Register Allocation and Optimal Spill Code Scheduling in Software Pipelined Loops Using 0-1 Integer Linear Programming Formulation

Santosh G. Nagarakatte[1] and R. Govindarajan[1,2]

[1] Department of Computer Science and Automation,
[2] Supercomputer Education and Research Center,
Indian Institute of Science, Bangalore 560012, India
{santosh,govind}@csa.iisc.ernet.in

Abstract. In achieving higher instruction level parallelism, software pipelining increases the register pressure in the loop. The usefulness of the generated schedule may be restricted to cases where the register pressure is less than the available number of registers. Spill instructions need to be introduced otherwise. But scheduling these spill instructions in the compact schedule is a difficult task. Several heuristics have been proposed to schedule spill code. These heuristics may generate more spill code than necessary, and scheduling them may necessitate increasing the initiation interval.

We model the problem of register allocation with spill code generation and scheduling in software pipelined loops as a 0-1 integer linear program. The formulation minimizes the increase in initiation interval (II) by optimally placing spill code and simultaneously minimizes the amount of spill code produced. To the best of our knowledge, this is the first integrated formulation for register allocation, optimal spill code generation and scheduling for software pipelined loops. The proposed formulation performs better than the existing heuristics by preventing an increase in II in 11.11% of the loops and generating 18.48% less spill code on average among the loops extracted from Perfect Club and SPEC benchmarks with a moderate increase in compilation time.

1 Introduction

Software pipelining [14] is the most commonly used loop scheduling technique for exploiting higher instruction level parallelism. In a software pipelined loop, instructions from multiple iterations are executed in an overlapped manner. Several heuristic methods [2,19] have been proposed to construct a software pipelined schedule. In addition a number of methods [10] have also been proposed to find an optimal schedule considering resource constraints. A schedule is said to be optimal if the initiation interval (II) of the schedule is not greater than that of any other schedule for the loop with the given resource constraints.

Software pipelining, like other instruction scheduling techniques, increases the register pressure. A number of heuristic approaches to reduce the register pressure

S. Krishnamurthi and M. Odersky (Eds.): CC 2007, LNCS 4420, pp. 126–140, 2007.

of the software pipelined schedule have been proposed [11]. Also, approaches to minimize the register pressure of the software pipelined schedule using linear [16] and integer linear program formulation have been reported in literature. However, these methods do not guarantee that the register requirements of the constructed schedule is less than the available registers. If the register need of the constructed schedule is greater than the available number of registers, either spill code needs to be introduced or the initiation interval needs to be increased [21]. In order to determine whether the constructed schedule is feasible for the given number of registers, register allocation must be performed with necessary spill code generation. Further the spill code must be scheduled in the compact schedule, without violating any resource or dependence constraints. Currently heuristic approaches [21] have been proposed for the introduction of spill code. Unfortunately, introduction of spill code can saturate the memory units and thereby force an increase in the initiation interval.

In this paper, we are interested in addressing the following problem: Given a modulo scheduled loop L, a machine architecture M and an initiation interval II, is it possible to perform register allocation with the given registers and optimally generate and schedule necessary spill code such that the register requirement of the schedule is lesser than or equal to the available number of registers? We propose a 0-1 integer linear programming formulation for register allocation, optimal spill code generation and spill code placement in software pipelined loops. The proposed approach is guaranteed to identify a schedule with necessary spill code, whenever such a schedule exists, without increasing the initiation interval. Further the proposed approach generates minimal spill code, thereby improving the code quality. The proposed formulation takes into account both the compactness of the schedule and memory unit usage. Further the formulation incorporates live range splitting [4] which allows a live range to be assigned to a register at specific time instances and be resident in memory in rest of the time instances. To the best of our knowledge, this is the first integrated formulation for register allocation, optimal spill code generation and scheduling for software pipelined loops. The formulation is useful in evaluating various heuristics and one can generate a better quality code with a moderate increase in compilation time. We have implemented the solution method on loops from Perfect Club and SPEC2000 benchmarks. On an average, we prevent an increase in the initiation interval in 11.11% of the 90 loops on an architecture with 32 registers and in 12% of the 157 loops on an architecture with 16 registers when compared to the heuristic approach [21]. We also generate roughly 18.48% less spill code compared to the heuristic solution.

The paper is organized as follows: Section 2 provides a brief motivation for optimal spill code generation and scheduling. In Section 3, we explain our integer linear programming formulation. Section 4 presents the simplified formulation. Section 5 presents the experimental methodology and results. In Section 6, we discuss the related work and concluding remarks are provided in Section 7.

2 Motivation

Traditionally, the process of adding spill code is done iteratively [21] for architectures with no rotating registers. First, the loop is modulo scheduled, then register allocation is performed. If the register pressure of the schedule is greater than the available number of registers, then spill candidates are chosen. Subsequently spill code is added and the loop is rescheduled. In the process above, since the selection of spill candidates is based on a certain heuristic, it may result either in the addition of extra spill code or the introduction of spill code at a time step where no memory unit is available. These, in turn, may increase the memory unit usage necessitating an increase in the initiation interval. Various heuristics have been proposed for generating spill code and scheduling spill code [1].

Critical cycle is one of the key characteristics used by heuristics to decide on the spill candidates. A time step t is said to be a *Critical cycle* in the kernel if the number of live ranges at that instant is greater than the number of available registers. In Figure 1(a), we show the live ranges of a software pipelined schedule with $II = 6$ and assume there are four registers available. For this schedule, cycle 2 is the critical cycle. To perform register allocation with the available four registers for the given schedule, one of the live ranges must be spilled. A commonly used heuristic gives priority to the spill candidate with longest live range [21]. Unfortunately, it is possible that the longest live range does not span through critical cycle. Hence, spilling the longest live range may not necessarily reduce the register pressure. A refined heuristic considering the above prioritizes the spill candidate which is live at the critical cycle and has the longest lifetime among the the spill candidates [21]. The heuristics may not be able to capture all the scenarios.

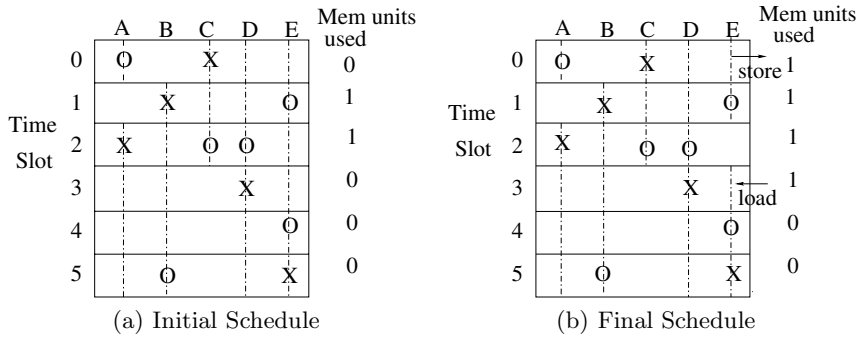


Fig. 1. Initial kernel with II = 6. X is the definition and O is the use of the live range.

Consider the kernel shown in Figure 1(a). In this example, we have assumed a load and a store latency of 1 cycle and the presence of a single memory unit and 4 registers. The memory unit usage in the kernel is indicated in the figure. The kernel is obtained for an initiation interval of 6. The register need of the schedule

is 5. So we need to insert spills in order to reduce register need. Figure 1(b) shows the kernel after the spill code has been scheduled. Among the spill candidates, variables D and E have the longest live range and pass through the critical cycle 2. In the kernel in Figure 1(b), though the spill store for E is scheduled at cycle 0, the value in the register continues and ends only at cycle 1. If we had chosen D as the spill candidate, we would not have been able to spill and hence reduce the register pressure at cycle 2. This is because of the use of D in cycle 2. As a result, it is not only necessary to select the right spill candidate but also to schedule the spill loads and stores so that the register need of the loop is reduced without unnecessarily requiring an increase in the initiation interval.

The recent work in spill code generation [21] addresses the iterative process of adding spill code by selecting a finite number of candidates for spilling based on a *quantity factor* which is determined experimentally. By adopting the notion of quantity factor, we are making the decision of selecting the spill candidate and scheduling them incrementally, considering a few candidates. It is possible that the greedy approach can fail. In our experimentation, the quantity factor of 0.5 resulted in an increase in the initiation interval in 12% of the loops that had sufficent register pressure and needed the addition of spill code.

Moreover, there are a plethora of factors that need to be considered while choosing the right spill candidate which can be suitably scheduled with a minimal amount of spill code. An injudicious selection and subsequent scheduling can result in an unnecessary increase in the initiation interval, which can be attributed to addition of otherwise superfluous spill code saturating the memory usage.

3 ILP Formulation for Spill Code Minimization and Scheduling

In this section, we explain our 0-1 integer linear programming formulation for register allocation and spill code scheduling in software pipelined loops assuming a load-store architecture with no rotating registers. A solution to the ILP formulation would represent a valid schedule with spill code suitably scheduled satisfying the register and functional resource constraints. Given a software pipelined loop with modulo variable expansion [14] carried out, our efficient register allocation and spill code scheduling formulation involves the association of decision variables to the live range, formulation of relationship between the decision variables that need to be satisfied, solving the integer linear program and rewriting the original code.

3.1 Generation of Decision Variables

Given a data dependence graph and a periodic schedule, we model a live range with a set of decision variables. The live range produced by instruction i is denoted by the temporary name TN_i. Without the loss of generality, we use the term temporary variable and live range interchangeably as each temporary

variable has exactly one definition point. The live range TN_i is represented with a series of liveness decision variables from its definition time (T_i^{def}) to its last use time (T_i^{end}). A live range can be allocated to any of the R registers. Hence corresponding to each time instant $t \in [T_i^{def}, T_i^{end}]$ and register r, we create liveness decision variables of the form $TN_{i,r,t}$. The decision variable $TN_{i,r,t} = 1$ represents the fact that the TN_i is allocated to register r at time instant t.

To determine where to introduce spill stores and loads in the schedule, we introduce two kinds of spill decision variables namely store decision and load decision variables.

1. Store decision variable: We introduce store decision variables $STN_{i,r,t}$ for every live range TN_i, for register r and time t. The store decision variable $STN_{i,r,t} = 1$ implies that there is a spill store of the live range TN_i in register r at time instant t. The store decision variable is defined only for a subset of the time steps in the kernel. More specifically, it is defined only for time step $t \in [T_i^{def} \oplus lat_i, T_i^{end} \ominus lat_{store} \ominus lat_{load}]$ where lat_i, lat_{store} and lat_{load} are latencies of instruction i, store and load respectively. This is because the spill store can be scheduled only after $T_i^{def} \oplus lat_i$. Further the spill store must be scheduled $lat_{store} + lat_{load}$ cycles before the last use. Since all time steps should be within [0, II -1], the add and subtract operations are performed modulo II and represented as $\oplus$ and $\ominus$ respectively. The store decision variable $STN_{i,r,t}$ is defined for time steps $t \in storeset(i)$ where $storeset(i) = [\ T_i^{def} \oplus lat_i, T_i^{end} \ominus lat_{load} \ominus lat_{store}]$.
2. Load decision variable: We introduce load decision variable $LTN_{i,r,t}$ for every live range TN_i, register r, and time step t. The load decision variable $LTN_{i,r,t} = 1$ implies that there is a spill load of the live range TN_i scheduled at time instant t. The load decision variable $LTN_{i,r,t}$ is defined for time steps $t \in loadset(i)$ where $loadset(i) = [\ T_i^{def} \oplus lat_i \oplus lat_{store}, T_i^{end} \ominus lat_{load}\ ]$.

We illustrate the introduction of live range and spill decision variables with a specific example in Figure 2. An instruction which defines the value of a temporary variable TN_1 is scheduled at time 0. The last use of TN_1 is scheduled at time 9. The liveness, spill load and store decision variables introduced corresponding to register R0 are shown in Figure 2. In this example, the latency of the instruction producing the live range TN_1 is 1, and that of store or load is 2. To represent whether the live range TN_1 is live in register R0 at various time steps during its live range, we use decision variables $TN_{1,0,0}$, ... $TN_{1,0,9}$. The store decision variables are defined for time steps [1, 5]. We do not define the store decision variable at time instant 0 since it is the definition time. Similarly the store decision variable is not defined for time steps [6, 9] as splitting the live range beyond time step 5 does not result in a meaningful spill load to be scheduled before the last use of TN_1. Similarly we do not create spill load decision variables at time steps [0, 2], since spill store would not have completed by that time, and at time steps [8, 9], as the spill load would not complete before the last use at 9.

Time		Decision variables for register R0		
0	TN_1 =	$TN_{1,0,0}$		
1		$TN_{1,0,1}$	$STN_{1,0,1}$	
2		$TN_{1,0,2}$	$STN_{1,0,2}$	
3		$TN_{1,0,3}$	$STN_{1,0,3}$	$LTN_{1,0,3}$
4		$TN_{1,0,4}$	$STN_{1,0,4}$	$LTN_{1,0,4}$
5	= .. op TN_1	$TN_{1,0,5}$	$STN_{1,0,5}$	$LTN_{1,0,5}$
6		$TN_{1,0,6}$		$LTN_{1,0,6}$
7		$TN_{1,0,7}$		$LTN_{1,0,7}$
8		$TN_{1,0,8}$		
9	=.. op TN_1	$TN_{1,0,9}$		

Fig. 2. Decision variables associated with live range TN_1 and register 0 with an II=10

3.2 Constraints

Having discussed the liveness, spill store and spill load decision variables corresponding to each time instant and register, we now explain how register allocation and spill code scheduling can be formulated using a set of constraints. Satisfaction of these constraints results in a schedule with valid register allocation and appropriate spill code placement.

Must-Allocate Definition Constraint: The Must-Allocate Definition Constraints ensure that a register is allocated to a live range when the live range is defined. That is, for each instruction that produces a value, a register must be allocated to the live range. If I is the set of instructions that produce a result value and TN_i be the temporary variable corresponding to instruction $i \in I$, the following must-allocate definition constraint must be satisfied.

$$\sum_{r \in R} TN_{i,r,t} = 1 \qquad \forall i \in I \; and \; t = T_i^{def} \tag{1}$$

There are exactly $|I|$ constraints produced by the above equation. For the example shown in Figure 2, corresponding to TN_1, the following must-allocate definition constraint must be satisfied.

$$\sum_{r \in R} TN_{1,r,0} = 1$$

Must-Allocate Use Constraint: Must-Allocate Use Constraints ensure that a live range is in a register at the time instant where there is an use. Let use(TN_i) represent the set of instructions that use the temporary variable TN_i produced

by instruction i. The live range TN_i must be available in a register at time instant t corresponding to its use since we assume a load-store architecture.

For each instruction j $\in use(TN_i)$, scheduled at time instant t,

$$\sum_{r \in R} TN_{i,r,t} - \sum_{r,t'} LTN_{i,r,t'} \geq 1 \quad for\ all\ t = T_j^{def}\ and\ j \in use(TN_i) \tag{2}$$

where $t' \in (t \ominus lat_{load},\ t]$. There are exactly $\sum_{i \in I} |use(TN_i)|$ constraints corresponding to the above equation. We refer to these as must-allocate use constraints.

For the example shown in Figure 2, corresponding to TN_1, the following must-allocate use constraints must be satisfied.

$$\sum_{r \in R} TN_{1,r,5} - \sum_{r \in R} (LTN_{1,r,4} + LTN_{1,r,5}) \geq 1; \quad \sum_{r \in R} TN_{1,r,9} \geq 1$$

At-most Single Store Constraints: The live range TN_i need to be stored at-most once. For every instruction $i \in I$, at-most one store constraint is given by

$$\sum_{t} \sum_{r \in R} STN_{i,r,t} \leq 1 \tag{3}$$

where t is in the range $[(T_i^{def} \oplus lat_i),\ (T_i^{end} \ominus lat_{load} \ominus lat_{store})]$.

As the objective minimizes the spill loads and stores, this constraint is redundant. However, this constraint reduced the solution time taken by the ILP solver.

Store Before Load Constraints: A spill load can be scheduled for a live range provided there is an earlier spill store for that temporary name. At every time instant where a spill load is possible, there must be a store which has been scheduled earlier. For every spill load corresponding to live range TN_i, the following constraints must be satisfied.

$$\sum_{r} LTN_{i,r,t} \leq \sum_{r} \sum_{t'} STN_{i,r,t'} \qquad \forall t \in loadset(i) \tag{4}$$

where t' is in the range $[(T_i^{def} \oplus lat_i),\ (t \ominus lat_{store})]$. There are exactly $|loadset(i)|$ such constraints for each TN_i

In Figure 2, each of the spill loads corresponding to time steps [3, 7] must satisfy the following constraints. We have assumed a store latency of 2.

$$\sum_{r \in R} LTN_{1,r,3} \leq \sum_{r \in R} STN_{1,r,1}$$

$$\sum_{r \in R} LTN_{1,r,4} \leq \sum_{r \in R} (STN_{1,r,1} + STN_{1,r,2})$$

$$\sum_{r \in R} LTN_{1,r,5} \leq \sum_{r \in R} (STN_{1,r,1} + STN_{1,r,2} + STN_{1,r,3})$$

$$\sum_{r \in R} LTN_{1,r,6} \leq \sum_{r \in R} (STN_{1,r,1} + STN_{1,r,2} + STN_{1,r,3} + STN_{1,r,4})$$

$$\sum_{r \in R} LTN_{1,r,7} \leq \sum_{r \in R} (STN_{1,r,1} + STN_{1,r,2} + STN_{1,r,3} + STN_{1,r,4} + STN_{1,r,5})$$

Spill Load Store Constraints: In order to schedule spill code in the compact schedule, we have introduced store and load decision variables at multiple time instants. The following set of constraints ensure that there are no unnecessary spill code instructions and formulation generated schedule is valid.

At each time instant t for any live range, if $t \in loadset(i)$ and $t \in storeset(i)$, then the store before load and at-most only one store constraints ensure that both load and store cannot be scheduled at t. For each store decision variable at time t corresponding to live range TN_i, a store can actually take place at that instant only if the variable is in the register.

$$STN_{i,r,t} \leq TN_{i,r,t} \quad \forall r \in R \; and \; \forall t \in \; storeset(i) \tag{5}$$

In Figure 2, the following constraints corresponding to store of live range TN_1 in register 0, at time steps [1, 5] must be satisfied.

$$STN_{1,0,1} \leq TN_{1,0,1}; \quad STN_{1,0,2} \leq TN_{1,0,2}; \quad STN_{1,0,3} \leq TN_{1,0,3};$$

$$STN_{1,0,4} \leq TN_{1,0,4}; \quad STN_{1,0,5} \leq TN_{1,0,5};$$

After a spill store, the live range in a register may continue to exist or cease to exist. But if there is a load in the subsequent time instant, then the load constraints can bring the live range back into existence in the register. If a spill store is possible for live range TN_i at time instant t and spill load is not possible at time instant $t + 1$, then the following constraints need to be satisfied.

$$TN_{i,r,t \oplus 1} \leq TN_{i,r,t} \quad \forall r \in R, for \; all \; t \in storeset(i) \; and \; t \oplus 1 \notin loadset(i) \tag{6}$$

In Figure 2, the following constraints must be satisfied corresponding to the live range TN_1 at time instant 1

$$TN_{1,0,2} \leq TN_{1,0,1}$$

The spill load brings back the live range into the register. There is no necessity of a spill load for any live range TN_i corresponding to register r if the live range is already in the register r. Further, a temporary name is live in a register r at time t either if it was live at time step $t \ominus 1$ or if a spill load is scheduled in time step t. For a spill load at time instant t, the following constraints need to be satisfied.

$$TN_{i,r,t} \leq TN_{i,r,t \ominus 1} + LTN_{i,r,t} \quad \forall r \in R, \forall t \in loadset(i) \tag{7}$$

In Figure 2, the spill loads at time steps [3, 7] in register 0 must satisfy the following constraints.

$$TN_{1,0,3} \leq TN_{1,0,2} + LTN_{1,0,3}; \quad TN_{1,0,4} \leq TN_{1,0,3} + LTN_{1,0,4}$$

$$TN_{1,0,5} \leq TN_{1,0,4} + LTN_{1,0,5}; \quad TN_{1,0,6} \leq TN_{1,0,5} + LTN_{1,0,6}$$

$$TN_{1,0,7} \leq TN_{1,0,6} + LTN_{1,0,7}$$

If a spill load is not possible at time instant t, i.e t $\notin$ $loadset(i)$ and a spill store is not possible at time instant $t \ominus 1$, i.e t$\ominus$1 $\notin$ $storeset(i)$, then the following continuation constraints must be satisfied.

$$TN_{i,r,t} \leq TN_{i,r,t\ominus 1} \quad \forall r \in R, for\ all\ t \notin loadset(i) \wedge t \ominus 1 \notin storeset(i) \quad (8)$$

In Figure 2, the continuation constraints corresponding to time instants 1, 8 and 9 for register 0 and live range TN_i are

$$TN_{1,0,1} \leq TN_{1,0,0}; \quad TN_{1,0,8} \leq TN_{1,0,7}; \quad TN_{1,0,9} \leq TN_{1,0,8}$$

Interference Constraints: It is important to ensure that the same register is not allocated to multiple live ranges. Interference constraints ensure that at any instant of time, a register holds a single live range. It is sufficient to ensure that after each live range definition, the register holds a single live range. At time instant t which is the definition time of live range TN_i, the following constraints must be satisfied for each register r

$$\sum_j TN_{j,r,t} \leq 1 \quad (9)$$

where $TN_{j,r,t} = 0$ for $t \notin [T_j^{def}, T_j^{end}]$.

Functional Unit Constraints: The spill loads and store generated require memory functional units. Thus a spill load or a store can be scheduled at a particular instant t provided there is a free memory unit available. Hence for scheduling spill loads or stores, the following memory unit constraints need to be satisfied for each time slot t' $\in$ [0, II-1].

$$\sum_{i,r} LTN_{i,r,t} + \sum_{j,r} STN_{j,r,t} \leq M \quad for\ all\ t \in [0, II-1] \quad (10)$$

TN_i is the live range with $t \in$ loadset(i) and TN_j is the live range with $t \in$ storeset(j). M is the number of memory units available for spill loads and stores after the memory requirements of instructions that are scheduled at time instant t in the kernel are satisfied. The above constraint ensures that sum of all spill loads and stores scheduled at any time instant t in the kernel is lesser than or equal to the number of free memory units available.

3.3 Objective Function

The objective function is to minimize the number of spill loads and stores.

$$Minimize : \sum_{i,r,t}(STN_{i,r,t} + LTN_{i,r,t}) \tag{11}$$

4 Simplified Formulation

The previous formulation can be simplified by omitting the r indices from the spill load and store decision variables. In this formulation, we decide whether a spill load or a store is necessary at a given time step without considering which register the store or load should use. The constraints are suitably modified to reflect the same. The register used by the spill store and loads can be easily inferred from the $TN_{i,r,t}$ variables as a post-processing step. The simplified formulation is given below:

$$Minimize \sum_{i,t}(STN_{i,t} + LTN_{i,t})$$

$$\sum_{r \in R} TN_{i,r,t} = 1 \qquad \forall i \in I \text{ and } t = T_i^{def} \tag{12}$$

$$\sum_{r} TN_{i,r,t} - \sum_{t'} LTN_{i,t'} \geq 1 \qquad \forall t = T_j^{def} \text{ and } j \in use(TN_i) \; t' \in (t \ominus lat_{load}, t] \tag{13}$$

$$LTN_{i,t} - \sum_{t"} STN_{i,t"} \leq 0 \qquad \forall t \in loadset(i) \; \forall i \; t" \in [T_i^{def} + lat_i, t \ominus lat_{store}] \tag{14}$$

$$STN_{i,t} - \sum_{r} TN_{i,r,t} \leq 0 \qquad \forall t \in storeset(i) \; \forall i \tag{15}$$

$$TN_{i,r,t} - TN_{i,r,t \ominus 1} - LTN_{i,t} \leq 0 \qquad \forall t \in loadset(i) \; \forall i \tag{16}$$

$$\sum_{r} TN_{i,r,t} - \sum_{r} TN_{i,r,t \ominus 1} - LTN_{i,t} \leq 0 \qquad \forall t \in loadset(i) \; \forall i \tag{17}$$

$$\sum_{j} TN_{j,r,t} \leq 1 \qquad \forall t \in [0, II - 1] \; \forall r \tag{18}$$

$$\sum_{i} LTN_{i,t} + \sum_{j} STN_{j,t} \leq M \qquad \forall t \in [0, II - 1] \tag{19}$$

$$TN_{i,r,t \oplus 1} - TN_{i,r,t} \leq 0 \qquad \forall t \oplus 1 \notin loadset(i) \; \forall i \; \forall r \tag{20}$$

Equation 17 ensures that each spill load loads the live range in at-most one register.

5 Experimental Evaluation

5.1 Experimental Methodology

We have used the SUIF [12] as the compiler front end for the benchmarks. For the compiler back end, we have used Trimaran [13] compilation and simulation environment for VLIW architectures. The data dependence graphs are generated using the Trimaran's back end . The initial modulo schedule is obtained using an integer linear program formulation [10]. The machine architecture used in the formulation is a load-store architecture with 3 memory units, 3 integer units and 4 floating point units. For the constructed schedule, modulo variable expansion [14] is performed to ensure that no live range is longer than II. We then generate the formulation proposed in this paper to perform register allocation and necessary spill code generation and scheduling. We have considered architectures with 16 and 32 registers. The integer linear programming formulation is solved using the CPLEX 9.0 solver [5] running on a Pentium 4, operating at 3.06 GHz with 4 GB RAM. A CPU-time limit of 600 seconds is used for solving our integer linear program. The loops in which the integer linear program timed out are not considered for evaluation.

5.2 Results

We compare our approach with the best performing heuristic [21], viz spilling uses, with a quantity factor of 0.5 and a traffic factor of 0.3. The quantity factor is used for deciding the number of spill candidates and traffic factor is used for the selection of spill candidates. We refer to the above heuristic as SU and our formulation as ILP.

Spill Code. The amount of spill code introduced impacts the code quality of the schedule. We evaluated the amount of spill code generated by ILP and SU. In this result, we do not consider amount of spill code generated with the loops requiring an increase in II with SU as it is not fair to compare schedules with

Table 1. Spill code and prevention of II increase with 32 registers

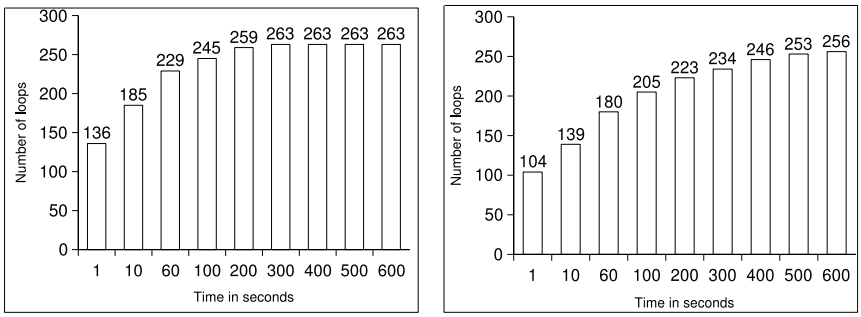
Benchmark	#loops	#loops with reg pressure	Total spill code		% decrease in spill code(ILP)	#loops without II increase(ILP)	% loops without II increase(ILP)
			ILP	SU			
168.wupwise	25	12	96	123	21.95	1	8.33
179.art	40	15	46	57	19.3	1	6.67
183.equake	42	9	44	53	16.98	1	11.11
188.ammp	46	14	56	63	11.11	2	14.29
200.sixtrack	46	9	70	84	16.67	1	11.11
Perfect Club	69	31	191	237	19.41	4	12.9
Total	268	90	503	617	18.48	10	11.11

Table 2. Spill code and prevention of II increase with 16 registers

Benchmark	#loops	#loops with reg pressure	Total spill code		% decrease in spill code(ILP)	#loops without II increase(ILP)	% loops without II increase(ILP)
			ILP	SU			
168.wupwise	25	19	128	152	15.79	0	0
179.art	40	26	85	106	19.81	1	3.85
183.equake	42	19	88	104	15.38	4	21.05
188.ammp	46	21	88	95	7.37	2	9.52
200.sixtrack	46	23	112	131	14.50	3	13.04
Perfect Club	69	49	313	346	9.54	9	18.37
Total	268	157	814	934	12.85	19	12.10

different initiation intervals. Table 1 and Table 2 report the amount of spill generated for an architecture with 32 and 16 registers respectively. Though number of loops with higher register pressure (greater than the available registers) is small, we find that there is fairly large spill code being generated. The amount of spill code reduction with ILP when compared to SU ranges from 11.11% to 21.95% for 32 registers and it ranges from 7.37% to 19.81% for 16 registers. On an average ILP produces 18.48% less spill code on an average for an architecture with 32 registers and 12.85% less spill code on an average for an architecture with 16 registers.

Initiation Interval. The throughput of a software pipelined loop is measured in terms of the initiation interval. Table 1 and Table 2 report the number of loops requiring an increase in the initiation interval in SU and do not require an increase in II while using ILP. ILP eliminates the need for an increase in II when compared to SU in 6.67% to 14.29% of the loops in various benchmarks. On an average, ILP eliminates an increase in II in 11% of the loops for an architecture with 32 registers and 12% of the loops for 16 registers.



(a) 16 registers (b) 32 registers

Fig. 3. Solution time taken by ILP

In summary, we observe that our ILP approach is able to reduce the amount of spill code by 18.48% and eliminate an increase in II by 11.11% on average among 90 loops on an architecture with 32 registers.

Solution Time. In Figure 3(a) and Figure 3(b), we report the time taken by the ILP, where the X-axis represents the time taken and Y-axis, the number of loops for which the solution can be found with the given time. For example, for the case of 16 registers, 136 out of 268 loops take less than one second each. The arithmetic mean of the time taken by ILP for each loop is 18.44 seconds in the case of 16 registers and is 77.79 seconds in the case of 32 registers.

6 Related Work

Software pipelining has been extensively studied and few of the contributions in this area are in [6,7,14,17,19]. A comprehensive survey is available in [2]. A considerable amount of work has been done to minimize the register requirements of the the software pipeline schedule. Among these, Huff [11] uses slack scheduling and tries to minimize the combined register pressure. In [8], ILP formulation for generating the schedule has been proposed and minimization of the number of buffers required in such a scenario is addressed in [10]. A number of modulo scheduling heuristics that reduce the register pressure and generate schedules with smallest number of registers have been proposed in [15]. All these do not consider the dual problem of scheduling with a given number of registers.

Register allocation for software pipelined loops was proposed by Rau et al. [18]. They consider an architecture that incorporates rotating registers. However spill code generation and scheduling was not considered. Ning et al. [16] have proposed an algorithmic framework for concurrent scheduling and register allocation. Their approach estimates the register requirement with the help of buffers. Zalamea et al. [21] have described methods for generating spill code when the register pressure is greater than the number of registers. But they did not consider register allocation and introduction of spill code was based on heuristics.

Goodwin et al. [9] have proposed a 0-1 integer linear programming formulation for global register allocation. Our model inherits certain ideas from their approach. They do not consider register allocation for software pipelined loops and hence does not deal with the problem of spill code scheduling in a cyclic schedule. Methods for generating spill code on-the-fly using heuristics have been proposed in [1]. Since the generation of spill code is based on heuristics, solution may not always be optimal.

Integer linear programming formulations for instruction scheduling have been proposed by Chang [3] and Wilken [20]. In [3], the authors consider instruction scheduling and spill code generation. However, they do not perform register allocation and their technique does not guarantee optimal spill code. They also do not address the problem of scheduling the generated spill code in a compact

cyclic schedule. Our work, for the first time proposes an integrated formulation for register allocation, optimal spill code generation and scheduling in software pipelined schedules.

7 Conclusions

The paper presents an optimal method for integrated register allocation and spill code scheduling in software pipelined loops, using a 0-1 integer linear programming formulation. We formulate it as an integer linear program because the selection of a spill candidate based on a certain heuristic can generate extraneous spill code, which in turn may necessitate an increase in the initiation interval. The formulation serves as a framework with which various heuristics can be evaluated. Experiments show that our formulation outperforms the best performing heuristic proposed in [21]

- By eliminating an increase in the initiation interval in 11.11% of the 90 loops that had sufficient register pressure for an architecture with 32 registers and in 12% of the cases with 157 loops on a machine with 16 registers.
- By generating on an average, 18.48% less spill code for an architecture with 32 registers and 12.85 % less spill code for an architecture with 16 registers.

Acknowledgments

The authors are thankful to the members of the High Performance Computing Laboratory for their useful comments and discussions. The authors are also thankful to the anonymous reviewer for suggesting the simplified formulation. The first author acknowledges the partial support provided by the Philips research fellowship.

References

1. Alex Aleta, Josep M. Codina, Antonio Gonzalez, and David Kaeli. Demystifying on-the-fly spill code. *SIGPLAN Not.*, 40(6):180–189, 2005.
2. Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Comput. Surv.*, 27(3):367–432, 1995.
3. C.M Chen C.M Chang and C.T King. Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. *Computers and Mathematics with Applications*, 34(9):1–14, 1997.
4. Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 174–187, London, UK, 1998. Springer-Verlag.
5. ILOG CPLEX:. http://www.ilog.com.
6. James C. Dehnert and Ross A. Towle. Compiling for the cydra 5. *J. Supercomput.*, 7(1-2):181–227, 1993.
7. Kemal Ebcioglu and Alexandru Nicolau. A global resource-constrained parallelization technique. In *ICS '89: Proceedings of the 3rd international conference on Supercomputing*, pages 154–163, New York, NY, USA, 1989. ACM Press.

8. Paul Feautrier. Fine-grain scheduling under resource constraints. In *LCPC '94: Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, pages 1–15, London, UK, 1995. Springer-Verlag.
9. David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocations using 0-1 integer programming. *Softw. Pract. Exper.*, 26(8):929–965, 1996.
10. R. Govindarajan, Erik R. Altman, and Guang R. Gao. A framework for resource-constrained rate-optimal software pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 07(11):1133–1149, 1996.
11. Richard A. Huff. Lifetime-sensitive modulo scheduling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–267, 1993.
12. SUIF Compiler Infrastructure. http://suif.stanford.edu/suif/.
13. Trimaran: An infrastructure for research in instruction level parallelism. http://www.trimaran.org.
14. M. Lam. Software pipelining: an effective scheduling technique for vliw machines. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 318–328, New York, NY, USA, 1988. ACM Press.
15. Josep Llosa, Mateo Valero, and Eduard Ayguade. Heuristics for register-constrained software pipelining. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 250–261, Washington, DC, USA, 1996. IEEE Computer Society.
16. Qi Ning and Guang R. Gao. A novel framework of register allocation for software pipelining. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 29–42, Charleston, South Carolina, 1993.
17. B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *MICRO 14: Proceedings of the 14th annual workshop on Microprogramming*, pages 183–198, Piscataway, NJ, USA, 1981. IEEE Press.
18. B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. *SIGPLAN Not.*, 27(7):283–299, 1992.
19. B. Ramakrishna Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74, New York, NY, USA, 1994. ACM Press.
20. Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 121–133, New York, NY, USA, 2000. ACM Press.
21. Javier Zalamea, Josep Llosa, Eduard Ayguade, and Mateo Valero. Improved spill code generation for software pipelined loops. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 134–144, New York, NY, USA, 2000. ACM Press.