Cooperative instruction scheduling with linear scan register allocation

Khaing Khaing Kyi Win and Weng-Fai Wong

Department Computer Science
National University of Singapore
3 Science Drive 2, Singapore 117543
khaing_khaing@alumni.nus.edu.sg, wongwf@comp.nus.edu.sg

Abstract. Linear scan register allocation is an attractive register allocation algorithm because of its simplicity and fast running time. However, it is generally felt that linear scan register allocation yields poorer code than allocation schemes based on graph coloring. In this paper, we propose a pre-pass instruction scheduling algorithm that improves on the code quality of linear scan allocators. Our implementation in the Trimaran compiler-simulator infrastructure shows that our scheduler can reduce the number of active live ranges that the linear scan allocator has to deal with. As a result, fewer spills are needed and the quality of the generated code is improved. Furthermore, compared to the default scheduling and graph coloring allocator schemes found in the IMPACT and Elcor components of Trimaran, our implementation with our pre-pass scheduler and linear scan register allocator significantly reduced compilation times.

1 Introduction

Instruction scheduling and register allocation are one of the most important phases in compiler optimization. In compilers for machines with instruction-level parallelism, the phases of instruction scheduling and register allocation can be antagonistic. This is the well-known phase ordering problem [7] as shown in Fig. 1. One of the ways to solve that problem is to combine instruction scheduling and register allocation such that these two phases can be performed together to generate efficient code. In current optimizing compilers, a compromise consisting of a phase of instruction scheduling (pre-pass scheduling) is first performed. This is followed by register allocation and another phase of instruction scheduling (post-pass scheduling). The linear scan register allocator proposed by Poletto and Sarkar [13] is very simple and significantly faster than algorithms based on graph coloring approaches. The performance of a linear scan register allocator is affected by the maximum number of active live intervals. If we can reduce the maximum number of active live intervals, the linear scan register allocator can generate a more efficient code by reducing the amount of spill code inserted. Thus, we propose a pre-pass local instruction scheduler which can reduce simultaneously live ranges so as to decrease the maximum number of active live

intervals. We combined the our proposed scheduler with a linear scan register allocator and evaluated the overall performance. Some previous experimental evaluation and improvements to the linear scan register allocation can be found in [6] and [14]. Previous studies [13], [6] and [14] investigated the linear scan register allocator in isolation rather than its combination with instruction scheduling. In addition, most previous works on phase ordering problem [7], [1], [12], [10], [2] and [4] had focused on combining the instruction scheduling phase with register allocator based on graph coloring approaches. In this paper, we focus on a *co-operative* approach that solves the phase ordering problem between instruction scheduling and linear scan register allocation. In order to evaluate the performance of combining our proposed scheduler with linear scan register allocation, we have implemented our proposed scheduler and linear scan register allocator in Trimaran. Trimaran [16] is a compiler infrastructure for supporting state of the art research in compiling for Instruction Level Parallel (ILP) architectures. The system is oriented towards EPIC (Explicitly Parallel Instruction Computing) architectures, and supports compiler research in what is typically considered to be "back end" techniques such as instruction scheduling, register allocation, and machine-dependent optimizations. In our framework, we first perform our proposed scheduler, followed by linear scan register allocation and finally the Trimaran-Elcor list scheduler. The results show that performing our proposed pre-pass scheduler can reduce the maximum active live intervals of the linear scan register allocator. This can decrease the amount of spill code inserted by the linear scan register allocator thereby increasing the quality of the generated code. Moreover, it also shows that combining our proposed pre-pass scheduler with linear scan register allocator is significantly faster than Trimaran's pre-pass scheduling, register allocation, and post-pass scheduling scheme using either the IMPACT or the region-based register allocator.

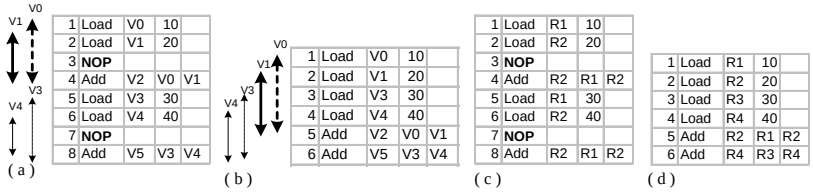


Fig. 1. Example of the phase ordering problem. (a)Sample code in a pseudo intermediate language with live ranges. (b)After instruction scheduling with live ranges. (c)Register allocation first. (d)Instruction scheduling first. Memory access operations are assumed to take two cycles while all other operations take one cycle. Comparing (a) and (b) we see the increase in the number of overlapping live ranges after instruction scheduling. If the register allocation is executed first, it would require eight cycles although only two registers is enough for register allocation. However, if instruction scheduling is done first, then although it would require only six cycles, four registers would be needed to avoid spilling. Which of these two orders is better depends upon the number of available registers and functional units.

The rest of the paper is organized as follows. In Section 2, we give the overview of our proposed scheduler. Section 3 discusses register allocators in Trimaran, followed by the experimental result and discussion of cooperative our proposed pre-pass scheduler with linear scan register allocator (Section 4). Section 5 concludes the paper.

2 Overview of our proposed scheduler

Our proposed scheduler is based on convergent scheduling [11]. As in the convergent scheduler, we used a weight matrix to compute the schedule time of an operation. However, while the convergent scheduler schedules the instructions at the earliest cycle, our proposed scheduler schedules the instructions in the latest cycle possible while maintaining optimal instruction scheduling length which we assumed to be the critical path of a basic block. In order to get our proposed schedule, we use a weight matrix to calculate the optimal schedule length. In particular, the weight $W_{i,t}$ of instruction i at time slot t is a value between zero and one.

Our heuristic is based on the earliest completion time (the longest path from the root node to the current node) and latest completion time (i.e., the critical path length - the longest path from the current node to a leaf node) of the dependence graph. The operation nodes exist either on the critical path (the longest path of the root node to leave node) or non-critical path of the DAG. The optimal schedule length can be assumed as the length of the critical path length if the available resources are not in conflict. Normally, we can only reorder the operation nodes which are not on the critical path length in order to get more efficient codes. This is especially in the case when there are two kinds of non-critical paths: one that starts with a node which has no dependence predecessor and ends at a node on the critical path, or a second kind that starts at a node on the critical path and ends with a node which has no dependence successor. We should schedule the operation nodes which are on the first kind of non-critical path at the latest cycle possible to reduce the simultaneously live ranges. In contrast, the second kind of non-critical paths should be scheduled at the earliest cycle possible. In [15], Chen reported that real dependence graphs of programs have more of the first kind of non-critical paths than the second. Therefore, our proposed scheduler applies as late as possible schedule. If l_e is the earliest completion time and l_l is the latest completion time, the instruction can be scheduled only in the time slots between l_e and l_l. If the instruction could not be scheduled between l_e and l_l due to insufficient parallel functional units, we increase l_l by one and reschedule again. To begin, the value of $W_{i,t}$ is initialized as follows:

$$W_{i,t} = \begin{cases} 0 & \text{if } t < l_e \text{ or } t > l_l; \\ 1/I(t) & \text{if } t \geq l_e \text{ and } t \leq l_l. \end{cases}$$

where $I(t)$ is the number of instructions that has its (l_e, l_l) crossing time t.

We give more weight to a specific instruction to be scheduled in a given time cycle by multiplying the weight with a constant value. Then, we normalize our weights:

$$\forall i, t, \hat{W}_{i,t} \leftarrow \frac{W_{i,t}}{\sum_{t'} W_{i,t'}}$$

The schedule time for each instruction is then $\max_t\{\hat{W}_{i,t}\}$. The full algorithm of our proposed scheduler is given in Fig. 2.

```
1.Compute the earliest completion time and latest completion time
2.Initialize the weight matrix
3.Mark the scheduling is not finished
4.While scheduling is not finished
5.  While the latest completion time is greater than zero
6.     For (each operation within a basic block)
7.        If (resource is available and weight matrix is not zero)
8.             Multiply weight matrix by 1.2 and put back into weight matrix
9.             Mark the operation with schedule
10.            Increase the number of current resources by one
11.            if the current resources reach the maximum resource limit
12.               Decrease the latest completion time by one
13.               Initialize the number of current resources by zero
14.           endif
15.       endif
16.     endfor
17.  endwhile
18. Mark the scheduling is finished
19.  For (each operation within a basic block)
20.     If one of the operations within a basic block is not scheduled
21.        Mark the scheduling is not finished
22.     endif
23.  endfor
24.endwhile
23.Normalize the weight matrix by dividing each weight with the
   total weights for each operation
24.Choose the cycle time which has the maximum weight for each
   operation as schedule time
```

Fig. 2. Our proposed scheduler

3 Register allocators in Trimaran

Global register allocation based on graph coloring was first proposed by Chaitin et al. [3]. A graph-coloring register allocator iteratively builds an undirected

graph called an *interference graph* that shows the overlap in live ranges. A node in an interference graph is a live range that is a candidate for register allocation and an edge connects two nodes when the corresponding live ranges overlap. The standard graph coloring method heuristically attempts to find a k-coloring for the interference graph. A graph is k-colorable if each node can be assigned to one of k-colors such that no two adjacent nodes have the same color. If the heuristic can find a k-coloring, then k registers are sufficient to hold the content of all the live ranges. Otherwise, some candidates are chosen to be spilled, and the interference graph must be rebuilt after a spill decision is made. Another attempt is then made to obtain a k-coloring. This whole process is repeated until a k-coloring is finally obtained. In practice, the cost of graph-coloring approach can be expensive by repeatedly constructing a register interference graph until the heuristic succeeds. However, the graph-coloring based register allocators have been used in many commercial compilers to obtain significant improvements over simple register allocation heuristic. In Trimaran, there have been two global register allocators: the IMPACT register allocator [8] and region-based register allocator [9], adapted from Chow and Hennessy graph-coloring framework [5].

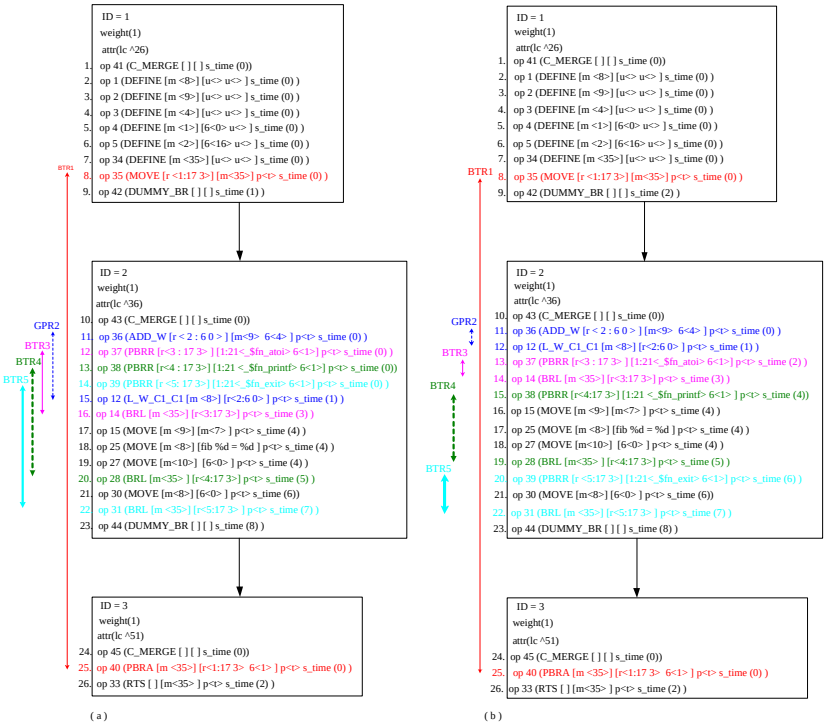


Fig. 3. Control Flow Graph (CFG) with long instructions within each basic block from Trimaran. (a) without instruction reordering. (b) with instruction reordering.

3.1 Linear scan register allocator

Register allocation based on graph-coloring is generally considered the state-of-the-art. However, the algorithm can be computationally expensive. In light of this, Poletto and Sarkar [13] proposed an alternative algorithm for fast register allocator called linear scan register allocation. Linear scan register allocation works on the topological ordering of live ranges (also known as live intervals). Live intervals of each temporary variable are computed and assigned registers. A temporary variable with overlapping intervals can be assigned to different registers and non-overlapping intervals can be assigned to same registers. A linear scan register allocator performs the following four steps [6]:

1. sort all the instructions in topological order;
2. calculate the set of live intervals;
3. assign each temporary variable to physical register for each interval (or spill into the memory) and finally
4. rewrite the code with the obtained allocation.

Ordering of instructions. The topological ordering of basic blocks required by the linear scan allocator is not unique. In particular, the ordering may be (1) depth-first, (2) preorder, (3) postorder, (4) breadth-first, (5) prediction, and (6) random. An experimental study of the impact of these orderings can be found in [6]. Among the different orderings, depth-first ordering was found empirically to reduce the most amount of false interference between live intervals [13, 6]. However, there has been no discussion of reordering of instructions *within* a basic block. The order of instructions within a basic block impacts the allocation and the number of spill code insertions. Our proposed schedule described in Section 2 reorders instructions within a basic block to reduce simultaneously live ranges. Fig. 3(a) and (b) show the original ordering and the ordering of instruction after applying our proposed scheduler respectively.

Computation of live intervals. Live ranges are determined by a set of instructions within each basic block. Each live range has a start position with the first definition of the temporary and an end position with the last use of the temporary. Then, all live intervals are sorted in the order of increasing start-points so as to make the allocation more efficient. The number of live intervals with start position and end position in Fig. 3 are given in Fig. 4.

As shown in Fig. 4,without instruction reordering, BTR1, BTR3, BTR4 and BTR5 are live at the same time. However, with instruction reordering, BTR1 is only live at the same time with BTR3, BTR4 or BTR5.

Register assignment. After sorting all live intervals by their start points, the allocation of registers to intervals can be done. In Trimaran, there are four register types : general purpose registers (GPRs), floating point registers (FPRs), branch target registers (BTRs) and predicate registers (PRs). We performed linear scan allocation on all four types.

Register name	Start position	End position
BTR1	8	25
GPR2	11	15
BTR3	12	16
BTR4	13	20
BTR5	14	22

(a)

Register name	Start position	End position
BTR1	8	25
GPR2	11	12
BTR3	13	14
BTR4	15	19
BTR5	20	22

(b)

Fig. 4. A number of live intervals for the data dependent graph in Fig. 3. (a) without instruction reordering (b) with instruction reordering.

Code rewrite. After register assignment, the code is rewritten to bind the temporary variables to physical registers.

Table 1. The maximum active live intervals of each procedure, which have long and narrow data dependent graph, of several benchmarks in Trimaran.

	GPR			FPR			BTR		
Benchmarks(procedure)	Act1	Act2	Reduce%	Act1	Act2	Reduce%	Act1	Act2	Reduce%
181.mcf(_insert_new_arc)	16	11	31.25%	0	0	0%	1	1	0%
181.mcf(_replace_weaker_arc)	17	12	29.41%	0	0	0%	1	1	0%
181.mcf(_price_out_impl)	29	24	17.24%	0	0	0%	2	2	0%
181.mcf(_suspend_impl)	19	13	31.58%	0	0	0%	1	1	0%
181.mcf(_global_opt)	3	3	0%	0	0	0%	7	2	71.43%
101.tomcatv	69	65	5.80%	33	32	3.03%	7	2	71.43%
wc(_main)	7	7	0%	0	0	0%	3	2	33.33%
bmm (_sumup)	6	6	0%	1	1	0%	3	1	66.67%
dag	11	11	0%	0	0	0%	1	1	0%
eight	8	8	0%	0	0	0%	2	1	50.00%
example_bench(_convert_to_int)	2	2	0%	0	0	0%	3	2	33.33%
fact2	3	3	0%	0	0	0%	3	2	33.33%
fib	4	4	0%	0	0	0%	3	2	33.33%
fib_mem	6	6	0%	0	0	0%	3	2	33.33%
fir	11	11	0%	3	3	0%	3	2	33.33%
hyper	5	5	0%	0	0	0%	1	0	100.00%
ifthen	13	13	0%	0	0	0%	2	1	50.00%
mm_ double (_matmult)	11	11	0%	3	3	0%	2	1	50.00%
mm_ int	14	13	7.14%	0	0	0%	2	1	50.00%
mm	11	11	0%	3	3	0%	3	2	33.33%
nested	6	6	0%	1	1	0%	2	1	50.00%

Act1 - the number of active live intervals after Elcor pre-pass scheduler
Act2 - the number of active live intervals after our pre-pass scheduler

4 Experimental evaluation

We use Trimaran infrastructure to compare the performance of our linear scan register allocator with the IMPACT register allocator and region based register allocator. We also implemented our pre-pass scheduler in Trimaran.

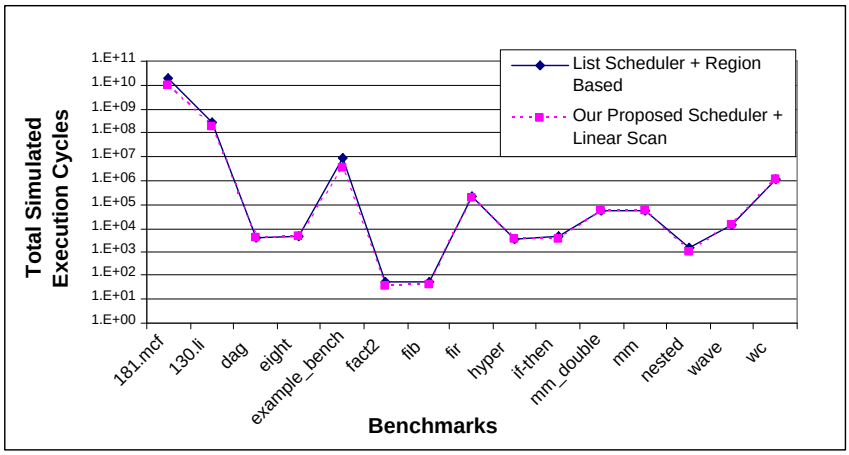


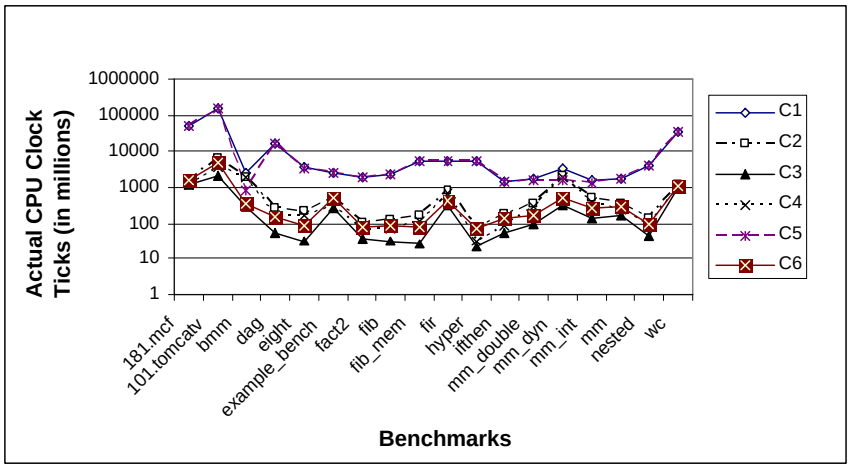
Fig. 5. Total simulated execution cycles.

4.1 Result and discussion

Table 1 gives the experimental results of combining our pre-pass scheduler with linear scan register allocator. The result suggests that combining our scheduler with linear scan register can significantly reduce the maximum active live intervals of basic block. This in turn may reduce spill code insertion and remove unnecessary dependencies. As a result, as shown in Fig. 5, the code generated by our simpler scheme performs just as well as that generated by list scheduling and graph-coloring based register allocation. In some cases, some minor gains were even achieved.

Fig. 6 shows the actual compilation time observed for the various combination of scheduler and register allocators. A linear scan register allocator attempts to find the number of live intervals which are currently active at a certain program point by visiting each lifetime interval in turn. The number of active live intervals represent the number of register needed at this point in the program. If there are insufficient number of free registers, then some active live intervals are chosen to spill and the scan proceeds. Since a linear scan register allocator scans the whole process linearly rather than repeating the process after spill code is inserted, it can operate faster than graph-coloring method based register allocators such as the IMPACT and region-based register allocators in Trimaran.

The list scheduler never unschedules already scheduled operations. Our pre-pass scheduler, on the other hand, unschedules the operations when all the operations cannot schedule within critical path length due to a shortage of function units. However, unlike the list scheduler, our proposed scheduler does not need to recalculate l_e of successor ops after an op has been scheduled. Thus, the compilation time of our pre-pass scheduler is comparable with the list scheduler. As a result, combining our pre-pass scheduler with linear scan register allocator is significantly faster than combining Trimaran's list scheduler with either



C1 - Trimaran list scheduler with IMPACT register allocator
C2 - Trimaran list scheduler with region-based register allocator
C3 - Our pre-pass scheduler with linear scan register allocator
C4 - Trimaran list scheduler with linear scan register allocator
C5 - Our pre-pass scheduler with IMPACT register allocator
C6 - Our pre-pass scheduler with region-based register allocator

Fig. 6. Actual compilation time on an 1.2 GHz AMD Athlon MP Linux system with 1 GByte RAM.

the IMPACT or region-based register allocator. Putting the above together, we therefore argue that combining our pre-pass scheduler with linear scan register allocation is the most cost-effective option.

5 Conclusion

In this paper, a cooperative approach utilizing our pre-pass local instruction scheduling and linear scan register allocation has been presented. As far as we know, this is the first study that combines instruction scheduling with linear scan register allocation. The results show that combining our proposed pre-pass scheduler with the linear scan register allocator can reduce the maximum number of active live intervals. This in turn can reduce register pressure and spill code insertion resulting in high quality code comparable to that generated by a list scheduler and a graph-coloring register allocator. Moreover, compared with the latter our scheme results in significantly lower compilation times. As a future work, we will consider the problem of how to do cooperative *global* scheduling with linear scan register allocation. Yet another approach is to fully integrate instruction scheduling with register allocation.

References

1. D.G. Bradlee, S.J. Eggers, and R.R. Henry. Integrating register allocation and instruction scheduling for RISCs. In Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 122-131, Santa Clara, CA, April 1991.
2. D. Berson, R. Gupta, and M.L. Soffa. Integrated instruction scheduling and register allocation techniques. In Eleventh International Workshop on Languages and Compilers for Parallel Computing, LNCS, Springer Verlag, North Carolina, Chapel Hill, August 1998.
3. G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein. Register allocation via coloring. Computer Languages, 6:47–57, Jan. 1981.
4. G. Chen, and M.D. Smith. Reorganizing global schedules for register allocation. In Proceedings of the 13th international conference on Supercomputing, May 1999.
5. F. Chow, and J. Hennessy. The Priority Based Coloring Approach to Register Allocation. ACM Transactions on Programming Languages and Systems, vol. 12, 501-536, 1990.
6. E. Johansson, and K. Sagonas. Linear scan register allocation in a high performance Erlang compiler. Practical Applications of Declarative Languages: Proceedings of the PADL'2002 Symposium (Lecture Notes in Computer Science, vol. 2257). Springer: Berlin, 2002; 299-317.
7. J.R. Goodman, and W.C. Hsu. Code scheduling and register allocation in large basic blocks. In 1988 International Conference on Supercomputing, pages 442-452, Orlando, Florida, November 1988.
8. R.E. Hank. Machine Independent Register Allocation for the Impact-I C Compiler. BS Thesis, University of Illinois at Urbana-Champaign, 1990.
9. H. Kim. Region-based register allocation for EPIC architecture. Ph.D. Thesis, New York University, January 2001.
10. C. Norris, and L.L. Pollock. An experimental study of several cooperative register allocation and instruction scheduling strategies. In Proceedings of the 28th Annual International Symposium on Microarchitecture, p.169-179, November 29-December 01, 1995, Ann Arbor, Michigan, United States.
11. W. Lee, D. Puppin, S. Swanson, and S. Amarasinghe. Convergent Scheduling. In Proceedings of the 35th Annual International Symposium on Microarchitecture (MICRO), Istanbul, Turkey, November 2002.
12. S.S. Pinter. Register allocation with instruction scheduling: a new approach. In Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation, June 1993.
13. M. Poletto, and V. Sarkar. Linear scan register allocation. ACM Transactions on Programming Languages and Systems (TOPLAS), v.21 n.5, p.895-913, Sept. 1999.
14. K.F. Sagonas, and E. Stenman. Experimental evaluation and improvements to linear scan register allocation. Softw., Pract. Exper. 33(11): 1003-1034 (2003).
15. G. Chen. Effective Instruction Scheduling with Limited Registers. Ph.D. thesis, Harvard University, Division of Engineering and Applied Sciences, March 2001.
16. http://www.Trimaran.org