Loop-Aware Instruction Scheduling with Dynamic Contention Tracking for Tiled Dataflow Architectures

Muhammad Umar Farooq and Lizy K. John

Department of ECE,
The University of Texas at Austin
ufarooq@mail.utexas.edu, ljohn@ece.utexas.edu
http://www.ece.utexas.edu

Abstract. Increasing on-chip wire delay along with the distributed nature of processing elements, makes instruction scheduling for tiled dataflow architectures very crucial. Our analysis reveals that careful placement of frequently executed sections of applications, and dynamic resource contention tracking can significantly improve the performance of the application. The former reduces the operand network latency, while the latter reduces stalls due to contention for processing elements. We augment one of the most recent instruction scheduling algorithms —hierarchical instruction scheduling —to better exploit spatial locality between instructions within a loop, thereby reducing expensive communication overhead by 6.5% and increasing average IPC by 5.13%. Secondly, in the presence of conditional branches and variable latency memory instructions, estimating resource contention, at compile time, is not only complex but also imperfect. We suggest dynamic tracking of contending instructions, and their re-location, once a contention threshold is exceeded. Results showed that dynamic contention tracking reduced the average ALU conflicts by 23%, thereby improving the average IPC by 14.22%. Combined together, these augmentations improve the average IPC by 19.39% and over 30% for some benchmarks.

Keywords: tiled dataflow architectures, instruction scheduling, resource contention, operand network latency.

1 Introduction

Tiled architectures are gaining popularity as an alternative to monolithic processors, because of their simpler designs, and scalability. TRIPS [1], WaveScalar [13], RAW [14], and Smart-Memories [5] are examples of such architectures. Some examples of these architectures consist of processing elements (PEs), distributed across a grid, and connected through an on-chip network [1][13]. Their performance largely depends on the instruction scheduling. As opposed to monolithic processors, instruction scheduling for tiled architectures has two aspects: (1) temporal —decides when to fetch an instruction (2) spatial —decides where

O. de Moor and M. Schwartzbach (Eds.): CC 2009, LNCS 5501, pp. 190–203, 2009.

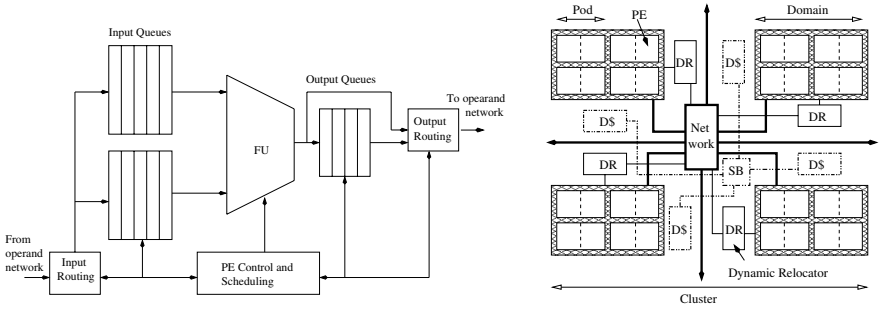

Fig. 1. a) Processing Element (PE) b)WaveScalar Cluster

to execute an instruction. Scheduling for monolithic processors focuses on the temporal aspect of scheduling. However, for tiled architectures, scheduling also decides *where* an instruction should be executed in the grid. A good instruction scheduling can reduce operand network latency by placing dependent instructions on the same or adjacent tiles, while minimizing the contention for tile resources and maintaining high ILP. Previous attempts at solving resource contention problem for tiled architectures relied on profiling[4] or compile time heuristics[2]. In the presence of variable latency memory instructions, estimating at compile time, whether an instruction will contend with another instruction is impossible. For example, in case a load instruction misses in L1 cache, all its consumers will execute at a time that is different from their estimated firing time. The focus of this work is:

- to reduce operand network latency by careful placement of instructions within the loop.
- to minimize resource contention by dynamically tracking and re-locating contending instructions.

Our target architecture is WaveScalar [13]. In this architecture, the basic processing element (PE) is a 5-stage in-order pipeline. Two adjacent PEs form a pod, and communicate through a low-latency bypass network. Four pods make a domain, communicating over a fixed latency pipelined network. Four domains constitute a cluster, and communicate through a fixed-route network switch. Several clusters combine to form a grid. Inter-cluster communication is through a dynamically-routed packet network. Recently, a hierarchical instruction scheduling algorithm [6] has been proposed for the WaveScalar architecture, which partitions the application's dataflow graph into smaller groups, and assign these groups to PEs. We augmented hierarchical instruction scheduling algorithm to take into consideration the control flow information (i.e. loops) while partitioning the dataflow graph. We also explored addressing contention for execution resources dynamically by re-locating instructions that contend with other instructions within their PE past some pre-defined threshold.

We compared the performance of the augmented algorithm with the original algorithm on WaveScalar [13] simulator using benchmarks from EEMBC [11] benchmark suites.

In the next section, we will discuss the background, and the original hierarchical scheduling algorithm. Section 3 describes our augmentations to the original algorithm. Our evaluation methodology, and results are shown in section 4. Section 5 discusses related work. Section 6 concludes the paper.

2 Background

We begin this section by giving an overview of the target architecture for the scheduling techniques presented here. We will then explain the state of the art in instruction scheduling for this architecture which is the baseline for our comparisons.

2.1 WaveScalar Architecture

WaveScalar is a dataflow architecture. As in other dataflow architectures, a program is represented as a dataflow graph, and instruction dependencies are explicit [3][10][8][13]. There is no program counter, instructions are fetched and placed on the grid as they are required. There is no register file, the result produced by an instruction is directly communicated to all the consumers. In this architecture, instructions are grouped in blocks called *Waves*. Waves can be defined as acyclic dataflow graphs, for which each instruction executes at most once every time the wave is executed, and to which control can enter at a single point. On exit and re-entry to this acyclic dataflow graph, the wave number is increased. Waves are used to support memory models of imperative programming languages such as C. Each dynamic instruction is identified by a tag, which is the aggregate of its wave number and location on the grid. When an instruction has received all its input operands for a particular matching wave number, it fires, provided an ALU is available, and there is room to store the result in the output queue. The output is temporary stored in the output queue before it is communicated to the consumers.

Figure 1(b) shows the basic WaveScalar Microarchitecture. The substrate consists of replicated clusters connected through a dynamically routed packet network. Each cluster consists of four domains, communicating through a fixed-route network switch, which has a 4 cycle latency. Additionally, each cluster has a 32KB 4-way set associative L1 data cache, and a store buffer. Each domain is composed of eight PEs, grouped into pairs of two. Each pair is called a pod. Pods communicate through a fixed 1 cycle latency pipeline network. Adjacent pods form a half-domain, with 2 cycle communication latency. Within PE, instructions communicate through a bypass network. Each PE, shown in Figure 1(a), is a 5-stage in-order pipeline with a small instruction cache capable of holding 64 static instructions. Each PE has a 16 entry input queue, and an 8 entry output queue.

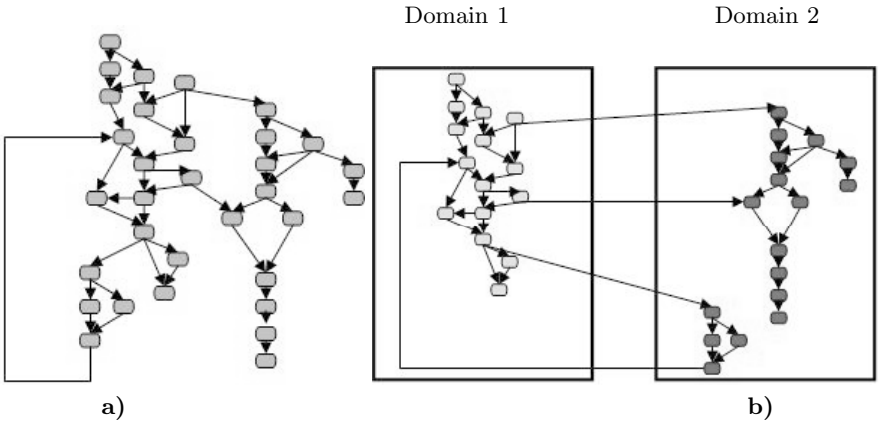

Fig. 2. a) Application Dataflow Graph b)Coarse Grain Scheduling

Loading instructions onto the grid is done through co-operation of the microarchitecture and the runtime system. When an instruction produces a result for a consumer instruction not already on the grid, the runtime system is signalled [6]. The placement of the incoming instruction is decided using either a statically constructed table, or an online algorithm which creates a new mapping.

2.2 Hierarchical Instruction Scheduling Algorithm

Recently, a hierarchical instruction scheduling algorithm [6] has been proposed. It breaks the instruction scheduling problem into two phases – Coarse grain and Fine grain. Coarse grain phase assigns instructions to domains according to their execution order, while fine grain phase refines initial placement, and assigns instructions to the PEs. Coarse grain scheduling uses instruction execution order, obtained through profiling, to assign instructions to a domain, and when the domain is full, it moves to the next domain, thereby assigning all the instructions to some domain. Figure 2 shows an example of how an application dataflow graph shown in 2(a) is assigned to domains during coarse grain scheduling 2(b).

Once all the instructions are assigned to some domain, fine grain scheduling refines the assignments, and generates the final placement of instructions to the PEs. Fine grain scheduling has two passes. First pass forms groups of instructions within each domain according to the topology of the dataflow graph. It uses two parameters (a)*MaxDepth* - which controls how many dependent instructions are assigned to the same PE, (b) *MaxWidth* - which limits the amount of parallelism within each PE. A higher value of *MaxDepth* will reduce operand network latency since more dependent instruction will be assigned to the same PE. A higher value *MaxWidth* will increase the ALU contention as more parallel instructions will share the PE resources. In the second pass, fine grain scheduling assigns the groups formed in first pass to the processing elements. In doing so, this phase uses the parameter *DepDegree*. This parameter has a value between zero and

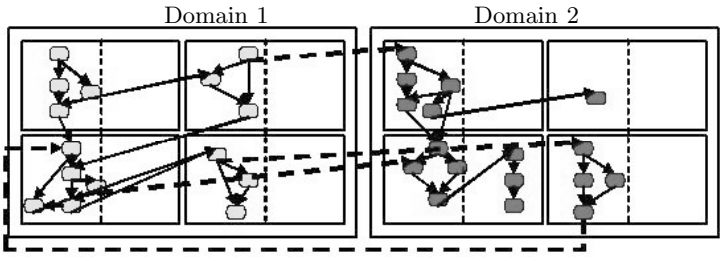

Fig. 3. Fine Grain Scheduling

one, and is used to control how much emphasis there is on inter-group operand dependencies in the choice of PE for a group. A value close to zero will assign dependent groups to the same PE. A value close to one will separate dependent groups by assigning them to different PEs. Figure 3 shows the final instruction placement after fine grain scheduling.

3 Enhanced Hierarchical Instruction Scheduling

This section will explain our augmentations to the baseline hierarchical instruction scheduling algorithm, namely loop-aware instruction scheduling, and dynamic tracking and re-location of contending instructions.

3.1 Loop Awareness

Loops are structures where a program spends most of its time. Careful placement of instructions within a loop can significantly improve the performance of the program by reducing long operand latencies. The baseline algorithm does not differentiate between sequential code within and outside loop constructs, when assigning instructions to domains. During the coarse grain scheduling, the baseline algorithm uses profiled execution order to assign instructions to domains. Once a domain is completely full (512 instructions for the current implementation), the algorithm moves to the next domain until all the instructions in the program are assigned to some domain in the grid. However, this sequential assignment of instructions to domain could result in a loop being split into two different domains (see Figure 2) thereby increasing the inter-domain traffic proportional to the execution frequency of the loop.

In our algorithm, a counter, S_{curr}, is maintained during the coarse grain placement phase. Every time an instruction is assigned to a domain, this counter is incremented. When the coarse grain placement algorithm encounters an instruction that is in a loop it checks the static size of the loop, S_{loop}. If S_{loop} is $<$ S_{max} - S_{curr}, where S_{max} is the maximum instructions that can fit in a domain, it continues the assignment to the current domain since the loop can completely fit in the current domain. If S_{loop} is $>$ S_{max}, implying that the loop can not completely fit in any domain it continues with the assignment to the current domain.

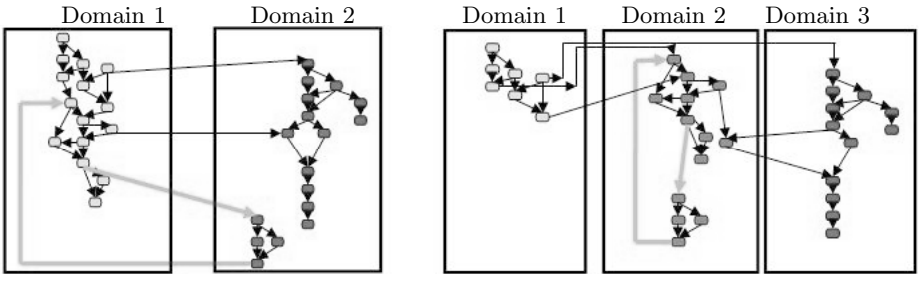

Fig. 4. a) Instruction placement using baseline algorithm. Note that, loop is split between domains. b)Loop-aware instruction placement. Avoids loop splitting by placing all loop instructions in a new domain.

However, if S_{max} - S_{curr} < S_{loop} < S_{max}, i.e. loop can not fit in current domain but can completely fit in a new domain, then the assignment to the current domain is stopped, and entire loop is assigned to a new domain. Doing so decreases the amount of inter-domain communication that would have taken place if the loop was divided between domains. Figure 4 shows a sample placement of a loop with the baseline instruction placement 4(a), and with loop-aware instruction placement 4(b).

3.2 Dynamic Contention Tracking

When more instructions become ready to execute in the same cycle than there are execution slots in the PE, we have resource contention problem. Reducing resource contention requires estimating the execution cycle of every instruction, and placing those instructions with same execution cycle into different PEs. One of the shortcomings of static instruction scheduling is its imperfect estimation of resource contention. For example, variable latency memory instructions make it impossible to statically identify firing time of instructions and their dependants. One can only estimate the firing time, but in case an estimation goes wrong (e.g. because of cache miss), firing time of dependent instructions will be different from their estimated time, and can cause contention with other instructions. Extensive research has been done to compute optimal schedule. Profiling is used in [4] to assign cache hit or miss latency to memory instructions. Load balancing heuristic is used in [2] to penalize instructions that can cause resource contention. To account for imperfect estimates, the algorithm leaves single cycle slack in either direction for the firing time. These attempts can, at best, produce an imperfect local (intra-block) contention estimate. When attempting to find a placement for the dataflow graph corresponding to an application on a grid, such estimates cannot be useful.

We propose a dynamic contention tracking algorithm. Instructions are assigned to the processing element according to the static scheduling algorithm. A *contention counter* is associated with each instruction i, which is incremented

whenever instruction i contends with other instructions in the processing element. When the *contention counter* for instruction i reaches the *contention threshold*, a re-location request for instruction i is generated. Implementation details of dynamic contention tracking algorithm are further explained in the next section.

To allow re-location within the same domain, processing elements are not filled to completion during initial instruction assignment. Re-location outside the initial domain is not considered, as it would increase the communication cost of the instruction with its producers and consumers, and could also result in a loop being split into different domains. For re-locating instruction i, the cost of placing instruction i is computed for all the PEs in that domain, including the original PE, using the algorithm shown in Figure 5. Instruction i is then assigned to the PE with the minimum cost. If no PE within the domain has cost less than the original PE, then instruction i is not re-located. After re-location, the *contention counter* for that instruction is reset. The cost consists of three components:

a) communication cost between producer of instruction i to instruction i.
b) contention cost of instruction i in PE.
c) communication cost between instruction i to its consumers.

Computing the (a) and (c) portions of the cost is straight forward as each instruction knows the locations of its producers and consumers. Computing contention cost involves finding whether an instruction i, if placed in PE p, will contend with the instructions already present in PE p. One simple approach for finding if two instructions will contend with each other is to check if they have the same producer. There are two problems with this approach. First, an instruction can have multiple producers for the same data, e.g. instructions in the merge block following a conditional branch. Second, instructions that do not have a common producer but can possibly fire at the same time are not considered. A second approach is to maintain a history of the last *'k'* cycles in which each instruction became ready. If two instructions have high correlation in their ready times, they are likely to contend with each other, if they are in the same PE. For our experiments we use k=1. The instruction re-locator keeps a count of how many instructions from each PE became ready every cycle. When a re-location request is issued, this information is used to calculate contention cost of each PE being considered. Results showed that even with k=1, ALU contention is reduced significantly. We have assumed a 20 cycle penalty for finding the best PE for the contending instruction, and announcing its new location to its producers and consumers. This penalty is the same as the penalty of bringing an instruction on demand from L2. When an instruction, not already present on the grid, is brought from L2, all its producers and consumers are updated with the instruction's location on the grid. Some or all of the 20 cycle penalty can be hidden since instruction re-location process starts as soon as the instruction reaches its *contention threshold*. If the next message to this re-located instruction arrives after x cycles, then the actual penalty is (20 - x) cycles, (zero if $x > 20$).

Input: Contending Instruction i
Output: PE_{new} —new location of instruction i
1: runningCost = infinity
2: **for all** PEs p in Domain **do**
3: Cost(i, p) = inputLatency(Producer(i), i) + contentionCost(i, p) + outputLatency(i, Consumers(i))
4: **if** Cost(i, p) < runningCost **then**
5: runningCost = Cost(i, p)
6: $PE_{new} = p$
7: **end if**
8: **end for**

Fig. 5. Dynamic Contention Tracking Algorithm

3.3 Implementation of Dynamic Contention Tracking Algorithm

The dynamic contention tracking algorithm is implemented using a hardware structure called the *dynamic re-locator*, see Fig 1(b). This is a distributed structure with one re-locator per domain. The purpose of this re-locator is to calculate the best PE location for an instruction, whose contention counter exceeds the threshold. This new PE location should have the least value of cost function among the 8 PEs in that domain.

When an instruction surpasses the threshold, it sends a re-location request along with the PE location of its sources and sinks to the dynamic re-locator in its domain. Once the re-locator receives this information, computing the communication cost between a candidate PE and PEs containing the sources/sinks is not costly. This is just a matter of adding numbers based on the distance of the source and sink PEs. For computing the contention cost, the re-locator requires information regarding how many instructions became ready in each PE in the previous cycle. Every cycle, each PE (8 of them) sends this information to the local re-locator in their domain. Upon receiving a re-location request, these 8 stored values are observed by the re-locator to calculate the contention cost of each PE, which is incorporated into the overall cost function of each PE.

Another job of re-locator is to update the source and sink instructions with the new location of the re-located instruction. This will only be done when an instruction is re-located, and not every cycle. All the sources and sinks are informed in the same way they would know the place of an instruction being initially brought in from L2 [12]. Due to these similarities to an L1 miss, we have considered a 20 cycle penalty for the re-location, same as the penalty of accessing L2 cache.

4 Experimental Evaluation and Results

The hierarchical placement explained in section 2 is the baseline for our evaluation. We carefully implemented the recent hierarchical instruction placement

Table 1. Parameter settings for experimental evaluation

PEs per Domain	8(4 pods)
PE Input Queue	16 entries
PE Output Queue	8 entries
Instructions per PE	64
ALUs per PE	2
L1 Cache	32KB, 4-way set associative, 128B line, 4 accesses per cycle
L2 Cache	16MB, 4-way set associative, 1024B line, 20 cycle access
Network Latencies	
Within Pod	1 Cycle
Within Half Domain	2 Cycles
Within Domain	4 Cycles
Within Cluster	7 Cycles
Inter Cluster	7 + hop count

presented in [6] within the publicly available WaveScalar toolchain. Then following changes were made to help evaluate our enhancements to the hierarchical instruction scheduling.

a) We augmented the binary translator of WaveScalar, which is used to translate binaries from an Alpha compiler to WaveScalar binaries, to consider control flow information about the loops during the coarse grain scheduling phase.

b) We added to the simulator the dynamic contention tracking algorithm explained in Figure 5.

In order to show the effects of loop-awareness and dynamic contention tracking, we ran benchmarks from the EEMBC benchmark suite. Each benchmark ran for all the combinations of the parameters of the hierarchical instruction placement algorithm ($MaxDepth \in \{2, 4, 8, 12, 16, 32, 50, 64, 128\}$, $MaxWidth \in \{1, 2, 3, 4, 6, 10\}$, and $DepDegree \in \{.1, .5, .9\}$). For each of the aforementioned combinations each benchmark ran four times: 1) without loop optimization without contention tracking, 2) with loop optimization without contention tracking, 3) without loop optimization with contention tracking, 4) with loop optimization with contention tracking. For each benchmark we averaged the results for all the combinations of *MaxDepth*, *MaxWidth* and *DepDegree*.

We setup three different experiments to see the effect of our approach on (a) intra-domain traffic (b) ALU contention and (c) IPC. Table 1 shows microarchitectural parameter settings used for the evaluation. Following subsections will discuss these experiments and their results.

4.1 Intra-domain Communication

Inter-domain communication latency is 7 cycles, compared to maximum 4 cycles within the domain. Reducing the inter-domain traffic can significantly improve

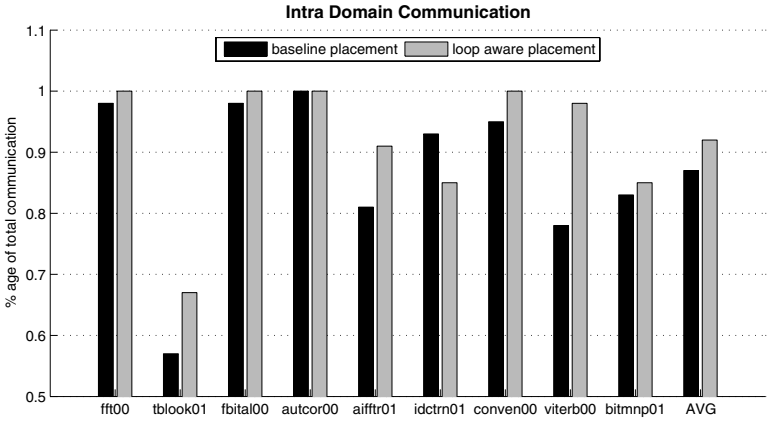

Fig. 6. Intra-domain communications: For each benchmark communication values shown are averaged for all 162 combinations of depth, width and depdegree

the performance. One way of reducing the inter-domain communication is to avoid splitting the loops of a program across multiple domains during instruction placement. This was the focus of our loop-awareness optimization. In order to evaluate the affect of this optimization on intra-domain and inter-domain communication, we measured these values with and without our loop awareness optimization. Experiments showed that by confining the loops to a domain, average intra-domain traffic increased by 6.5% and as high as 25.64% for some benchmarks (see Figure 6). 4 out of 10 benchmarks achieve 99.9+% intra-domain communication. Some of the benchmarks (e.g. idctrn01) however showed a decrease in the intra-domain traffic. This situation arises when a big loop with small iteration count is assigned a new domain by our algorithm, thereby separating the loop instructions from their parent instructions. Since the iteration count is small, assigning a new domain will not increase the intra-domain traffic, however it increases the inter-domain communication between the parent instructions in one domain and loop instructions in another. This situation can be avoided by profiling the loop and starting new domains for only those loops that execute enough times to justify paying the cost of separating the loop's instructions from its producer instructions and is the subject of future work.

4.2 ALU Contention

This experiment shows that dynamic re-location of contending instructions helps reduce ALU contention. Our experiment shows that dynamic contention tracking algorithm reduces average ALU contention by 23%(see Table 2). Total ALU contention with and without dynamic contention tracking algorithm, and the average number of instructions selected for re-location along with their re-location frequencies are shown in Table 2. We would also like to evaluate how accurate our heuristics for re-location of instructions are. Table 2 shows that 46.1% of

Table 2. ALU contention and frequency of instruction re-location during dynamic contention tracking

	Static Instr.		ALU conflicts		% of instructions that are moved			
	Total	Moved	before optimization	after optimization	1-5 times	6-10 times	11-20 times	above 21 times
fft00	15903	602	622365	549867	47.1	3.4	5.7	43.8
tblook01	15498	1198	791137	679818	49	22.7	6.9	20.9
fbital00	14774	383	1263916	883164	47.2	4.6	4.6	43.2
autcor00	14194	77	2823384	2179031	38.8	9	9	42.6
aifftr01	11856	1970	1698660	1286201	39.4	12.8	10.8	36.1
pntr01	16208	501	667252	580794	27.2	3.1	1.7	67
idctrn01	21412	3832	2054979	1618157	68.8	7.7	5.7	17
conven00	14677	363	2510123	1864493	40.4	2.4	4.9	51.4
viterb00	15278	627	938309	620596	58.8	12.3	5.1	22.4
bitmnp01	17437	1349	446836	303761	45.9	23.2	22	8
Average			**1381696**	**1056588**	**46.1**	**10.1**	**7.6**	**36.2**

instructions were moved less than 6 times during the execution of the program. An interesting observation from Table 2 is that most of the instructions are either moved less than 6 times or they are moved for more than 20. This is because during re-location an instruction either finds a PE in which it rarely contends with other instructions early, and stays there for a long period of time or it keeps bouncing back and forth between two PEs. The back and forth movement of instructions between two PEs can be related to the following scenario. A number of instructions start contending within a PE for execution resources. A subset of these instructions reaches the re-location threshold and is moved. The remainder of the contending instructions will also soon reach the threshold and will then look for a place to be re-located. During the re-location process, this second subset of instructions find the same PE that the first set had found due to the fact that the producer and consumer communication benefits of that PE outweigh its contention cost. This movement of instructions back and forth between two PEs is clearly not desirable. The trade off that exists here is between the threshold at which the re-location is initiated and the number of times we pay the price of re-location in order to separate contending groups. A small threshold has the advantage of separating contending groups quickly in order to acheive more parallelism while having to pay the price of re-location more often, whereas a large threshold pays the re-location price less but pays more in terms of contention while we wait for the threshold to be met. An appropriate threshold value can be decided upon with help from profiling and using a metric such as the number of iterations of loops in the program to guide the choice. This will set some sort of upper bound on the value of the threshold in order for there to be any use in re-locating instructions and benefiting from less contention in future iterations of the loop. A lower bound for the threshold value will involve the re-location cost. We experimentally found the threshold of 20 used in these experiments to be a sweet spot for the threshold.

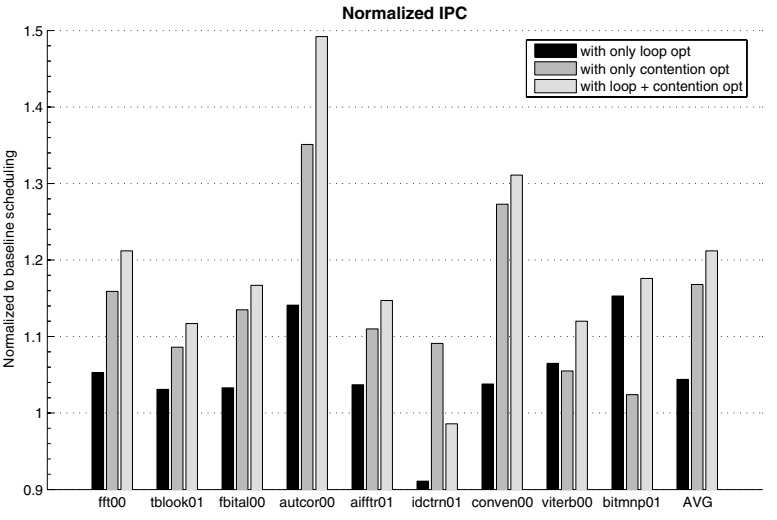

Fig. 7. IPC: For each benchmark IPC values shown are averaged for all 162 combinations of depth, width and depdegree

4.3 IPC

Increased intra-domain traffic as well as reduced ALU contention have significant positive impact on the IPC. Reducing the inter-domain communication through loop-awareness and ALU contention by dynamic contention tracking algorithm improves average IPC by 19.39% and over 30% for some benchmarks e.g., conven00 and autcor00. Figure 7 shows the individual and combined effect of our enhancements on IPC. Note that for benchmarks idctrn01, combined IPC decreases because of decrease in the intra-domain communication explained in Section 4.1. However for this benchmark increase in IPC due to reduction in ALU contention is still achieved although outweighed by the decrease in IPC due to decreased intra-domain communication.

5 Related Work

In this section we dicuss the most relevant work on algorithms designed for instruction scheduling which have a spatial component.

5.1 Spatial Path Scheduling Algorithm

One of the most recent instruction schedulers proposed for tiled dataflow architectures is the SPS algorithm presented for the EDGE architecture in [2]. The spatial path scheduling algorithm factors in previously fixed locations which it calls anchor points for each placement. An anchor point is an instruction whose placement is known because it accesses a known location such as the register

file (if the architecture has one), a cache bank or other resources. As the placement for instructions is decided upon, the instructions that have been placed become new anchor points for the remainder of the instructions to be placed. The proposed approach uses simulated annealing to estimate the best results that are possible and uses heuristics to close the gap between the basic algorithm explained above and the results obtained via simulated annealing. To do so the basic algorithm is augmented with three heuristics: (1) local and global ALU and network link contention modeling, (2) global critical path estimates and (3) dependence chain path reservation. Using these heuristics the placement cost function is modified to account for the mentioned criteria. In [2] it is shown that with all the heuristics in place, the final scheduler improves over the basic SPS algorithm by 7%, and is within 5% of the annealed results. This method is not very suitable for a dataflow architecture such as WaveScalar because such an architecture does not have register files to use as anchor points at the beginning of the algorithm.

5.2 Instruction Scheduling for Clustered VLIW

A number of instruction scheduling algorithms have been proposed for clustered VLIW architectures [4][7][9]. Unified assign and schedule [7] is a general scheduling framework which is augmented with heuristics for the target architecture by the compiler writer. This work was compared to the baseline hierarchical fine grain algorithm in [1]. [9] predicts the inter-cluster communication cost of a loop, and uses an integer-optimization method to control loop unrolling and unroll-and-jam to limit the effects of inter-cluster data transfers. This method differs from our algorithm in the way information from loops is used since it does not address inter-cluster communication by taking loops into account in instruction scheduling. By addressing minimization of inter-domain communication through placement our algorithm does not need to restrict unrolling to limit this communication if a loop structure can fit in a domain of its own.

6 Conclusion

Loop-Awareness and dynamic contention tracking techniques were added to the hierarchical placement algorithm [6]. Loop-aware hierarchical instruction scheduling improves the performance of tiled architectures by considering control flow information, specifically loop information, when doing coarse grain instruction placement. This avoids splitting of loops into multiple domains thereby increasing average intra-domain communication by 6.5% and average IPC by 5.13% and as high as 15% for some benchmarks. Dynamic tracking and relocation of contending instructions resulted in an average 23% reduction in ALU conflicts thereby increasing average IPC by 14.22%. These two enhancements put together achieved an average IPC improvement of 19.39% and over 30% for some presented benchmarks.

References

1. Burger, D., Keckler, S.W., McKinley, K.S., Dahlin, M., John, L.K., Lin, C., Moore, C.R., Burrill, J., McDonald, R.G., Yoder, W., The TRIPS Team: Scaling to the End of Silicon with EDGE Architectures. Computer 37(7), 44–55 (2004)
2. Coons, K.E., Chen, X., Burger, D., McKinley, K.S., Kushwaha, S.K.: A Spatial Path Scheduling Algorithm For EDGE Architectures. In: ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, pp. 129–140. ACM Press, New York (2006)
3. Dennis, J.B., Misunas, D.P.: A preliminary architecture for a basic data-flow processor. SIGARCH Comput. Archit. News 3(4), 126–132 (1974)
4. Gibert, E., Sanchez, J., Gonzalez, A.: Effective instruction scheduling techniques for an interleaved cache clustered VLIW processor. In: MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture, pp. 123–133. IEEE Computer Society Press, Los Alamitos (2002)
5. Mai, K., Paaske, T., Jayasena, N., Ho, R., Dally, W.J., Horowitz, M.: Smart Memories: A Modular Reconfigurable Architecture. In: ISCA 2000: Proceedings of the 27th annual international symposium on Computer architecture, pp. 161–171. ACM Press, New York (2000)
6. Mercaldi, M., Swanson, S., Petersen, A., Putnam, A., Schwerin, A., Oskin, M., Eggers, S.J.: Instruction scheduling for a tiled dataflow architecture. In: ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, pp. 141–150. ACM Press, New York (2006)
7. Ozer, E., Banerjia, S., Conte, T.M.: Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In: MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture, pp. 308–315. IEEE Computer Society Press, Los Alamitos (1998)
8. Papadopoulos, G.M., Culler, D.E.: Monsoon: An explicit token-store architecture. In: ISCA 1998: 25 years of the international symposia on Computer architecture (selected papers), pp. 398–407. ACM Press, New York (1998)
9. Qian, Y., Carr, S., Sweany, P.H.: Optimizing Loop Performance For Clustered VLIW Architectures. In: PACT 2002: Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques, Washington, DC, USA, pp. 271–280. IEEE Computer Society Press, Los Alamitos (2002)
10. Sakai, S., Yamaguchi, y., Hiraki, K., Kodama, Y., Yuba, T.: An architecture of a dataflow single chip processor. In: ISCA 1989: Proceedings of the 16th annual international symposium on Computer architecture, pp. 46–53. ACM Press, New York (1989)
11. EEMBC Benchmark Scores, `http://www.eembc.org`
12. Swanson, S., Michelson, K., Schwerin, A., Oskin, M.: Dataflow: The Road Less Complex. In: WCED 2003: Proceedings of the 3rd Workship on Complexity-Effective Design (2003)
13. Swanson, S., Michelson, K., Schwerin, A., Oskin, M.: WaveScalar. In: MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, p. 291. IEEE Computer Society Press, Los Alamitos (2003)
14. Waingold, E., Taylor, M., Sarkar, V., Lee, V., Lee, W., Kim, J., Frank, M., Finch, P., Devabhaktumi, S., Barua, R., Babb, J., Amarsinghe, S., Agarwal, A.: Baring it all to Software: The Raw Machine. Technical report, Cambridge, MA, USA (1997)