MIRS: Modulo Scheduling with Integrated Register Spilling

Javier Zalamea, Josep Llosa, Eduard Ayguadé and Mateo Valero

Departament d'Arquitectura de Computadors (UPC)*
Universitat Politècnica de Catalunya
{jzalamea,josepll,eduard,mateo}@ac.upc.es

Abstract. The overlapping of loop iterations in software pipelining techniques imposes high register requirements. The schedule for a loop is valid if it requires at most the number of registers available in the target architecture. Otherwise its register requirements have to be reduced by spilling registers to memory. Previous proposals for spilling in software pipelined loops require a two–step process. The first step performs the actual instruction scheduling without register constraints. The second step adds (if required) spill code and reschedules the modified loop. The process is repeated until a valid schedule, requiring no more registers than those available, is found.
The paper presents *MIRS* (Modulo scheduling with Integrated Register Spilling), a novel register–constrained modulo scheduler that performs modulo scheduling and register spilling simultaneously in a single step. The algorithm is iterative and uses backtracking to undo previous scheduling decisions whenever resource or dependence conflicts appear. *MIRS* is compared against a state–of–the–art two–step approach already described in the literature. For this purpose, a workbench composed of a large set of loops from the Perfect Club and a set of processor configurations are used. On the average, for the loops that require spill code a speed–up in the range 14–31% and a reduction of the memory traffic by a factor in the range 0.90–0.72 are achieved.
Keywords: Instruction-Level Parallelism, Software Pipelining, Register Allocation, Spill Code .

1 Introduction

With the improvements in the underlying technology, current processors are being designed with the aim of supporting increasing levels of instruction–level parallelism (ILP). Software pipelining [10] is an instruction scheduling technique able to exploit this ILP out of a loop by overlapping operations from various successive loop iterations. Different approaches have been proposed in the literature [2] for the generation of software pipelined schedules. Some of them mainly focus on achieving high throughput [1, 12, 17, 24, 25, 27].

* This work has been supported by the Ministry of Education of Spain under contract TIC 98/511, and by CEPBA (European Center for Parallelism of Barcelona). Javier Zalamea is granted by the Agencia Española de Cooperación Internacional.

Register allocation consists in finding the final assignment of registers to loop variables (variants and invariants) and temporaries. It has been extensively studied in the framework of acyclic schedules [4, 6, 7, 8] based on the original graph coloring proposal [9]. However, software pipelining imposes some constraints that prevent the use of these techniques for register allocation. For instance, the lifetime of loop variables may cross the boundary of iterations and may last for more than one iteration, interfering with themselves. Several proposals have been made to handle these constraints [14, 15, 26].

The main drawback of these aggressive scheduling techniques is their high register requirements [21, 23]. This has motivated some recent modulo scheduling proposals whose main objective is the minimization of the register pressure [11, 13, 16, 19, 22]. Figure 1.a (dashed lines) shows the percentage of loops in our workbench (Section 4) that require less than a specific number of registers for the set of processor configurations considered in this paper and assuming an unbounded number of registers. Figure 1.a (solid lines) shows the percentage of the total execution cycles spent on them. Notice that although the percentage of loops that require a high number of registers is small, the percentage of time is important. For instance, only 8–14% of the loops require more than 32 registers but they represent 34–45% of the total execution time.

Having a schedule that uses more registers than those available in the target architecture requires some additional actions [26]. One of the options is to reschedule the loop with a reduced execution rate (i.e. with less iteration overlapping); this reduces the number of overlapped operations and variables. Unfortunately, the register reduction is at the expense of a reduction in performance. Another option is to spill some variables to memory, so that they do not occupy registers for a certain number of clock cycles. This requires the insertion of store and load instructions that free the use of these registers. The evaluation in [20] shows that reducing the execution rate tends to generate worse schedules than spilling variables.

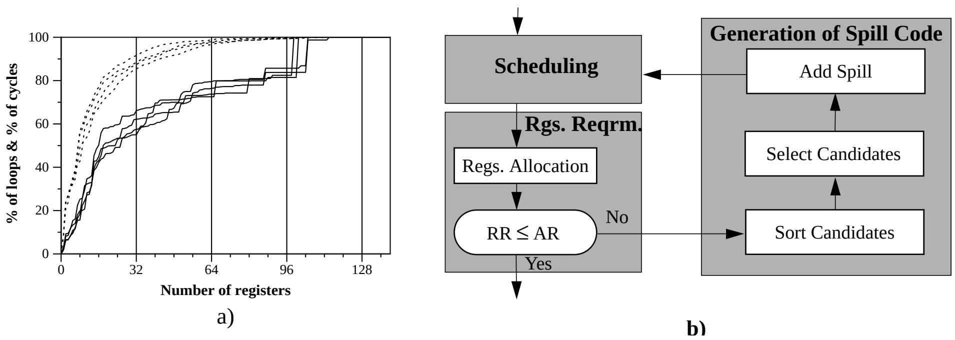

Fig. 1. a) Percentage of loops and percentage of execution cycles that require less than a number of registers for the processor configurations considered in this paper. b) Flow diagram for the two–step spill algorithm.

To the best of our knowledge, all the previous approaches to register spilling for modulo scheduled loops [20, 28, 29, 30], follow a two–step process, as shown in Figure 1.b. In the first step, the loop is scheduled without considering register constraints. Once the loop is scheduled, register allocation is performed over the existing schedule. If the loop requires more registers than those available, spill code is added in the original code and the loop is scheduled again. This second step is repeated until a valid schedule requiring no more registers than available is found. Unfortunately, at a given point in this process it is possible that a different set of lifetimes than those already spilled might need to be spilled.

This paper presents a novel approach for register spilling in modulo scheduled loops. In this approach, instruction scheduling, register allocation, and register spilling are simultaneously in the same step. To achieve this, it uses the ability of some previous iterative modulo scheduling techniques [11, 12, 16, 27] to backtrack, i.e. to undo previous scheduling decisions and reschedule operations. In order to have reasonable low spill code requirements, *MIRS* is based on *HRMS* (Hypernode Reduction Modulo Scheduling [22]), a register–sensitive modulo scheduler. Our proposal is compared with the ideal case (which is an upper bound for performance) and a representative state–of–the–art two–step approach [30].

The paper is organized as follows: Section 2 provides a brief overview of modulo scheduling, register allocation and describes some related work. Then, Section 3 presents the iterative *MIRS* algorithm. Section 4 evaluates and compares the behavior of the proposed algorithm. Finally, Section 5 states our conclusions and outlines some future work.

2 Basic Concepts and Related Work

2.1 Modulo Scheduling

In a software pipelined loop, the schedule of an iteration is divided into stages so that the execution of consecutive iterations, which are in distinct stages, overlaps. The number of stages in one iteration is termed **Stage Count** (*SC*). The number of cycles between the initiation of successive iterations in a software pipelined loop determines its execution rate and is termed the **Initiation Interval** (*II*).

The *II* is bounded either by recurrence circuits in the dependence graph of the loop (*RecMII*) or by resource constraints of the target architecture (*ResMII*). The lower bound on the *II* is termed the **Minimum Initiation Interval** ($MII = max(RecMII, ResMII)$). The reader is referred to [12, 27] for an extensive dissertation on how to calculate *RecMII* and *ResMII*.

The scheduling step of a modulo scheduling technique builds the schedule progressively by adding instructions to a partial schedule. Sometimes the scheduler reaches a partial schedule in which an instruction, requiring a non–saturated resource, cannot be placed. In this case there are two alternative solutions: increasing the *II* [18, 19, 22], or applying backtracking [11, 16, 27]. Backtracking involves unscheduling some instructions in the partial schedule in order to make room for the new instructions. All these unscheduled instructions will be scheduled later. Techniques that include backtracking are also termed *iterative techniques*. In order to avoid an excessive increase in the scheduling complexity, the amount of backtracking is usually limited.

In this paper, a register–sensitive software pipelining technique has been used in order not to overestimate the necessity for spill code. In particular we propose an iterative version of Hypernode Reduction Modulo Scheduling (*HRMS*). When an unbounded number of registers is considered, the original algorithm [22] achieves the *MII* for 97.9% of the workbench considered in this paper. The iterative version proposed in this paper achieves it for 99.0% of the loops. In both implementations, schedules with very low register requirements are generated.

2.2 Register Allocation

Each value in a loop is alive for a certain amount of cycles termed **lifetime** (*LT*). For a loop–variant variable, the *LT* spans from the beginning of the producer operation until the start of the last consumer operation. The *LT* of a variable can be divided into several sections (called **uses**), each spanning from a consumer to the next one. The maximum number of simultaneously live values (*MaxLive*) is an accurate approximation of the number of registers required for the schedule [26]. The cycle where *MaxLive* values are alive is termed the **Critical Cycle**. In our proposal, *MaxLive* is used to estimate the registers required by a partial schedule. Notwithstanding, once the final schedule is found, registers are allocated using wands–only strategy with end–fit and adjacency ordering [26] and additional spill is added if required. This strategy usually achieves a register allocation that uses *MaxLive* registers and almost never requires more than *MaxLive* + 1 registers.

2.3 Adding and Scheduling Spill Operations

Previous approaches to introduce spill code [20, 28, 29, 30] are based on an algorithm similar to the one shown in Figure 1b. After instruction scheduling and register allocation, if the loop requires more registers than those available (required registers $RR >$ available registers AR), a set of spilling candidates is obtained and ordered. The algorithm then decides how many candidates are selected and introduces the necessary memory accesses in the original dependence graph. The loop is rescheduled again because modulo schedules tend to be very compact —the goal is to saturate the most used resource— and it is very difficult to find empty slots to allocate the new memory operations in the schedule. The process is repeated until a schedule, requiring no more registers than those available, is found. *MIRS* avoids the rescheduling of the loop by adding spill code in the partial schedule, where more empty slots are available. In addition it reschedules operations already placed in the partial schedule, when necessary, in order to make room for the spill code.

Some criteria are required in order to assign priorities to the spilling candidates. The priority is usually computed according to the *LT* (the number of cycles that the candidate is alive) [20] or to some ratio between its LT and the memory traffic (*trf*) that would be introduced when spilled (LT/trf) [20, 28, 29]. *MIRS*, like [30], uses (LT/trf) to select only among those uses whose lifetime spans the critical cycle.

After giving priorities to each spilling candidate, the algorithm decides how many candidates are actually spilled to memory. For instance, [20] proposes to spill one candidate at a time. This heuristic avoids overspilling at the expense of a high scheduling time. To avoid this negative aspect, the compiler described in [28] performs several tries by spilling 2^n candidates, where n is increased by one each time. Another alternative, used in [29] consists on selecting as many candidates as necessary to saturate the memory units with the current II, which unfortunately produces an excessive memory traffic. In order to minimize the number of trials, [20] selects as many candidates as necessary so as to reduce, in one step, RR to AR. In order to avoid overspilling, [30] introduces a parameterizable factor, named Quantity Factor, to give more control over this process. Due to the iterative nature of the proposed algorithm, *MIRS* does not require to spill more than a lifetime per iteration; therefore, spill code is introduced in a more accurate way avoiding overspilling.

In order to guarantee that the spill effectively decreases RR, the spilled operations have to be scheduled as close as possible to their producer/consumer operations. This is accomplished by scheduling the spill operation and its associated producer/consumer operation as a single complex operation [20, 30]. On the contrary, *MIRS* is less restrictive and allows to schedule them some cycles apart of the producer/consumer. This feature is controlled by a parameterizable control. When set to its lowest value is equivalent as scheduling the spill operation and its associated producer/consumer as a single complex operation.

3 MIRS: Modulo Scheduling with Integrated Register Spilling

3.1 Definitions

Dependence relations between operations in a loop are usually represented using a dependence graph. This graph is used to schedule the operations so that these dependence relations are honored when the loop is executed in a specific target architecture.

Definition 1: The Dependence Graph $G = DG(VE, \delta, \lambda)$ is a directed graph in which V is the set of vertices and where each vertex $v \in V$ represents an operation of the loop. E is the dependence edge set, where each edge $(u, v) \in E$ represents a dependence between two operations u and v. Edges may correspond to any of the following types of dependences: register dependences, memory dependences or control dependences.

Definition 2: The dependence distance $\delta_{(u,v)}$ is a nonnegative integer associated with each edge $(u, v) \in E$. There is a dependence of distance $\delta_{(u,v)}$ between two nodes u and v if the execution of operation v depends on the execution of operation u , $\delta_{(u,v)}$ iterations before.

Definition 3: The latency λ_u is a nonzero positive integer associated with each node $u \in V$ and is defined as the number of cycles taken by the corresponding operation to produce a result.

The iterative approach presented in this paper first pre-orders the nodes in what we call the *priority_list*. After that the actual iterative scheduling process constructs a *partial_schedule* by scheduling nodes (once at a time) following the order in the *priority_list*. During this iterative process, new nodes may be added to G due to the insertion of spill code. These new store/load nodes will inherit the priority from the associated producer/consumer nodes.

The pre-ordering of nodes is done with the goal of scheduling the loop with an *II* as close as possible to *MII* and using the minimum number of registers. To achieve this, it gives priority to recurrences in order not to stretch any recurrence circuit. It also ensures that, when a node is scheduled, the current partial scheduling contains only *predecessors* or *successors* of the node, but never both (unless the node is the last node of a recurrence circuit to be scheduled).

During the scheduling step, each node is placed on the partial schedule as close as possible to the *neighbor* nodes that have already been scheduled. In order to enable this, the following definitions are useful:

Definition 4: *Early_start* for a node u in graph G is the earliest cycle at which the node can be scheduled such that all its predecessors have completed their execution. It is computed as follows:

$$Early\_start_u = \max_{v \in PSP(u)} t_v + \lambda_v - \delta_{(v,u)} \times II$$

Where t_v is the cycle where v has been scheduled, λ_v is the latency of v, $\delta_{(v,u)}$ is the dependence distance from v to u, and $PSP(u)$ is the set of *predecessors* of u that have been previously scheduled.

Definition 5: *Late_start* for a node u in graph G is the latest cycle at which the node can be scheduled such that it completes its execution before its successors start executing. It is computed as follows:

$$Late\_start_u = \min_{v \in PSS(u)} t_v - \lambda_u + \delta_{(u,v)} \times II$$

Where $PSS(u)$ is the set of *successors* of u that have been previously scheduled.

In order to find a free slot where to schedule each node in G, the current partial schedule is scanned in a specific order. For this purpose, the following situations are possible:

- If operation u has only *predecessors* in *partial_schedule*, then u is scheduled as soon as possible. In this case the scheduler scans starting at cycle $Early\_start_u$ until the cycle $Early\_start_u + II - 1$ is reached.
- If operation u has only *successors* in *partial_schedule*, then u is scheduled as late as possible. In this case the scheduler scans starting at cycle $Late\_start_u$ until the cycle $Late\_start_u - II + 1$ is reached.
- If operation u has *predecessors* and *successors*, then the scheduler scans the partial schedule starting at cycle $Early\_start_u$ until the cycle

$$\min(Late\_start_u, Early\_start_u + II - 1).$$

The next definitions are useful to understand the step that decreases the register requirements. Values used in a loop correspond either to loop–variants or loop–invariants variables. Each loop–invariant variable has only one value for all iterations of the loop. For each loop–variant variable a new value is generated in each iteration of the loop and therefore has a different lifetime. The lifetime of a variable spans from its definition to its last use. The lifetime of a variable can be divided into severals sections (called *uses*) whose lifetime spans from the previous use to the current one. The lifetime section corresponding to latency of the producer functional unit of the variable is call *non–spillable* section.

Definition 6: *MaxLive* is the maximum number of simultaneously live values in the schedule. This is an accurate approximation of the number of registers required by the loop.

Definition 7: *Critical cycle* is the scheduling cycle for which the number of live values is *MaxLive*.

In order to control the iterative nature of the algorithm, the following definitions are useful.

Definition 8: *Budget_ratio* is the number of attempts that the iterative algorithm is allowed to perform per node in G.

Definition 9: *Budget* is the number of attempts that the iterative algorithm can still perform before giving up the current value of *II*. Initially, the *Budget* is set to the product of the number of nodes in G by the *Budget_ratio*. During the iterative algorithm, every time a new node (due to spill code) is inserted in the graph, the *Budget* is increased by *Budget_ratio*. When the number of scheduled nodes exceeds *Budget*, the currently obtained partial schedule is discarded, the *II* is incremented by 1, and the scheduling process is re-started with the original graph.

3.2 Algorithm

Figure 2 summarizes the main steps of the iterative *MIRS* algorithm. The algorithm uses the node ordering strategy defined in the HRMS modulo scheduling algorithm to assign priorities to the nodes in G, although other priority heuristics could be used. For each node i it computes the *Early_start* and *Late_start* values, as well as the search direction between these two values.

With this information, the algorithm tries to find a cycle in the current *partial_schedule* in which the node can be placed without causing any resource conflict or dependence violation. If such a cycle is found, the *partial_schedule* is updated, including resources and registers usage. If the algorithm fails finding a scheduling cycle in the specified cycles, it applies the *Force_and_eject* heuristic which forces the node in a specific cycle and forces some nodes to be ejected due to resource constraints or dependence violations. This heuristic is defined later in this section.

Once the schedule for node i is done, which includes all possible unscheduling actions, the *Check_and_Insert_Spill* heuristic is applied. This heuristic evaluates the necessity of introducing spill code and decides which lifetimes are selected for spilling. After applying spill and inserting new nodes in the dependence graph,

- Step 1. Initialize *II* = *MII*.
- Step 2. Compute priorities for each node in G using pre–ordering step of HRMS.
- Step 3. Set *Budget* = *Budget_ratio* * *nodes(G)*.
- Step 4. Pick node i with the highest priority from priority list; Budget = Budget -1.
- Step 5. Determine *Early_start*, *Late_start* and *Direction* for node i.
- Step 6. Find a *cycle* in *partial_schedule* for node i.
- Step 7. If a valid *cycle* is not found for node i Then apply the *Force_and_Eject* heuristic.
- Step 8. Schedule i and apply the *Check_and_Insert_Spill* heuristic.
- Step 9. If all instructions are scheduled Then goto Step 11.
- Step 10. If (!*Restart_Schedule*) Then goto step 4; Else *II++* and goto Step 3.
- Step 11. Finalize scheduling process.

Fig. 2. Skeleton of the iterative integrated instruction scheduling/register spilling algorithm.

the *Restart_Schedule* heuristic decides if the current *partial_schedule* and value of *II* are still valid to continue with the scheduling process or it is better to restart the process with an increased value for *II*.

The scheduling process finishes in *Step 9* when the algorithm detects that the *priority_list* is empty. At this point, the algorithm performs the real register allocation[1] and generates VLIW code.

Force_and_eject heuristic When the scheduler fails in finding a cycle where to schedule a node, it forces the node at a specific cycle given by:

$$forced\_cycle = max(Early\_start, (Prev\_cycle(i) + 1))$$

if the search direction is from *Early_start* to *Late_start*, or

$$forced\_cycle = min(Late\_start, (Prev\_cycle(i) - 1))$$

otherwise. In both expressions, *Prev_cycle(i)* is the cycle at which the node was scheduled in a previous partial schedule (before a possible ejection).

When forcing a node in a particular cycle, the heuristic ejects nodes that cause resource conflicts with the forced node. If for a particular resource conflict several candidate nodes are possible, the heuristic selects the one that was first placed in the *partial_schedule*. Other iterative algorithms [11, 16, 27] eject all the operations that cause each resource conflict. In our iterative algorithm, the minimum number of operations are ejected. The heuristic also ejects all previously scheduled predecessors and successors whose dependence constraints are violated due to the forced placement.

Notice that all the unscheduled operations are put back in the *priority_list* with their original priority. Therefore, they will be immediately picked up by the iterative algorithm for rescheduling.

[1] Since *MaxLive* is an approximation of the actual register requirements, sometimes it is needed to insert some additional spill code to ensure that the *RR* fit in *AR*, as mentioned in Section 2.2.

Check_and_Insert_Spill heuristic This heuristic first compares the actual number of registers used in the current *partial_schedule* (RR) and the total number of registers available (AR) and decides whether to insert spill code or proceed with the next node in *priority_list*. In our implementation, spill code is introduced whenever $RR > SG \times AR$, SG being the *spill_gauge*. SG may take any positive value larger or equal than one. If set to one, it means that the algorithm adds spill code as soon as the register limit is reached. When set to a very large value it causes the algorithm to perform spill after obtaining a partial schedule for all the nodes in the dependence graph. The effects of intermediate values for this parameter on the quality of the schedule are discussed in an extended version of this paper [31].

In order to efficiently reduce the register requirements, the spill heuristic tries to select the use, among those that cross the *critical_cycle* in the *partial_schedule*, that has the largest ratio between its lifetime and the memory traffic its spilling would generate (number of load and store operations to be inserted). If such a use is not found or it does not spans a minimum number of cycles, one of the nodes already scheduled in the *critical_cycle* is ejected and placed back in the *priority_list*. This forces a reduction of the register requirements in the *critical_cycle* by moving the non–spillable section of the lifetime outside this cycle. The minimum number of cycles that the selected lifetime has to span (named *minimum_span_gauge MSG*) is another parameter of our algorithm that influences the quality of the schedules generated and it is experimentally evaluated in [31].

For the lifetime selected, spill load/store nodes are inserted in the dependence graph. These operations are also inserted in the *priority_list* with the priority of their associated consumer/producer nodes minus one. In addition, these nodes are forced to be placed as close as possible to their associated consumer/producer nodes. To achieve this, the *Early_start* of an spill load node is set to its *Late_start* - *DG* and the *Late_start* of an spill store node is set to its *Early_start* + *DG*, *DG* being the *distance_gauge*. The influence of this gauge is also discussedin the extended version of this paper [31].

Once spill nodes are inserted in the dependence graph, the *Budget* is increased by the number of nodes inserted times the *Budge_ratio* in order to give further chances to the iterative algorithm to complete the schedule.

Restart_Schedule heuristic The iterative algorithm discards the current partial schedule in two cases: 1) if the number of trials is exhausted (*Budget* reaches 0) and 2) if the processor configuration with the current value of *II* can not support the memory traffic generated due to the newly inserted spill operations. After inserting a set of spill memory operations, it may be better to directly increase *II* instead of continuing trying with its current value. In order to forestall this situation, the heuristic estimates the memory traffic (number of load and store instructions) due to the newly introduced spill code (*NewTrf*). If the maximum traffic (*MaxTrf*) that can be supported with the current value of *II* is not enough to absorb *NewTrf*, then the algorithm directly increases the *II* and discards the current *partial_schedule*. In both situations, the algorithm is restarted with a larger value for *II*. If both tests are passed, the algorithm proceeds with the next node in the *priority_list*.

4 Performance evaluation

The effectiveness of *MIRS* is statically evaluated to find out the number of loops for which it obtains better schedules than the proposal presented in [30] (from now on named *2–STEP*). After that, a dynamic evaluation is performed in order to show its effectiveness.

The evaluation is done using a set of statically scheduled processor configurations *PiMjLk* defined as follows: i is the number of functional units used to perform each type of computation (adders, multipliers and div/sqr units); j is the number of load/store units; and k is the latency of the adders and the multipliers. In all configurations, the latencies of load and store accesses are two cycles and one cycle, respectively. Divisions take 17 cycles and square roots take 30 cycles. All functional units are fully pipelined, except for the div/sqr functional units. In particular, four different configurations are used: *P2M2L4*, *P2M2L6*, *P4M2L4* and *P4M4L4*, with 32 registers.

For the evaluation we use all the innermost loops of the Perfect Club benchmark suite [5] that have neither subroutine calls nor conditional exits. Loops with conditional statements have been previously IF-converted [3] so that they behave as a single basic block loop. A total of 1258 loops, that represent about 80% of the total execution time of the benchmark have been scheduled.

4.1 Static Evaluation

The static metrics used to evaluate the performance are the following: ΣII, which measures the sum of the individual *II* for the loops considered; and Σtrf, which measures the sum of the individual number of memory operations used in the scheduling for the loops considered.

Table 1 compares *MIRS* against *2–STEP* in terms of the above mentioned metrics. Columns on the left, under label "Loops with Spill" show the comparison considering only the loops that require spill code. For instance, for processor configuration *P4M2L4*, *MIRS* improves the *II* metric in 61 loops, achieves the same value in 74 loops and produces a worse result for 14 loops. In total, the ΣII is reduced from 3045 to 2670. In terms of memory traffic, *MIRS* reduces *trf* in 69 loops, achieves the same value in 42 loops and produces more *trf* in 38 loops. In total, the Σtrf is reduced from 5185 to 4555. The two columns under label "All Loops" show the comparison considering all the 1258 loops in the benchmarks. Notice that, when only considering the loops that require spill the ΣII is reduced by 375 cycles and when considering all loops it is reduced by 411 cycles. This extra benefit is caused by the iterative nature of the scheduling algorithm which improves the *II* in loops with complex operations. Similar results are obtained for the other processor configurations, showing a natural tendency to produce

Table 1. Static evaluation.

	Loops with Spill								All Loops	
	$II_{MIRS} < II_{2\text{-}STEP}$		$II_{MIRS} = II_{2\text{-}STEP}$		$II_{MIRS} > II_{2\text{-}STEP}$		Total		Total	
Item	2-STEP	MIRS	2-STEP	MIRS	2-STEP	MIRS	2-STEP	MIRS	2-STEP	MIRS
P4M2L4	61		74		14		149		1258	
ΣII	1902	1505	794	794	349	371	3045	2670	8152	7741
Σtrf	3272	2679	1286	1256	627	620	5185	4555	10269	9639
$< trf$	21	31	9	33	8	5	38	69	38	69
P4M4L4	50		113		19		182		1258	
ΣII	1178	973	950	950	372	398	2500	2321	6838	6639
Σtrf	3707	3208	2502	2502	1291	1198	7695	6908	12381	11594
$< trf$	20	27	18	56	5	14	43	97	43	97
P2M2L4	42		58		9		109		1258	
ΣII	1836	1565	1089	1089	197	216	3122	2870	9398	9102
Σtrf	2799	2552	1283	1361	329	341	4411	4254	9907	9750
$< trf$	20	16	20	14	4	3	44	33	44	33
P2M2L6	68		80		10		158		1258	
ΣII	2581	2206	1154	1154	256	270	3991	3630	10659	10241
Σtrf	4125	3620	1613	1572	465	473	6203	5665	11152	10614
$< trf$	16	40	21	28	3	6	40	74	40	74

schedules that improve both the execution rate and reduce the memory traffic. Only in a small number of loops, the algorithm produces schedules with both worse *II* and *trf* values.

4.2 Dynamic Evaluation

In order to dynamically compare the two algorithms, the two following metrics are used:

- Ratio between the ideal number of execution cycles needed to execute the loops assuming an unbounded number of registers and the actual execution cycles needed to execute them once scheduled with one of the heuristics. The execution cycles are estimated as $II * (N + (SC - 1) * E)$, N being the total number of iterations and E the number of times the loop is executed. The values of N and E are obtained by profiling.
- Ratio between the memory traffic caused by the loops when scheduled with one of the heuristics and the ideal memory traffic when an unbounded number of registers is available. In both cases, the traffic is estimated as $M * N$, M being the number of memory operations in the kernel code of the software pipelined loop. Although, a memory traffic reduction is not directly observable in performance, it reduces the instruction fetch bandwidth requirements for spill operations and the memory bandwidth requirements. This two factors can influence power consumption, which is a major concern in current state-of-the-art processor designs.

Figure 3.a-c shows the above mentioned ratios and the speed–up for *MIRS* and *2–STEP* for the four different processor configurations considering the loops

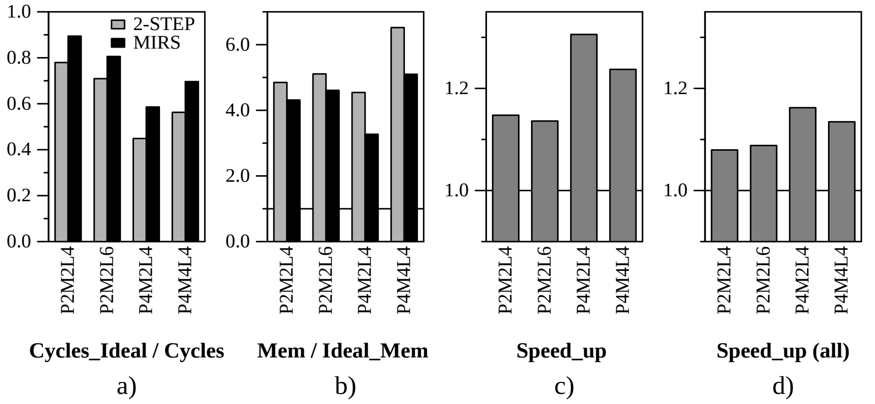

Fig. 3. Dynamic evaluation of relative execution cycles, memory traffic and speed–up for four processor configurations *P2M2L4*, *P2M2L6*, *P4M2L4* and *P4M4L4*.

that require spill code. On the average, the reduction in execution cycles represents an speed–up of 15%, 14%, 31% and 24% in those loops for for processor configurations *P2M2L4*, *P2M2L6*, *P4M2L4* and *P4M4L4*, respectively. Memory traffic is reduced by factors 0.89, 0.90 0.72 and 0.78, respectively. As shown in Figure 3.d, for the whole benchmark set, the reduction in execution cycles represents a speed–up of 7.9%, 8.8%, 16.2% and 13.4% respectively.

Figure 4 shows the above mentioned ratios for each program in the Perfect Club for processor configuration *P4M2L4*. Notice that for 5 applications (CSSI, LGSI, MTSI, SDSI and TISI) *2–STEP* already reaches the optimal execution time and memory traffic, and therefore no speed-up at all can be obtained. For 5 more applications (APSI, LWSI, SRSI, TFSI and WSSI) *2–STEP* is very close to the optimal and only small speed-ups (up to 3%) can be obtained with the new approach. However, there are three applications for which substantial improvements are obtained. For NASI and SMSI, whose memory traffic due to spill code is high, the proposed algorithm achieves a speed–up of 26% and 7.8% respectively. In the case of OCSI the improvement is not due to the reduction in spill code but to the iterative nature of the scheduling algorithm which favors applications with complex operations. In OCSI these kind of complex operations limit the performance of some loops and the application of an iterative algorithm produces a speed-up of 13.5%.

4.3 Scheduling Time

MIRS takes between 15 and 20 minutes to schedule all 1258 loops, which is about 5 times the time required by *2–STEP*. However, this is still a very reasonable scheduling time for 13 applications. We have observed that *MIRS* spends almost 84% of this time on only three of the loops. These loops are extremely large, initially require more than 100 registers and the *II* has to be increased a large number of times (i.e. the scheduler has to be restarted 40–70 times) to reach the

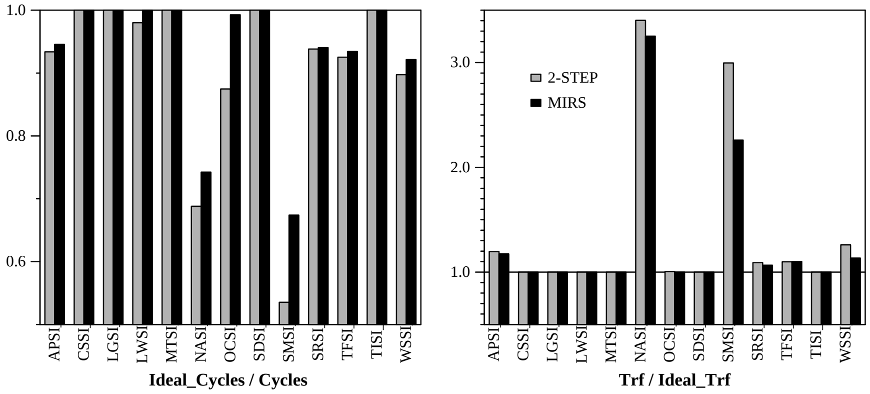

Fig. 4. Dynamic results for each program in the Perfect Club and configuration *P4M2L4*. Light bars correspond to the *2–STEP* algorithm and dark bars to *MIRS*.

available number of registers. An heuristic to estimate a tight lower bound of the final *II* could dramatically reduce the scheduling time.

5 Conclusions

Software pipelining is a very effective technique for scheduling loops that produces schedules with high register requirements. When the number of registers required exceed the number of registers available in the target architecture, some of them must be spilt to memory. Unfortunately the addition of spill code degrades the performance of the schedule. On the one hand, the additional spill code (performed through store/load operations) increases the usage of the memory ports. If they became saturated, the initiation interval of the loop must be increased to accommodate the additional operations, producing a direct degradation of performance. On the other hand, even if the initiation interval of the loop is not affected by the additional memory traffic, it produces some negative side effects: 1) more instruction fetch bandwidth is required for the additional load/store operations, which may hurt performance indirectly; 2) the additional memory traffic may interfere in the cache with the regular traffic and increases miss ratio hurting also performance; and 3) the additional memory accesses and fetch bandwidth increase the overall power consumption.

Previous proposals for register spilling in software pipelined loops are based in a two–step process. First a schedule with no register constraints is generated. Then, registers are allocated and spill code is added if required. The addition of spill code requires rescheduling the loop. This process may be repeated several times since more spill might be required after the reschedule.

In this paper we have presented *MIRS*, an iterative Modulo scheduling approach with Integrated Register Spilling. *MIRS* schedules the loops, allocates registers and adds the required spill code all in a single step. It is based on the ability of iterative modulo scheduling to undo previous scheduling decisions, eject operations and leave room to schedule the spill code.

Performance figures show that *MIRS* obtains an average speed–up ranging from 14% to 31% (depending on the processor configuration) for all the loops that require spill code. This represents an average speed–up from 7.9% to 16.2% for the whole benchmark set. However for particular applications speed-ups up to 26% are obtained. In addition, noticeable traffic reductions (with all the implications mentioned above) are also obtained.

References

[1] A. Aiken and A. Nicolau. A realistic resource-constrained software pipelining algorithm. *Advances in Languages and Compilers for Parallel Processing*, pages 274–290, 1991.

[2] V. Allan, R. Jones, R. Lee, and S. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, September 1995.

[3] J. Allen, K. Kennedy, and J. Warren. Conversion of control dependence to data dependence. In *Proc. 10th annual Symposium on Principles of Programming Languages*, January 1983.

[4] D. Bernstein, D. Goldin, M. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R. Pinter. Spill code minimization techniques for optimizing compilers. In *Proc. of the ACM SIGPLAN'89 Conf. on Programming Languages Design and Implementation*, pages 258–263, July 1989.

[5] M. Berry, D. Chen, P. Koss, and D. Kuck. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. Technical Report 827, Center for Supercomputing Research and Development, November 1988.

[6] P. Briggs, K. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proc. of the ACM SIGPLAN'89 Conf. on Programming Language Design and Implementation*, pages 275–284, June 1989.

[7] P. Briggs, K. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[8] D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. In *Proc. of the ACM SIGPLAN'91 Conf. on Programming Language Design and Implementation*, pages 192–203, June 1991.

[9] G. Chaitin. Register allocation and spilling via graph coloring. In *Proc. ACM SIGPLAN Symp. on Compiler Construction*, pages 98–105, June 1982.

[10] A. Charlesworth. An approach to scientific array processing: The architectural design of the AP120B/FPS-164 family. *Computer*, 14(9):18–27, 1981.

[11] A. K. Dani, V. J. Ramanan, and R. Govindarajan. Register-sensitive software pipelining. In *Procs. of the Merged 12th International Parallel Processing and 9th International Symposium on Parallel and Distributed Systems*, april 1998.

[12] J. Dehnert and R. Towle. Compiling for the Cydra 5. *The Journal of Supercomputing*, 7(1/2):181–228, May 1993.

[13] A. Eichenberger and E. Davidson. Stage scheduling: A technique to reduce the register requirements of a modulo schedule. In *Proc. of the 28th Annual Int. Symp. on Microarchitecture (MICRO-28)*, pages 338–349, November 1995.

[14] C. Eisenbeis, S. Lelait, and B. Marmol. The meeting graph: a new model for loop cyclic register allocation. In *Proc. of the Fifth Workshop on Compilers for Parallel Computers (CPC95)*, pages 503–516, June 1995.

[15] L. Hendren, G. Gao, E. Altman, and C. Mukerji. Register allocation using cyclic interval graphs: A new approach to an old problem. ACAPS Tech. Memo 33, Advanced Computer Architecture and Program Structures Group, McGill University, 1992.

[16] R. Huff. Lifetime-sensitive modulo scheduling. In *Proc. of the 6th Conference on Programming Language, Design and Implementation*, pages 258–267, 1993.

[17] S. Jain. Circular scheduling: A new technique to perform software pipelining. In *Proc. of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 219–228, June 1991.

[18] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[19] J. Llosa, A. González, E. Ayguadé, and M. Valero. Swing modulo scheduling: A lifetime-sensitive approach. In *IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques (PACT'96)*, pages 80–86, October 1996.

[20] J. Llosa, M. Valero, and E. Ayguadé. Heuristics for register-constrained software pipelining. In *Proc. of the 29th Annual Int. Symp. on Microarchitecture (MICRO-29)*, pages 250–261, December 1996.

[21] J. Llosa, M. Valero, and E. Ayguadé. Quantitative evaluation of register pressure on software pipelined loops. *International Journal of Parallel Programming*, 26(2):121–142, April 1998.

[22] J. Llosa, M. Valero, E. Ayguadé, and A. González. Hypernode reduction modulo scheduling. In *Proc. of the 28th Annual Int. Symp. on Microarchitecture (MICRO-28)*, pages 350–360, November 1995.

[23] W. Mangione-Smith, S. Abraham, and E. Davidson. Register requirements of pipelined processors. In *Proc. of the Int. Conference on Supercomputing*, pages 260–246, July 1992.

[24] S. Ramakrishnan. Software pipelining in PA–RISC compilers. *Hewlett-Packard Journal*, pages 39–45, July 1992.

[25] B. Rau and C. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc. of the 14th Annual Microprogramming Workshop*, pages 183–197, October 1981.

[26] B. Rau, M. Lee, P. Tirumalai, and P. Schlansker. Register allocation for software pipelined loops. In *Proc. of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 283–299, June 1992.

[27] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, November 1994.

[28] J. Ruttenberg, G. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. In *Proc. of the ACM SIGPLAN'96 Conf. on Programming Languages Design and Implementation*, pages 1–11, May 1996.

[29] J. Wang, A. Krall, M. A. Ertl, and C. Eisenbeis. Software pipelining with register allocation and spilling. In *Proc. of the 27th Annual Int. Symp. on Microarchitecture*, pages 95–99, November 1994.

[30] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Improved spill code generation for software pipelined loops. In *Procs. of the Programming Languages Design and Implementation (PLDI'00)*, pages 134–144., June 2000.

[31] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. *MIRS*: Modulo scheduling with integrated register spilling. Technical Report UPC-DAC-2000-68, Universitat Politècnica de Catalunya, November 2000.