Register Pressure in Software-Pipelined Loop Nests: Fast Computation and Impact on Architecture Design

Alban Douillet and Guang R. Gao

Department of Electrical and Computer Engineering
University of Delaware, Newark, DE 19716-3130
{*douillet,ggao*}*@capsl.udel.edu*

Abstract. Recently the Single-dimension Software Pipelining (SSP) technique was proposed to software pipeline loop nests at an arbitrary loop level [18–20]. However, SSP schedules require a high number of rotating registers, and may become infeasible if register needs exceed the number of available registers. It is therefore desirable to design a method to compute the register pressure quickly (without actually performing the register allocation) as an early measure of the feasibility of an SSP schedule. Such a method can also be instrumental to provide a valuable feedback to processor architects in their register files design decision, as far as the needs of loop nests are concerned.
This paper presents a method that computes quickly the minimum number of rotating registers required by an SSP schedule. The results have demonstrated that the method is always accurate and is 3 to 4 orders of magnitude faster on average than the register allocator. Also, experiments suggest that 64 floating-point rotating registers are in general enough to accommodate the needs of the loop nests used in scientific computations.

1 Introduction

Software pipelining [1, 4, 9, 10, 13] is an efficient and important method to schedule loops by overlapping the execution of successive iterations. The most popular technique, modulo-scheduling (MS) [3, 8, 10, 12, 16, 21], only addresses single loops or the innermost loop of a loop nest. Traditional approaches to schedule loop nests mainly focus on scheduling the innermost loop and extending the schedule toward the outer levels by hierarchical reduction [10, 14]. An alternative way is to perform MS after loop transformations [2]. A new resource-constrained scheduling technique named Single-dimensional Software-Pipelining (SSP) [18–20] does not restrain itself to the innermost loop and can software pipeline any given loop in a loop nest. If the innermost level is chosen, SSP is proven to be equivalent to MS. Experimental results have shown that SSP often outperforms MS, and is fully compatible with the wide array of loop optimizations and transformations used for MS. The technique can currently be applied to any source imperfect loop nests with no conditional statements or function calls and with run-time constant trip counts.

In the SSP compilation process, shown in Figure 1, registers are allocated after the one-dimensional (1-D) schedule is computed. However, both phases are time-consuming (the register allocation problem is NP-complete [18], even for single loops

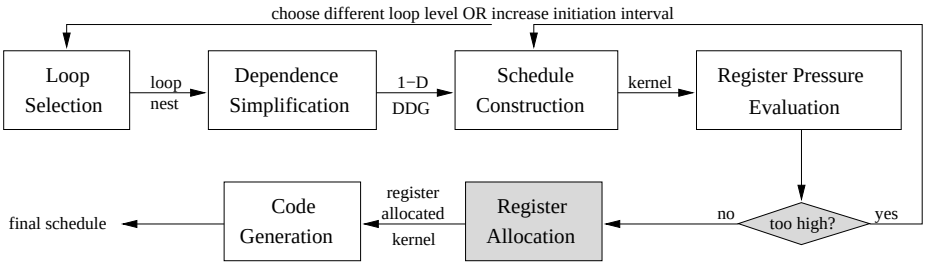


Fig. 1. SSP Compilation Flow

[16]). Therefore, it is preferable to detect early if the register allocator is bound to fail because of a too high register pressure. The scheduler can then compute a different, but more favorable schedule. We propose in this paper a fast evaluation method to measure the rotating register pressure, named $MaxLive$, of any kernel computed by the SSP scheduler. It is defined as the maximum number of lifetimes at any time during the execution of the loop nest scheduled with SSP. It is a theoretical lower bound that may not be achievable. Only loop variants, allocated to rotating registers, are considered. Loop invariants are assumed to be allocated to static registers. When unspecified, 'register' will always refer to 'rotating register'. Any register spilling technique is assumed to have been applied earlier to the 1-D schedule and is not the subject of the paper.

Such an evaluation method is important and has many uses. (1) First, it allows the compiler to avoid running the expensive register allocator when it is bound to fail. A new 1-D schedule with lower requirements can then be computed by increasing the initiation interval or choosing another loop level, for instance. (2) Second, because the register pressure is a direct function of the 1-D schedule , the method can be used to compare the register pressure of 1-D schedules computed by different SSP scheduling methods. (3) Third, the computed register pressure can also be used to measure the effectiveness of any register allocator. (4) Last, the method provides a valuable feedback to processor architects in their register files design decision, as far as the needs of loop nests are concerned. Other questions can then be answered. Is the register pressure the same for both floating-point (FP) and integer (INT) registers? Are the register files of the target architectures balanced enough to efficiently handle the register pressure? Can we anticipate the final register pressure or the number of registers allocated by a specific register allocator?

Several issues specific to SSP must be handled. First, the final schedule is composed of more than one repeating pattern. Second, some lifetimes are stretched to honor resource constraints. Last, the initiation rate of the lifetimes is irregular. In this paper, we propose a method to compute the rotating register pressure of any given 1-D schedule. The method is fast: it approximates $MaxLive$ by skipping the initialization and conclusion phases of the final schedule and considers a unique outermost loop iteration, or outermost iteration for short. A second method, comprehensive, accurate, but very slow, is used as reference. For clarity and space reasons, the second method is not presented in the paper, but is accessible in [5] instead. We will refer to them as the fast method and the comprehensive method, respectively.

It is the first time a method to compute the register pressure of an SSP schedule is proposed. With single loops, where MS is used, the traditional technique is to count the number of lifetimes in the kernel, also named $MaxLive$ [17]. Our method can be seen as its natural extension to handle the more complex issues specific to the multidimensional case, presented in section 3.2. $MaxLive$ was the chosen method to evaluate the efficiency of register allocators in [6, 11]. Other work [15] considered the theoretical register pressure during the scheduling phase by counting the number of buffers required for each functional units. However the number of buffers did not take into account that some buffers could be reused. The register pressure was also studied for non software-pipelined schedules, such as the concept of $FatCover$ in [7]. Llosa et al. [11] used $MaxLive$ to measure the register pressure of floating-point benchmarks. Their results also show that a FP register file of 64 registers would accommodate most of the register pressure and limit accesses to memory in the case of MS scheduled loops. The results were later confirmed in [22].

The methods presented in this paper were implemented in the Open64/ORC 2.1 compiler on an Itanium workstation. The experiments were conducted on a set of 125 loop nests of various depths. The experiments lead to several conclusions. (1) The fast method is at least 3 orders of magnitude faster than the register allocator and could therefore be used in a compiler framework to quickly determine the feasibility of an SSP schedule. (2) Most of the loop nests of depth 3 or less require less than 96 INT registers and about half of the loop nests of depth 4 or higher cannot be scheduled because of a too high INT register pressure. (3) The FP register pressure never exceeds 47 registers and therefore more than half of the FP register file is never used, showing an imbalance in the usage of the register files between INT and FP. (4) If half of the FP register file is used for INT values instead, then 76% of the loop nests of depth 5 could be software-pipelined with SSP.

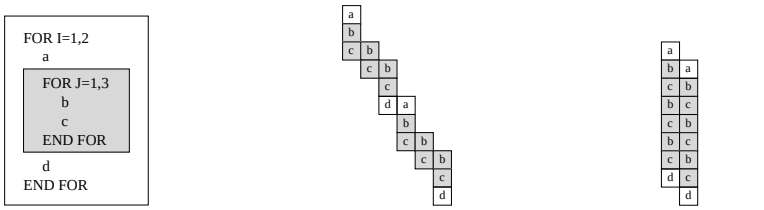
The paper is organized as follows. Section 2 briefly introduces the SSP method. Section 3 defines some notations and conventions used in the paper, formulates the problem and explains in details the issues to tackle. Our solution is then described in Section 4. Experiments and results are presented in Section 5 before concluding in Section 6.

2 Single-dimension Software Pipelining

2.1 Overview

Single-dimension Software Pipelining (SSP) [18–20] is a resource-constrained scheduling method to software pipeline perfect and imperfect loop nests with constant trip counts at run-time. Unlike traditional innermost-loop-centric approaches [10, 14, 16], SSP does not necessarily software pipeline the innermost loop of a loop nest, but directly software pipelines the loop level estimated to be the most profitable. The enclosing loops of the selected loop, if any, are untouched. If the innermost loop level is chosen, SSP is equivalent to MS applied to single loops. SSP retains the simplicity of MS, and yet may achieve significantly higher performance [19].

Figure 2(a) shows an example of a double loop nest. In Figure 2(b), the innermost loop is modulo scheduled, whereas, in Figure 2(c), the outermost loop is software



(a) A double loop nest (b) MS schedule (c) SSP schedule

Fig. 2. Simple SSP software pipelining example

pipelined using SSP. Note that, although the two outermost iterations are running in parallel, the innermost loop is running sequentially within each outermost iteration. In our example the SSP schedule is shorter by 2 cycles.

SSP proceeds in several steps to produce the final schedule [18–20]. First, the most profitable loop level is chosen for scheduling based on instruction-level parallelism or other criterion. Second, multi-dimensional dependences are simplified into a 1-dimensional problem from which a 1-D schedule is computed, represented by a kernel. Registers are then allocated to the loop variants in the kernel. Last, the 1-D schedule is mapped back to the multi-dimensional iteration space and the final schedule is generated as an assembly code.

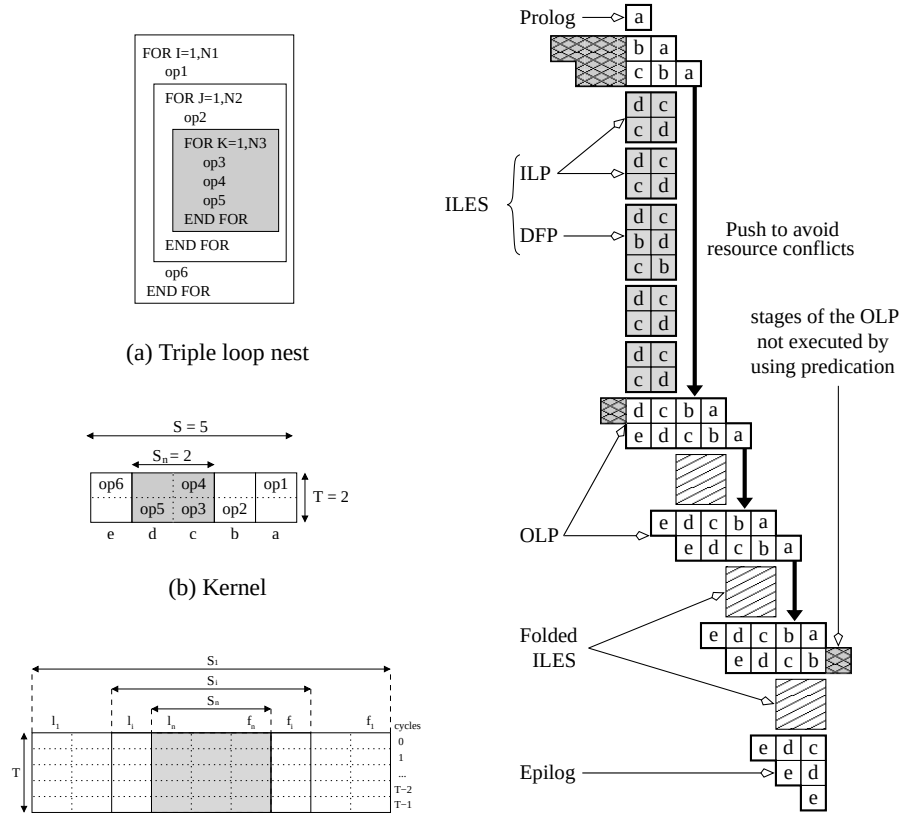
Because the enclosing loops to the selected loop are untouched, they are ignored from our point of view and we will always see the chosen loop as the outermost loop of the loop nest. The loops are then referred as L_1, L_2, . . ., L_n from the outermost level to the innermost level where n is the depth of the loop nest.

2.2 From the Kernel to the Final Schedule

The final schedule is exclusively made of multiple copies of the kernel, with sometimes variations or truncations. As such, one only needs to consider the kernel when counting the lifetimes in the final schedule. A kernel is composed of S stages. Each stage takes T cycles to execute. Zero or more operations are scheduled in each modulo-cycle of each stage with the restriction that operations from different levels must be scheduled into different stages.

Figure 3(b) shows the kernel of the triple loop nest from Figure 3(a). There are 5 stages a, b, c, d , and e. The outermost loop is made of all the $S = S_1 = 5$ stages, the middle loop of $S_2 = 3$ stages (b, c, d), and the innermost loop of $S_n = 2$ stages (c, d). Each stage is made of $T = 2$ modulo-cycles and some stages have empty schedule slots.

A more generic kernel is shown in Figure 3(c). The indexes of the first and last stage of loop level i are noted f_i and l_i respectively. The number of stages at level i is noted $S_i = l_i - f_i + 1$. The total number of stages is noted S and is equal to S_1. All the stages have the same initiation interval T. In Figure 3(b), $f_1 = 0$, $f_2 = 1$, $f_3 = 2$, $l_3 = 3$, $l_2 = 3$, $l_1 = 4$, and $T = 2$.



(a) Triple loop nest

(b) Kernel

(c) Generic Kernel

(d) Final Schedule (N_1=8, N_2=2, N_3=3)

Fig. 3. A More Complex Example

Figure 3(d) shows the final schedule of our example. The stages are symbolized by their letters for clarity purposes. We assume that the trip counts for each loop are $N_1 = 8$, $N_2 = 2$, and $N_3 = 3$ (stage b appears only twice in each column, and stages c and d appear three times after each instance of stage b). A column represents the execution of a single outermost iteration (8 total). Both inner loops are represented only for the first two outermost iterations. Afterwards, they are symbolized by a dashed box. Because of resource constraints, only a group of $S_n = 2$ outermost iterations can fully be executed in parallel [20, 19]. The other outermost iterations are delayed and pushed later in the schedule, as illustrated by the thick vertical arrow.

Because of the delays and the repetitive nature of the schedule, the final schedule can be decomposed into five different patterns: the prolog, the outermost loop pattern (OLP), the innermost loop pattern (ILP), the draining and filling pattern[1] (DFP), and

[1] also called *transition code* in [20]

the epilog. The ILP and DFP form the Inner Loop Execution Segment (ILES).Each pattern can be fully derived from the kernel. The ILP and DFP are obtained by cyclicly considering S_n consecutive stages among the S_i stages of the kernel for loop level i [19]. Predication is used in the OLP to truncate unnecessary stages.

3 Problem Formulation & Lifetimes Classification

3.1 Lifetimes Notations & Conventions

The distance in terms of outermost iterations between the definition and the use of a loop variant is called the $omega$ value of the use. The maximum $omega$ value of all the uses of a loop variant represents the number of live-in values required for the variant. Similarly, if live-out values are required from a loop variant, we note $alpha$ the number of values. Those notations are consistent with Rau's conventions [17]. A loop variant is statically defined only once per loop level.

The time period when an instance of a loop variant v is live is called the *scalar lifetime*, or *lifetime* for simplicity, of that instance. In our examples, as shown in Figure 4(a), a circle represents the start of a lifetime, a cross the end, and a dash a non-killing use of the variable. At any given cycle c of the final schedule, the number of lifetimes is called the $FatCover$ at cycle c. $MaxLive$ is the maximum of all the $FatCovers$.

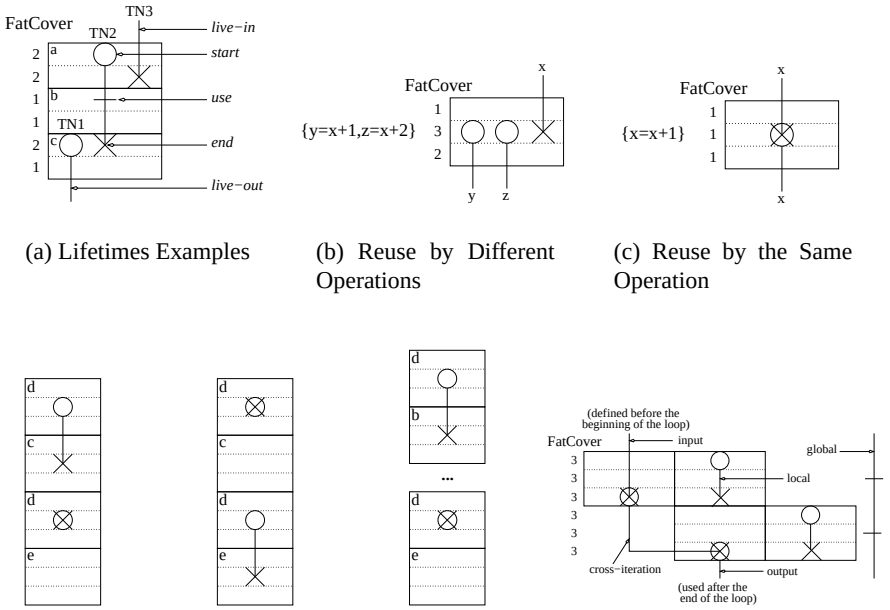
In order for the operations to be interruptible and restartable in a VLIW machine and to avoid dependencies between operations scheduled in the same cycle, a lifetime is started at the beginning of the cycle of the defining operation and is killed at the end of the cycle of the killing operation. This convention matches Rau's convention about scalar lifetimes in [17]. A register cannot be used and defined in the same cycle, except if it is by the same operation, as shown in Figure 4(b) and 4(c). We assume that the intermediate representation follows the same conventions. A loop variant can be redefined by the same operation like in Figure 4(c). In the latter case, the operation will be considered only as a use of the variant for the purpose of our algorithms.

3.2 Problem Formulation & Issues

The problem can be formulated as follows: *given a loop nest and a SSP schedule for it, evaluate the rotating register pressure $MaxLive$ of the final schedule.*

The problem presents several issues. First, the lifetimes do not exhibit regular patterns like with modulo scheduling. Successive instances of the same lifetime do not reappear every T cycles: because of the push operations, some delays are encountered. For the same reason, some lifetimes appear to be *stretched* until the stalled outermost iterations they belong to resume their execution. Examples can be seen in Figure 5.

Second, the number of lifetimes in the same stage and modulo-cycle may vary, depending on the position of the stage in the final schedule. For instance, Figure 4(d) shows a part of the final schedule presented in Figure 3(d). The loop variant is defined in the first instance of stage d and used in stage c. The same loop variant is defined again in the second instance of d but never used. However, the register required for the



(a) Lifetimes Examples (b) Reuse by Different Operations (c) Reuse by the Same Operation

(d) Variant 1 (e) Variant 2 (f) Variant 3 (g) Lifetimes Classification

Fig. 4. Lifetimes Notations, Situations, and Classification

definition must be accounted for during the only cycle where the second instance of the loop variant is live. Symmetrically, a value may be defined each iteration and never used until the last iteration, where the value is used in the enclosing loop (Figure 4(e)).

Similarly, whether the stage belongs to the last instance of the enclosing loop also influences the number of local lifetimes. In Figure 4(f), the last instance of the loop variant is used at the beginning of the enclosing loop. If it is the last iteration of the enclosing loop, then the value is never used and the local lifetime is reduced to a single cycle. We refer to those two situations as $first$ and $last$.

Finally, the method must be fast in order to be used as a tool by the register allocator and the scheduler to help detect infeasible solutions early.

3.3 Lifetimes Classification

For the purpose of the algorithms described in this paper, lifetimes are classified into 5 categories, illustrated in Figure 4(g). Global lifetimes covers the whole execution of the loop nest. This is typical of loop invariants and those lifetimes are not considered by our algorithm. Output lifetimes hold values computed within the loop nest that will be used outside. The number of parallel live-out values of the same loop variant is equal to the $alpha$ value of the variant. Input lifetimes start before the beginning of the loop and terminates before the end. The number of parallel live-in values of the

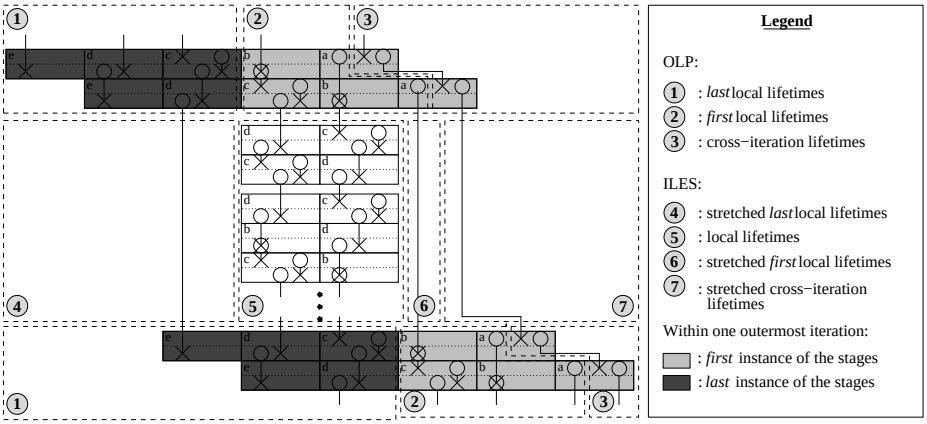


Fig. 5. Register Pressure Computation Overview

same loop variant is the maximum of all the $omega$ values of the variant among all its uses. Cross-iteration lifetimes cross outermost iterations. By construction, a sequence of cross-iteration lifetimes start with input lifetimes. Every other lifetime is said to be local to the current outermost iteration.

4 Register Pressure Computation

This section presents the details of our solution. We make the assumption that the maximum register pressure will appear in the steady phase (OLP and ILES) of the final schedule. Therefore, input and output lifetimes are ignored and only local and cross-iteration lifetimes are considered. Experiments in Section 5.1 will show that this assumption is always correct.

A snapshot of our final schedule during the steady phase is shown in Figure 5. The lifetimes can be partitioned into 7 groups, shown in the legend. To compute the maximum register pressure of the final schedule, we count the number of lifetimes in each of the seven groups. Cross-iteration lifetimes are counted by analyzing the definition and uses of each cross-iteration loop variant. Local lifetimes are counted for each single stage of the kernel for both situations: first or last in the current outermost iteration. The exact algorithms are available in [5]. An overview is given in the next subsections.

4.1 Cross-Iteration Lifetimes

Because the outermost loop level is the only level actually software pipelined, only variants defined in the outermost level can have a cross-iteration lifetime. The first step consists of identifying the cross-iteration variants. They are defined in the stages appearing in the outermost loop only and show at least one use with an omega value greater than 0. Then, for each variant, the stage and modulo-cycle of the definition and of the last use are computed and noted S_{def}, c_{def}, S_{kill}, and c_{kill}, respectively. The definition of each

```
COMPUTE_CROSS_ITERATION_LT():
  civs ← ∅ // cross-iteration variants set
  ovs ← set of the variants defined in the outermost loop

  // Identify the cross-iteration variants
  for each operation op in the schedule
    for each source operand src of op
      if omega(op, src) > 0 and src ∈ ovs then
        civs ← civs ∪ {src}
        initialize S_def, c_def, S_kill, c_kill for src to −1

  // Collect the parameters for each cross-iteration variant
  for each stage s from l_1 to f_1, backwards
    for each cycle c from T − 1 to 0, backwards
      for each operation op in s at cycle c
        for each source operand src of op in civs
          if S_kill(src) = s + omega(op, src) then
            S_kill(src) unchanged
            c_kill(src) ← max(c_kill(src), c)
          else if S_kill(src) < s + omega(op, src) then
            S_kill(src) ← s + omega(op, src)
            c_kill(src) ← c
        for each result operand res of op in civs
          c_def(res) ← c
          S_def(res) ← s
```

```
COMPUTE_LOCAL_LT():
  // Start recursive analysis from the outermost level
  ∀(s, c, p) ∈ [f_1, l_1]X[0, T]X{first, last}
    LT_local(s, c, p) ← −1,    Visit_Level(1, ∅)

  // Initialize first with last value if first uninitialized
  for each stage s from f_1 to l_1
    for each cycle c from 0 to T
      if LT_local(s, c, first) = −1 then
        LT_local(s, c, first) ← LT_local(s, c, last)

VISIT_LEVEL(level level, live set live):
  // Count the local lifetimes for loop level 'level'
  for each stage s from l_level to f_level, backwards
    for each cycle c from T to 0, backwards
      live ← live ∪ DEF(s, c) ∪ USE(s, c)
      if LT_local(s, c, last) = −1 then
        LT_local(s, c, last) ← |live|
      else
        old ← LT_local(s, c, first)
        LT_local(s, c, first) ← max(old, |live|)
      live ← (live − DEF(s, c)) ∪ USE(s, c)
    // Recursive call for the inner levels
    if level < n and s = f_level+1 then
      Visit_Level(level + 1, live)
```

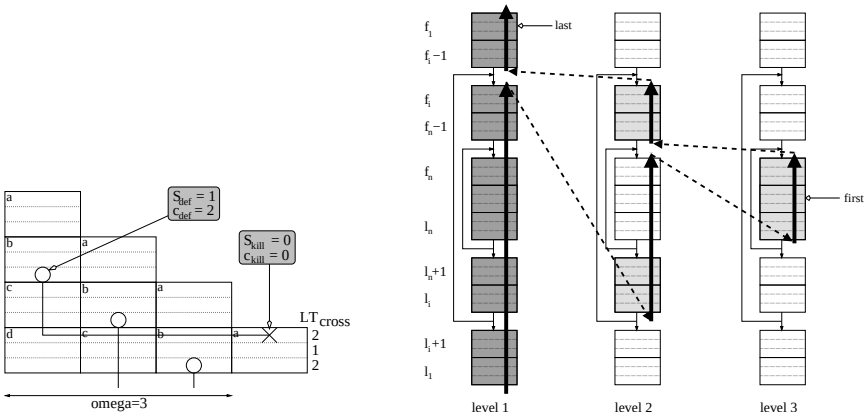
Fig. 6. Fast Method Algorithms

variant is unique and therefore easily found. Because cross-iteration lifetimes span several outermost iterations, the last use of a such lifetimes must be searched among each of the spanned iterations. The stage index of the last use is computed by adding the omega value of the use to its stage index.

Afterward, the number of cross-iteration variants lifetimes at modulo-cycle c in the OLP is then given by $LT_{cross}(c)$, shown in Figure 8. $S_{kill}(v) - S_{def}(v) + 1$ represents the length in stages of the lifetime of v. The two other δ terms are adjustment factors to take into account the exact modulo-cycle the variant is defined or killed in the stage. Figure 7(a) shows an example of a cross-iteration lifetime. The lifetime starts at $S_{def} = 1$, corresponding to stage b, and $c_{def} = 2$, and stops $omega = 3$ iterations later in stage $S_{kill} = 0 + omega$ at modulo-cycle $c_{kill} = 0$. Then the number of cross-iteration lifetimes for that variant is equal to 2, 1, and 2 at modulo-cycle 0, 1, and 2 respectively.

4.2 Local Lifetimes

The computation of the local lifetimes is done by using traditional backwards data-flow liveness analysis on the control-flow graph (CFG) of the loop nest where each loop level is executed only once. A generic example for a loop nest of depth 3 is shown in Figure 7(b). The final schedule is partitioned into $2 * n - 1$ blocks of stages. For each level but the innermost, there are two blocks. The first is made of the stages exclusively belonging to the loop level and executed before the ILP, and the second of the stages exclusively belonging to the same level but executed after. The innermost level has only one block made of the S_n innermost stages. The separations correspond to the separations between stages of different levels in the kernel and the order in which the



(a) Cross-iteration Lifetimes Example (b) Local Lifetime Computation Order

Fig. 7. Lifetimes Computation

stages are visited is the order of the stages in the kernel. The figure shows the stage indexes for each block. Stages visited as $first$ are represented in light gray whereas stages visited as $last$ are in dark gray.

4.3 Register Pressure

The OLP is composed of S_n kernels, each made of all the S stages. The register pressure is the sum of the cross-iteration and local lifetimes for each stage. The distinction between first and last instance of the local lifetimes must be made, leading to $S - n$ different cases. We then obtain the formula for LT_{olp} shown in Figure 8. The first term counts all the cross-iteration lifetimes. The second is the maximum number of local lifetimes among the S_n possible instances of kernel in the OLP.

The formula for the ILP and DFP is LR_{iles}. The first three terms correspond to the three types of stretched lifetimes shown in Figure 5: 7, 4, and 6 in that order. Their number is fixed for the entire execution of the ILES and equal to the number of lifetimes live at the exit of the OLP. The fourth term of the formula corresponds to the local lifetimes of the ILES (5). $MaxLive$ is then the maximum between between the maximum register pressure of the OLP and the maximum register pressure of the ILES patterns.

Although it is possible to modify the algorithms and formulas to make the $MaxLive$ computation incremental, it is not believed that our method is fast enough to help guide the instruction scheduler.

5 Experiments

The algorithms were implemented in the ORC 2.1 compiler and tested on an 1.4GHz Itanium2 machine with 1GB RAM running Linux. The benchmarks are SSP-amenable

$$
\begin{aligned}
LT_{cross}(c) &= \sum_{v \in civs} ((S_{kill}(v) - S_{def}(v) + 1) + \delta_{def}(c,v) + \delta_{kill}(c,v)) \\
&\quad \text{where } \begin{cases} \delta_{def}(c,v) = -1 \text{ if } c < c_{def}(v), 0 \text{ otherwise} \\ \delta_{kill}(c,v) = -1 \text{ if } c > c_{kill}(v), 0 \text{ otherwise} \end{cases} \\
LT_{iles}(c) &= LT_{cross}(T) + \sum_{s=l_n}^{l_1} LT_{local}(s,T,last) + \sum_{s=f_1}^{f_n-2} LR_{local}(s,T,first) \\
&\quad + \max_{l \in [2,n]} \left(\max_{i_0 \in [0,S_l-1]} \left(\sum_{i=0}^{S_n-1} LT_{local}(f_l + (i_0+i)\%_0 S_l, c, first) \right) \right) \\
LT_{olp}(c) &= LT_{cross}(c) + \max_{i \in [1,S_n]} \left(\sum_{s=l_n-i}^{l_1} LT_{local}(s,c,last) + \sum_{s=f_1}^{l_n-1-i} LT_{local}(s,c,first) \right) \\
FatCover_{olp} &= \max_{\forall c \in [0,T-1]} (LT_{olp}(c)) \\
FatCover_{iles} &= \max_{\forall c \in [0,T-1]} (LT_{iles}(c)) \\
MaxLive &= max(FatCover_{iles}, FatCover_{olp})
\end{aligned}
$$

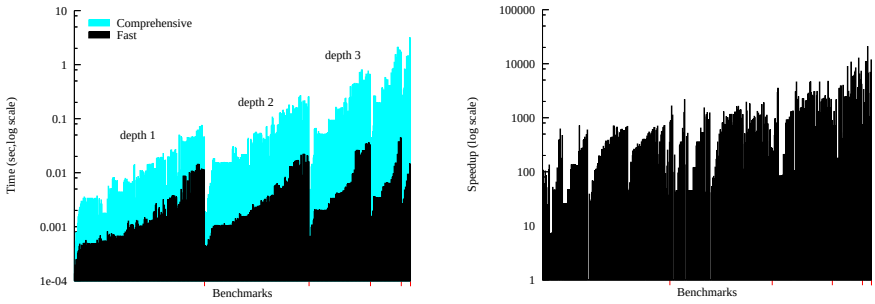
Fig. 8. Register Pressure Computation Formulas

loop nests extracted from the Livermore Loops, the NPB 2.2 benchmarks and the SPEC2000 FP benchmark suite. A total of 127 loop nests were considered. When all the different depths are tested, 328 different test cases were available. There were 127, 102, 60, 30, and 9 loop nests of depth 1, 2, 3, 4, and 5, respectively.

The main results are summarized here and explained in details in the next subsections. (1) The fast method is 1 to 2 orders of magnitude faster than the comprehensive method, and 3 to 4 orders of magnitude faster than the register allocator. (2) Despite the approximations made by the fast method, its computed $MaxLive$ is identical to $MaxLive$ computed by the comprehensive method. No rule of thumb could be deduced to predict $MaxLive$ by only considering the 1-D schedule parameters such as kernel length, number of loop variants, and others. Rotating Register pressure increases quickly for integer values as the loop nest gets deeper and about half of the loop nests of depth 4 or 5 show a $MaxLive$ higher than the size of the INT register file. (3) The floating-point rotating register pressure remains about constant as the depth of the loop nests increases, and never exceeds 47 registers. Consequently, the floating-point rotating register file could be reduced from 96 to 64 registers. The extra 32 registers could be added to the integer register file instead.

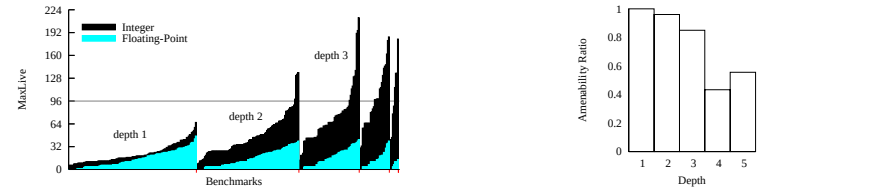
5.1 Compilation Time

The time measurements are presented in Figure 9(a) where the loop nests have been sorted first by increasing depth, delimited by tics on the horizontal axis, then by increasing kernel length. Note the logarithmic scale for the vertical axis. The comprehensive and fast methods take up to 3.18 and 0.04 seconds respectively, with an average of 0.16 and 0.005 seconds. The running time of each method is directly related to the kernel length. The shape of the graph confirms the quadratic running time of the fast method and the influence of the depth of the loop nest. The fast method is 22.9 times faster than



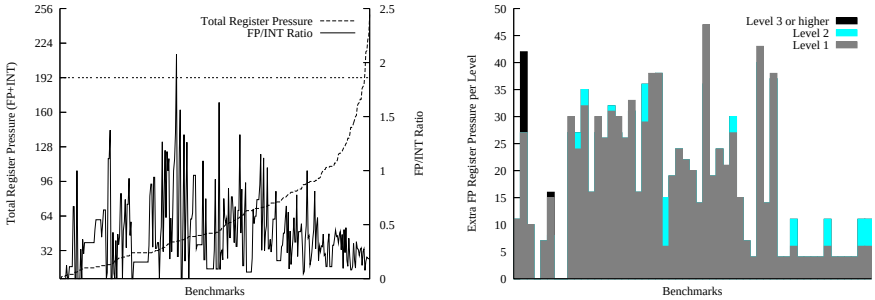
(a) Running Time

(d) Fast Method vs. the Reg. Allocator



(b) MaxLive

(e) Ratio of Loop Nests Amenable to SSP



(c) Total Pressure & FP/INT Ratio

(f) FP MaxLive Progression

Fig. 9. Experimental Results

the comprehensive method, with a maximum of 217.8. As the loop nest gets deeper, the speedup becomes exponentially more significant.

The running time of the fast method and the register allocator from [18] are compared in Figure 9(d). On average, the fast method is 3 orders of magnitude faster than the register allocator with a maximum of 20000. As the loop nest gets deeper, i.e. as the $MaxLive$ increases and the need for a quick method to evaluate the register pressure a

priori becomes stronger, the speedup increases, making the fast method a valid tool to detect infeasible schedules before the register allocator.

Although the fast method does not take into account live-in and live-out lifetimes, the computed $MaxLive$ was identical for the two other methods in all the benchmarks tested. $MaxLive$ is indeed less likely to appear in the prolog and epilog.

5.2 MaxLive

The computed $MaxLive$ is a optimistic lower bound on the actual register pressure. It does not take into account that a value held in one register at cycle c must remain in the same register at cycle $c + 1$ or that the use of rotating registers reserves a group of consecutive registers at each cycle, even if some of them are not currently used. The actual register allocation solution computed by an optimal register allocator may allocate more registers than $MaxLive$. However, with the addition of register copy instructions, $MaxLive$ registers can always be reached

The computed $MaxLive$ is shown in Figure 9(b) for INT and FP loop variants. The benchmarks have been sorted by increasing depth, indicated by small tics on the horizontal axis, and by increasing $MaxLive$. The average $MaxLive$ for INT and FP are 47.2 and 15.0 respectively with a maximum of 213 and 47. If we only consider rotating registers, the 96 hard limit on the number of available FP registers in the Itanium architecture is never reached. However the 96 limit for INT registers is reached more often as the depth of the loop nests increases, up to 56% for the loop nests software pipelined at level 4 as shown in Figure 9(e).

INT $MaxLive$ increases faster than FP $MaxLive$. INT $MaxLive$ indeed increases as the nest gets deeper because more inner iterations are running in parallel. It is particularly true for INT values that are used as array indexes. If an array index is defined in the outermost loop, then there is one instance of the index for each concurrent outermost iteration in the final schedule. For FP values however, this is not the case. They are typically defined in the innermost loop only and have very short lifetimes.

We also tried to approximate $MaxLive$ by looking at the 1-D schedule parameters. However no rule of thumb could be derived by looking at one parameter such as S, S_n, the length of the kernel or the number of loop variants. The $MaxLive$ was also compared to the actual number of registers allocated by the register allocator. Unlike in MS where the number of registers allocated rarely exceeds $MaxLive$+1 [17], the difference with SSP varies between 0% and 77%. Such results are explained by the higher complexity of SSP schedules compared to MS and because $MaxLive$ is not a tight lower bound.

5.3 Floating-Point Register File Size

Figure 9(c) shows the total register pressure, defined as the sum of $MaxLive$ for INT and FP registers, and the ratio between $MaxLive$ for FP and INT registers. The benchmarks are sorted by increasing ratio. The total register pressure rarely exceeds 192 registers, the size of the rotating register file in the Itanium architecture. Although FP $MaxLive$ can be twice higher than INT $MaxLive$, the FP/INT ratio remains lower than 0.5 when the total register pressure is greater than 96.

Figure 9(f) shows FP $MaxLive$ as the same loop nest is scheduled at deeper levels. FP $MaxLive$ does not or barely increases as a same loop nest is scheduled at a deeper level. The maximum FP $MaxLive$ never exceeds 47 registers.

Several conclusions, that may be useful for future designs of architectures with the same number of functional units and superscalar degree than the Itanium architecture, can be drawn from these remarks. First, the INT register file may benefit from a smaller FP register file with a ratio of 2 for 1. The FP register size can either be decreased to save important chip real estate, or the INT register file increased to allow more SSP loops to be register allocated. Second, for the set of benchmarks used in our experiments, the optimal size for the FP register file would be 64. It would not prevent any other loop nests from being register allocated while giving extra registers to the INT register file. If a size of 64 and a INT/FP ratio of 2 are chosen, the feasibility ratio for loop nests of depth 4 and 5 would jump from 43% and 56% to 77% and 67%, respectively. The FP/INT ratio chosen for the Itanium architecture is not incorrect, but was chosen with MS loops in mind, which exhibits a lower INT $MaxLive$.

6 Conclusion

Single-dimension Software Pipelining (SSP) software pipelines a loop nest at an arbitrary level. However the register pressure is too high for half of the loop nests of depth 4 or more. It is therefore necessary to know the register pressure early in the compilation process to avoid calling the register allocator when it is bound to fail. The results of the evaluation could also be used to evaluate the efficiency of any SSP register allocator. We proposed in this paper a methodology that quickly computes the rotating register pressure of an SSP schedule

Results showed that our method is accurate and at least 3 orders of magnitude faster than the register allocator on average, making it a valid tool to detect infeasible schedules early. From a hardware co-design point of view, experimental results suggest that SSP schedules would benefit from a smaller floating-point rotating register file of 64 registers and a twice as large integer rotating register file.

7 Acknowledgments

We would like to acknowledge Dr. Hongbo Rong for his enthusiastic moral and technical support during the course of this work, and Jean-Christophe Beyler and the anonymous reviewers for their insightful comments. This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No.NBCH30904, by NSF grants No.0103723 and No.0429781, and by DOE grant No.DE-FC02-OIER25503.

References

1. A. Aiken, A. Nicolau, and S. Novack. Resource-constrained software pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 6(12):1248–1270, Dec. 1995.
2. S. Carr, C. Ding, and P. Sweany. Improving software pipelining with unroll-and-jam. In *Proc. 29th Annual Hawaii Int'l Conf. on System Sciences*, pages 183–192, 1996.
3. A. Dani, V. Ramanan, and R. Govindarajan. Register-sensitive software pipelining. In *Proc. of 12th Int'l Par. Processing Symp./9th Int'l Symp. on Par. and Dist. Systems*, 1998.
4. A. Darte, R. Schreiber, B. R. Rau, and F. Vivien. Constructing and exploiting linear schedules with prescribed parallelism. *ACM Trans. on Design Automation of Electronic Systems*, 2001.
5. A. Douillet and G. R. Gao. Register pressure in software-pipelined loop nests: Fast computation and impact on architecture design. CAPSL TM 58, Univ. of Delaware, Newark, Delaware, 2005. In ftp://ftp.capsl.udel.edu/pub/doc/memos.
6. A. Eichenberger, E. Davidson, and S. Abraham. Minimum register requirements for a modulo schedule. In *Proc. of the 27th int'l symp. on Microarchitecture*, pages 75–84, 1994.
7. L. J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. In *Proc. of the 4th Int'l Conf. on Compiler Construction*, pages 176–191. Springer-Verlag, 1992.
8. R. Huff. Lifetime-sensitive modulo scheduling. In *Proc. of the conf. on Programming language design and implementation*, pages 258–267. ACM Press, 1993.
9. S. Jain. Circular scheduling: A new technique to perform software pipelining. In *Proc. of the Conf, on Programming Language Design and Implementation*, pages 219–228, 1991.
10. M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. of the conf. on Programming language design and implementation*, 1988.
11. J. Llosa, E. Ayguadé;, and M. Valero. Quantitative evaluation of register pressure on software pipelined loops. *International Journal of Parallel Programming*, 26(2):121–142, 1998.
12. J. Llosa, A. González, E. Ayguadé, and M. Valero. Swing modulo scheduling: A lifetime sensitive approach. In *Proc. Conf. on Par. Arch. and Compil. Tech.*, pages 80–86, 1996.
13. S.-M. Moon and K. Ebcioğlu. Parallelizing nonnumerical code with selective scheduling and software pipelining. *ACM Trans. on Prog. Lang. and Systems*, 19(6):853–898, 1997.
14. K. Muthukumar and G. Doshi. Software pipelining of nested loops. In *Proc. of the Int'l Conf. on Compiler Construction*, volume 2027, pages 165–181. LNCS, 2001.
15. Q. Ning and G. R. Gao. A novel framework of register allocation for software pipelining. In *Proc. of the symp. on Principles of programming languages*, pages 29–42, 1993.
16. B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proc. of the int'l symp. on Microarchitecture*, pages 63–74, 1994.
17. B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *Proc. of the conf. on Prog. lang. design and impl.*, pages 283–299, 1992.
18. H. Rong, A. Douillet, and G. R. Gao. Register allocation for software pipelined multi-dimensional loops. In *Proc. of the conf. on Prog. lang. design and impl.*, 2005.
19. H. Rong, A. Douillet, R. Govindarajan, and G. R. Gao. Code generation for single-dimension software pipelining of multi-dimensional loops. In *Proc. of Int. Symp. on Code Generation and Optimization*, page 175, 2004.
20. H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao. Single-dimension software pipelining for multi-dimensional loops. In *Proc. of Int. Symp. on Code Generation and Optimization*, pages 163–174, 2004.
21. J. Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown: optimal vs. heuristic methods in a production compiler. In *Proc. of the conf. on Prog. lang. design and impl.*, pages 1–11, 1996.
22. J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Two-level hierarchical register file organization for vliw processors. In *Proc. of the symp. on Microarch.*, pages 137–146, 2000.