RISC processors”, *13th ISCA PDCS-00),* Las Vegas, Aug. 8-10, 2000, pp 41-47

[9] K.M. Kavi, B. Lee and Ali Hurson. “Multithreaded systems: A survey”, In *Advances in Computers,* Vol. 46, pp 287-327, (Edited by M. Zerkowitz), Academic Press, 1998.

[10] H. Terada, S. Miyata and M. Iwata, “DDMP's: Self-timed Super-Pipelined Data-Driven Multimedia Processor” *Proceedings of the IEEE*, Feb. 1999, pp. 282-296

[2] See http://www.trimaran.org/

Table 2. Performance for Matrix Multiply using Varying Number of Functional Unit

Data Size	□	2SP/2EP	Reduction	3SP/3EP	Reduction	4SP/4EP	Reduction
53*53	old	3,824,933	34.9%	3,549,651	40.6%	3,527,179	38.9%
□	new	2,488,273	□	2,109,058	□	2,154,002	□
104*104	old	24,041,721	50.9%	21,921,785	52.2%	21,748,729	51.4%
□	new	11,798,841	□	10,479,289	□	10,565,817	□
151*151	old	55,600,979	45.0%	48,897,485	46.3%	48,350,261	47.6%
□	new	30,589,490	□	26,257,300	□	25,345,260	□

Table 3. Performance Comparisons for Zoom (1SP-1EP)

Data Size		Execution	Reduction	Total	Reduction
		Cycles		Instructions	
251*251*4	old	9690158	5.7%	16834137	3.5%
	new	9139480		15920537	
519*519*4	old	41542369	5.3%	72288465	3.4%
	new	39346496		68425588	
1024*1024*4	old	160650292	4.1%	278488133	2.4%
	new	154126404		267791469	

Table 4. Performance for Zoom using Varying Number of Functional Units

Data Size		2SP/2EP	Reduction	3SP/3EP	Reduction	4SP/4EP	Reduction
251*251*4	old	4201776	20.2%	1997996	24.1%	1380536	32.0%
	new	3354416		1516845		938792	
519*519*4	old	18061236	20.6%	8726502	25.6%	5939991	35.3%
	new	14340022		6490666		3842728	
1024*1024*4	old	68790308	19.4%	32693284	21.6%	21226532	30.1%
	new	55479348		25630772		14835764	

Table 5. Performance Improvements for matrix multiply (1Integer/1FP units)

Data Size	□	Execution	Reduction	Total	Branch	Reduction	Branch	Reduction
□	□	Cycles	□	Instructions	Instructions	□	Miss predict	□
53*53	Old	15,106,721	3.3%	15,649,081	144,467	46.7%	5772	48.6%
□	New	14,600,815	□	14,949,347	77,049	□	2,964	□
104*104	Old	112,770,174	1.7%	115,620,787	974,858	44.4%	21838	49.5%
□	New	110,811,643	□	112,818,763	542,216	□	11022	□
151*151	Old	345,245,133	3.0%	352,755,560	2,988,537	48.1%	45856	49.7%
□	New	334,801,382	□	340,282,505	1,551,888	□	23053	□

Table 6. Matrix multiplication with varying number of functional units

Data Size	Old/new	2INT/2FP	Reduction	3INT/3FP	Reduction	4INT/4FP	Reduction
53*53	Old	7,577,777	3.4%	5,523,459	4.0%	5,088,545	3.6%
□	New	7,316,968	□	5,304,544	□	4,903,061	□
104*104	Old	56,632,806	1.7%	41,861,802	2.3%	38,654,964	2.0%
□	New	55,645,718	□	40,916,784	□	37,881,737	□
151*151	Old	173,293,194	3.0%	128,050,264	3.5%	118,277,086	3.2%
□	New	168,083,742	□	123,557,479	□	114,446,011	□

Table 7. Performance Improvements for Zoom (1Integer/1FP units)

Data Size		Execution	Reduction	Total	Branch	Reduction	Branch	Reduction
		Cycles		Instructions	Instructions		Miss-predict	
251*125*4	Old	24526903	0.3%	25148193	116914	22.3%	598	42.5%
	New	24452194		25003374	90816		344	
519*519*4	Old	105370877	0.5%	8299544	605579	17.1%	11505	4.6%
	New	104855281		107969098	502303		10981	
1024*1024*4	Old	253180650	0.3%	264844370	2645926	15.6%	22613	4.5%
	New	252390873		264042569	2233264		21590	

Table 8. Performance data for Matrix Multiplication on VLIW

		2-wide		8-wide		Total	
Data Size	Old/New	Execution	Reduction	Execution	Reduction	Branch	Reduction
		Cycles		Cycles		Instructions	
53*53	Old	7883420	12.7%	7872130	16.0%	73088	53.8%
	New	6885154		6615489		33762	
104*104	Old	58852344	14.9%	58808975	17.6%	486825	48.9%
	New	50089427		48434579		248873	
151*151	Old	180929457	17.8%	180838101	20.4%	1527819	52.2%
	New	148754490		144011882		729784	

Table 9. Performance data for Zoom on VLIW

		2-wide		8-wide		Total	
Data Size		Execution	Reduction	Execution	Reduction	Branch	Reduction
		Cycle		Cycles		Instructions	
251*251*4	Old	63049140	0.1%	57698823	0.2%	89859	15.9%
	New	62984131		57579096		75552	
519*519*4	Old	270573583	0.5%	247686201	0.6%	490456	10.6%
	New	269344072		246300990		438556	
1024*1024*4	Old	969789639	0.2%	934544582	0.3%	1778689	11.9%
	New	967771335		931922118		1567745	