University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

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

Alban Douillet
Hongbo Rong
Guang R. Gao

CAPSL Technical Memo 58
May 3, 2005

Department of Electrical and Computer Engineering
University of Delaware
{douillet,ggao,rong}@capsl.udel.edu

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

Recently the Single-dimension Software Pipelining (SSP) technique was proposed to software pipeline loop nests at an arbitrary loop level [23, 24, 25]. However, SSP schedules require a high number of rotating registers, and often become infeasible if the number of available rotating registers in the target architecture is exceeded. 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 two feedback methodologies to computes the minimum number of rotating registers required by an SSP schedule. The first is comprehensive but slow, whereas the second makes some approximations but is very fast. We have implemented both methods in the Open64/ORC compiler framework. The results have demonstrated that the fast method is always accurate and is impressively fast, 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.

Contents

List of Figures

1 Introduction

Software pipelining [1, 4, 8, 9, 13] is an efficient and important method to schedule loops by overlapping the execution of successive iterations. The most popular technique, modulo-scheduling [3, 7, 9, 11, 21, 26], 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 [9, 15]. An alternative way is also to perform modulo scheduling after loop transformations [2]. A new resource-constrained scheduling technique named Single-dimensional Software-Pipelining (SSP) [23, 24, 25] 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 modulo-scheduling. Experimental results have shown that SSP often outperforms modulo-scheduling, and is fully compatible with the wide array of loop optimizations and transformations used for modulo-scheduling. 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.

To software pipeline a loop nest, SSP selects the most profitable loop in a loop nest via heuristics. Then the multi-dimensional data dependence graph (DDG) is simplified into a one-dimensional DDG that is used to produce the one-dimensional schedule, represented by the kernel. Registers are then allocated before the generation of the assembly code. However, the register allocation phase is NP-complete [23], even for single loops [21], and good allocation solutions cannot be quickly computed. Moreover, because several iterations of the selected loops (and its inner loops) are running in parallel, the number of live variables at any point in time is high. If the register pressure is higher than the number of available processor registers, then the SSP method fails and either a new schedule must be computed, or another loop selected.

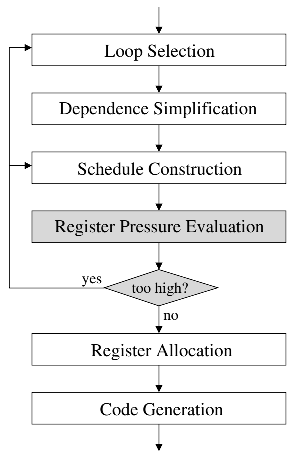


Figure 1: SSP Compilation Flow

The topic of the paper is to propose a fast evaluation method to measure the register pressure of any kernel computed by the SSP scheduler. We assume that any register spilling technique has already been applied to the schedule and is not the subject of the paper. Such method is important and has many uses.

(1) First, it allows the compiler to avoid running the expensive register allocator when it is guaranteed that no feasible register allocation solution can be found. The SSP framework then has the choice between computing another schedule with a lower register pressure (but lower time performance) or to choose another loop level to schedule, as shown in Figure 1. (2) Second, because the register pressure is a direct function of the schedule and not of the register allocation algorithm, the method can be used to evaluate the impact of any scheduling methods for SSP on the register pressure. (3) Third, the computed register pressure can also be used to measure the effectiveness of any register allocator. A register allocation algorithm would be considered effective if the number of allocated registers is close to the actual register pressure. (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. Several questions can then be answered. How high exactly is the register pressure? Is the register pressure the same for both floating-point and integer 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, that did not exist for single loops and are specific to SSP schedules, and presented in details in Section 3.2, must be handled. First, the 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. A comprehensive method and a fast method that solve all the issues are proposed to compute the rotating register pressure, named $MaxLive$, of any SSP schedule for a loop nest of depth 1 or higher. $MaxLive$ 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. The fast method is significantly faster than the comprehensive method: it skips the initialization and conclusion phases of the schedule and considers a unique outermost iteration. Moreover cross-iteration lifetimes are summarized with a single compact mathematical formula. The comprehensive method is always accurate, very slow, and used as reference.

No method was ever introduced to compute the register pressure of an SSP schedule. In the case of single loops, where modulo scheduling is used, the traditional technique is to count the number of lifetimes in the kernel, also named $MaxLive$ [22]. The fast 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 allocator for modulo scheduled loops [5, 10]. Other work [18] 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 [6]. Llosa et al. [10] used $MaxLive$ to measure the register pressure of floating-point benchmarks. Their results also show that a floating-point register file of 64 registers would accommodate most of the register pressure and limit accesses to memory in the case of modulo scheduled loops. The results were later confirmed in [28] .

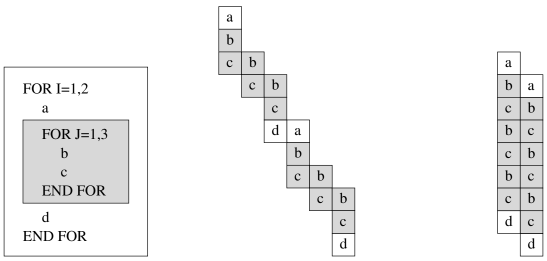
The methods were implemented in the Open64/ORC 2.1 compiler [19, 20] on an Itanium workstation. The experiments were conducted on a set of 125 loop nests of various depths leading to 325 different test cases. The experiments lead to several important conclusions. (1) The fast method proved to be at least 3 orders of magnitude faster than the register allocator and could therefore be used in a compiler framework to quickly determine the feasibility of a SSP schedule. The register pressure computed with the fast method is the same as the comprehensive method. (2) Most of the loop nests of depth 3 or less require less than 96 integer registers and that about half of the loop nests of depth 4 or higher cannot be scheduled because of a too high integer register pressure with 213 being the maximum. (3) The floating-point register pressure never exceeds 47 registers and therefore more than half of the floating-point register file is never used, showing an imbalance in the usage of the register files between integer and floating-point. (4) Further analyses showed that if half of the floating-point register file is used for integer 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 with an example that will be reused in the following sections. Section 3 defines some notations and conventions used in the paper about lifetimes in an SSP schedule, defines the problem formulation and explains in details the issues to be solved. The fast method is then described in Section 4, and the comprehensive method in Section 5. Experiments and results are presented in Section 6 before concluding in Section 7.

2 Single-dimension Software Pipelining

2.1 Overview

Single-dimension Software Pipelining (SSP) [23, 24, 25] 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 [9, 16, 21], 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 classical modulo scheduling applied to single loops. SSP retains the simplicity of modulo scheduling, and yet may achieve significantly higher performance [24].



(a) A double loop nest (b) MS schedule (11 cycles) (c) SSP schedule (9 cycles)

Figure 2: Simple SSP software pipelining example

Figure 2(a) shows an example of a double loop nest to intuitively understand how SSP software pipelines a loop nest. In Figure 2(b), the innermost loop is modulo scheduled, whereas, in Figure 2(c), the outermost loop is software pipelined using SSP. Note that, although the two outermost iterations are running in parallel, the innermost loop is running sequentially within each outermost iteration. If the number of outermost and innermost iterations are respectively 2 and 3 as shown in the figures, then the SSP schedule is shorter by 2 cycles.

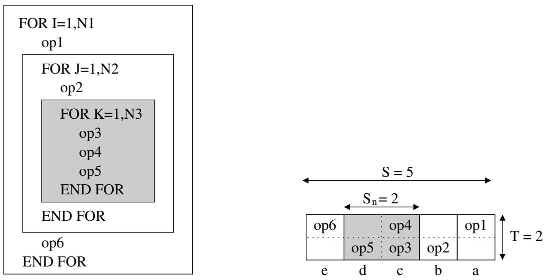
SSP proceeds in several steps to produce the final schedule [23, 24, 25]. First, a set of heuristics looking at some optimization objectives, such as instruction-level parallelism or cache data reuse, determines which level is deemed the most profitable and therefore which level should be software-pipelined. Second, the multi-dimensional scheduling problem for the selected loop and its inner loops is simplified into a 1-dimensional problem. A 1-dimensional schedule is computed, represented by a kernel, as in modulo scheduling. Registers are then allocated to the loop variants in the kernel. Last, the 1-dimensional schedule is mapped back to the multi-dimensional iteration space to form the final schedule semantically equivalent to the original chosen loop and the assembly code is generated.

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

In this paper, we assume that a loop has been selected and that a kernel has already been computed. Details can be found in [25]. To understand how to measure the register pressure in the final schedule, we now explain what a kernel is and how it is used to build the final schedule.

Intuitively, the kernel is a template for 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 cycle of each stage with the restriction that operations from different levels must be scheduled in different stages.



(a) A triple loop nest (b) Its Kernel

Figure 3: A More Complex Example

Figure 3(b) shows the kernel of a triple loop nest. 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). When representing a kernel, we use a vertical dotted line and a vertical plain line to separate stages from the same loop level and from different loop level, respectively. Each stage is made of $T = 2$ cycles and some stages have empty schedule slots.

A more generic kernel is shown in Figure 4. 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, $f_1 = 0$, $f_2 = 1$, $f_3 = 2$, $l_3 = 3$, $l_2 = 3$ and $l_1 = 4$. In the paper, the cycle of an operation refers to the cycle index within a stage, also called modulo-cycle.

Figure 5 shows the final schedule corresponding to the example in Figure 3. A column represents the execution of a single outermost iteration. The stages are symbolized by their letters for clarity purposes. To simplify further, we assume that $N_1 = 8$, $N_2 = 2$, 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) and both inner loops are represented only for the first two outermost iterations. Afterwards, they are symbolized by a dashed box. However, the outermost loop has 8 iterations, represented by the 8 columns. Because of resource constraints, only a group of $S_n = 2$ outermost iterations can fully be executed in parallel [25, 24]. To avoid resource conflicts, the other outermost iterations are delayed and pushed later in the schedule, as illustrated by the thick vertical arrow.

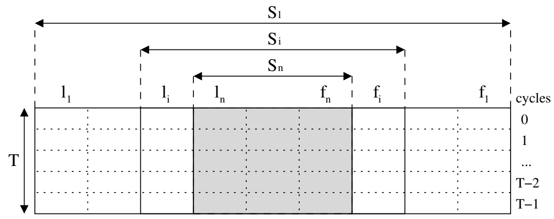


Figure 4: Generic Kernel

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 (DFP), and the epilog[1]. The OLP corresponds to the execution of operations from every level whereas the ILP corresponds to the execution of operations from the innermost loop only. The DFP corresponds to every other level in-between. The DFP and ILP form the inner patterns. The DFP is used to flush the current loop level and start a new one. For instance, in our schedule, the DFP flushes the innermost loop level and starts a new iteration of the middle loop level by issuing stage *b*. In double loop nests, the DFP does not exist. In single loops, the schedule is only made of the prolog, epilog and OLP.

The five different patterns are sufficient to form the final schedule. More iterations of any loop level simply means more occurrences of the corresponding pattern and its inner patterns. Moreover, each pattern can be fully derived from the kernel. The prolog, epilog and OLP are truncated versions of the kernel, while the inner patterns are obtained by cyclicly considering S_n consecutive stages among the S_i

[1]The DFP corresponds to the *transition code* and *[Fill L_i Pipeline]* instruction mentioned in earlier work [24].

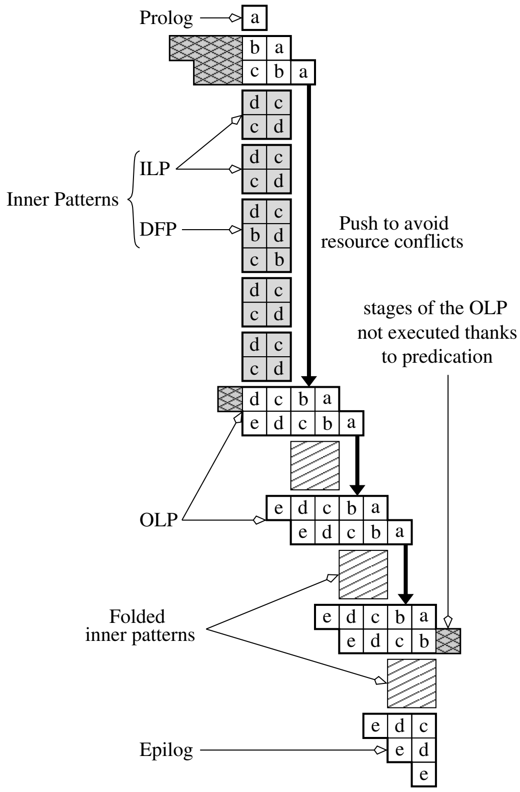


Figure 5: More Complex Schedule ($N_1 = 8$, $N_2 = 2$, $N_3 = 3$)

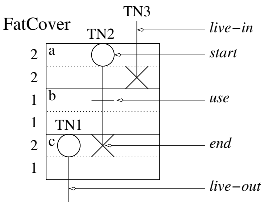
stages of the pattern for loop level i. Predication is used in the OLP truncate unnecessary stages. Therefore, it is sufficient to only consider the kernel when counting the lifetimes in the final SSP schedule of any loop nest.

3 SSP Lifetimes

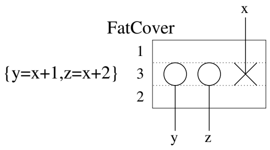
3.1 Lifetimes Notations & Conventions

Following the conventions of the compiler used for implementation [19], loop variants are noted TNx where TN stands for *temporary name* and x is a positive integer. If the value of a TN v is used i outermost iterations later by operation u, we say that $omega(u, v) = i$. $omega$ represents the number of live-in values required for that TN. Similarly, if live-out values are required from a TN, we note $alpha$ the number of values. Those notations are consistent with Rau's conventions [22]. A loop variant is defined only once per loop level, with the exception of redefinitions where the same operation uses and redefines the same loop variant.

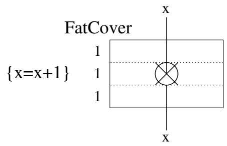
The time period when an instance of a loop variant v is live is called the *scalar lifetime*, or *lifetime* for simplicity, of that instance. It starts at the beginning of cycle of the operation defining v and ends at the end of the cycle of the last operation using v. In our examples, as shown in Figure 6(a), a solid box represents a stage and a dashed horizontal line the beginning of a new cycle within a stage. A circle represents the start of a lifetime, a cross the end, and a dash a non-killing use of the variable. Vertical lines represent the lifetimes themselves. At any given cycle c, the number of lifetimes is called the $FatCover$ of the schedule at cycle c. By extension, $FatCover_{ilp}$ is the maximum $FatCover$ of all the cycles of the ILP. A similar notation is used for the prolog, epilog, the DFP and the OLP. $MaxLive$ is the maximum of all the $FatCovers$. In order for the operations to be interruptible and restartable in a



(a) Lifetimes Examples



(b) Reuse by Different Operations



(c) Reuse by the Same Operation

Figure 6: Lifetimes Notations

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 [22]. A register cannot be reused in the same cycle, except if it is by the same operation, as shown in Figure 6(b) and 6(c). We assume that the intermediate representation follows the same conventions. A variable can be redefined by the same operation like in Figure 6(c). In the latter case, the operation will be considered only as a use of the variable for the purpose of our algorithms.

While allocating rotating registers, several issues arise. For instance, when the innermost loop iterates, only the rotating registers associated with operations from the innermost loop should rotate. This issue and others have been solved in [23, 24] and are not the topic of this paper. The later phases of the SSP framework will ensure that the right rotating registers will be used and rotated when needed. Therefore, when measuring the rotating register pressure of an SSP schedule, we do not have to consider how rotating registers are handled.

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 of the schedule*. The rotating register pressure is defined as the maximum number of lifetimes of rotating loop variants at any cycle c of the schedule. The register pressure at a given cycle is called the $FatCover$ of that cycle. We assume that the schedule is represented by its kernel and register spills, if any, are already present in the schedule. The source loop nest can be imperfect, but must not contain any conditional statements or function calls. The loop trip counts are assumed to known at run-time.

The problem presents several issues. First, the lifetimes do not exhibit regular pattern like with modulo scheduling applied to single loops. 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 4. Those lifetimes must be considered when measuring the register pressure in the inner patterns.

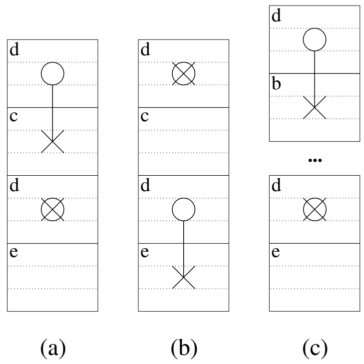


Figure 7: Local Lifetimes Variances

Second, the number of local lifetimes in the same stage and cycle may vary, depending on the position of the stage in the schedule. For instance, Figure 7(a) shows a sample of the schedule presented in Figure 5. The loop variant is defined in the first instance and used in the second. The same loop variant is defined again in the second instance but never used. However, the register required for the 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 7(b)). Therefore the two local lifetimes of the same loop variant have different lengths, and the register pressure within the same stage may vary depending on if it is the last instance or not.

Similarly, whether the stage belongs to the last instance of the enclosing loop also influences the number of local lifetimes. In Figure 7(c), 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. The fast method refers to those two situations as $first$ and $last$.

Moreover, although the OLP has more stages (S) than the inner patterns (S_n), the maximum register pressure is not necessarily located in the OLP. Therefore the inner patterns cannot be omitted. However, the prolog and epilog, truncated versions of the OLP, can be omitted if they do not prevent the algorithms from running properly. They are actually omitted by the fast method.

Finally, the method must be accurate enough so that the results can be used to measure the register pressure of a schedule or evaluate the efficiency of register allocation algorithms. On top of everything, 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 8. Global lifetimes exist before the beginning of the loop nest and either terminates after the end or in the last outermost iteration of the loop nest. This is typical of a constant variable being read by the operations of the loop nest, but never written, also called loop invariant. Output lifetimes hold values computed within the loop nest that will be used outside. The number of parallel live-out values of the same variable is the $alpha$ value of the variable. Input lifetimes start before the beginning of the loop and terminates before the end. The number of parallel live-in values of the same variable is the maximum of all the $omega$ values of the variable 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.

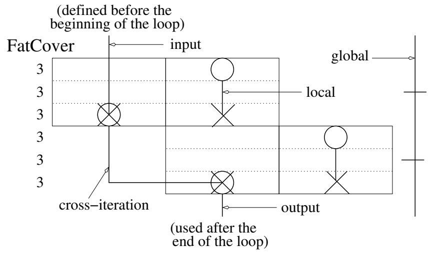


Figure 8: Lifetimes Classification Example

Because loop invariants are usually allocated to non-rotating registers, global lifetimes are not considered by our algorithms.

4 Fast Register Pressure Computation

This section presents the details of the algorithms to quickly compute the register pressure of a loop nest already scheduled with SSP. In our solution, we make the assumption that the maximum register pressure will appear in the steady phase of the schedule. Therefore, input and output lifetimes are ignored and only local and cross-iteration lifetimes are considered. The impact of such an approximation is measured in Section 6.1.

A snapshot of our schedule during the steady phase is shown in Figure 4. This time, the lifetimes have been made visible. Using the classification presented in Section 3.3, the schedule can be partitioned into 7 groups. In the OLP, there are the local lifetimes and the cross-iteration lifetimes (3). The local lifetimes themselves can be separated into 2 groups depending on if they appear in the first (2) or last (1) instance of a stage in the current outermost iteration. The difference is important, as explained in Section 3.2. In the inner patterns, the lifetimes are either local to the stages being executed (5) or stretched from the stalled outermost iterations. The stretched lifetimes are sorted into 3 categories: stretched local *after* the current outermost iterations (4), stretched local *before* the current outermost iterations (6), and the stretched cross-iteration lifetimes (7).

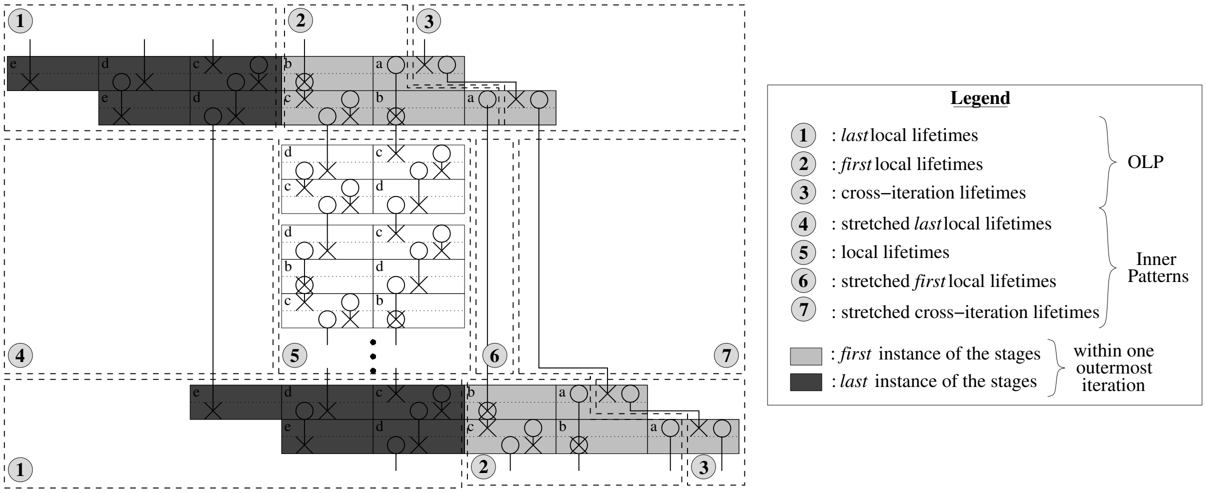


Figure 9: Register Pressure Computation Overview

To compute the maximum register pressure of the schedule, we count the number of lifetimes in each of the seven groups. Then the sum of lifetimes is computed for both the OLP and the inner patterns. The $MaxLive$ is the maximum value between the two. The algorithm handles cross-iteration and local lifetimes differently. 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 following subsections explain in details each phase of the algorithm.

4.1 Cross-Iteration Lifetimes

We identify cross-iteration lifetimes first. By elimination, the remaining lifetimes are local. Because the outermost loop level is the only level actually software pipelined, only variables defined in the outermost level can have a cross-iteration lifetime.

The algorithm to analyze the cross-iteration lifetimes, *Compute_Cross_Iteration_LT()*, is shown in Figure 10. The first step consists of identifying the cross-iteration variables. Cross-iteration variables 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 variable, the stage and cycle of the definition and of the last use are stored for later use and noted S_{def}, c_{def}, S_{kill}, and c_{kill} respectively. The definition of each variable is unique and easily found (using the assumptions made in Section 3.1). 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.

Once the cross-iteration variables are known and their four parameters computed, the number of

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

  // Identify the cross-iteration variables
  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 variable
  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
```

Figure 10: Cross-Iteration Lifetimes Computation Algorithm

cross-iteration variables lifetimes at cycle c in the OLP is then given by $LT_{cross}(c)$:

$$LT_{cross}(c) = \sum_{v \in civs} ((S_{kill}(v) - S_{def}(v) + 1) + \delta_{def}(c, v) + \delta_{kill}(c, v)) \tag{1}$$

$$\text{where } \begin{cases} \delta_{def}(c, v) = -1 \text{ if } c < c_{def}(v), 0 \text{ otherwise} \\ \delta_{kill}(c, v) = -1 \text{ if } c > c_{kill}(v), 0 \text{ otherwise} \end{cases}$$

$S_{kill}(v) - S_{def}(v) + 1$ represents the length in stages of the lifetime of v. The two other δ terms are adjustment factors to take into account the exact cycle the variable is defined or killed in the stage.

Figure 11 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 cycle $c_{kill} = 0$. Then the number of cross-iteration lifetimes for that variable is equal to 2, 1, and 2 at 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 [14] on the control-flow graph (CFG) of the loop nest where each level is executed only once. A generic example for a loop nest of depth 3 is shown in Figure 13. The 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

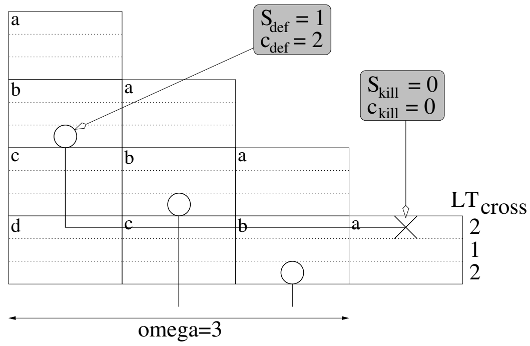


Figure 11: Cross-iteration Lifetimes Example.

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 stages are visited is the order of the stages in the kernel. The figure shows the stage indexes for each block.

<u>COMPUTE_LOCAL_LT</u>():
  // Start recursive analysis from the outermost level
  $LT_{local}(s,c,p) \leftarrow -1, \forall(s,c,p) \in [f_1, l_1]X[0,T]X\{first, last\}$
  Visit_Level(1, $\emptyset$)

  // Initialize first *with* last *value if* first *still 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**
        $LT_{local}(s,c,first) \leftarrow LT_{local}(s,c,last)$

<u>VISIT_LEVEL</u>(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
      $live \leftarrow live \cup DEF(s,c) \cup USE(s,c)$
      if $LT_{local}(s,c,last) = -1$ **then**
        $LT_{local}(s,c,last) \leftarrow |live|$
      else
        $LT_{local}(s,c,first) \leftarrow max(LT_{local}(s,c,first), |live|)$
      $live \leftarrow (live - DEF(s,c)) \cup USE(s,c)$
    // Recursive call for the inner levels
    if $level < n$ **and** $s = f_{level+1}$ **then**
      Visit_Level($level + 1$, $live$)

Figure 12: Local Lifetimes Computation Algorithms

Compute_Local_LT, the algorithm to count the local lifetimes for each cycle of the schedule, is shown in Figure 12. It serves as a wrapper routine and calls the recursive routine Visit_Level for the outermost loop, starting with an empty current live set. The stages are then visited in the order shown in Figure 13. The solid arrows represents the computation whereas the dashed arrows represents a recursive call to Visit_Level(). Within each stage, the cycles are analyzed backwards, from cycle $T - 1$ to cycle 0.

Because the number of local lifetimes in the same stage may vary, as explained in Section 3.2, two values, $first$ and $last$, are computed for each stage and cycle. They correspond to the first and last stages in Figure 4. Stages visited as $first$ are represented in light gray whereas stages visited as $last$ are in dark gray. Because the liveness analysis is a backwards data-flow algorithm, the $last$ value of each stage and cycle is computed during the first call to Visit_Level(). The $first$ value is then gradually computed in the subsequent calls. The last visit of a stage determines the final $first$ value for each cycle of that stage. The $first$ and $last$ values of the stages exclusively appearing the in the outermost loop are equal and are therefore visited only once. The values are copied at the end of Compute_Local_LT().

Visit_Level() is called once per loop level by the enclosing loop. At that moment, the live set parameter is equal to the live-out set of the previous encounter of the same loop level by the algorithm. Therefore the live set is guaranteed to correspond to the $first$ instance of the stages appearing in that level but not deeper. The outermost loop level is an exception however. The live set is initialized to empty and therefore the instances of all the stages is guaranteed to be $last$.

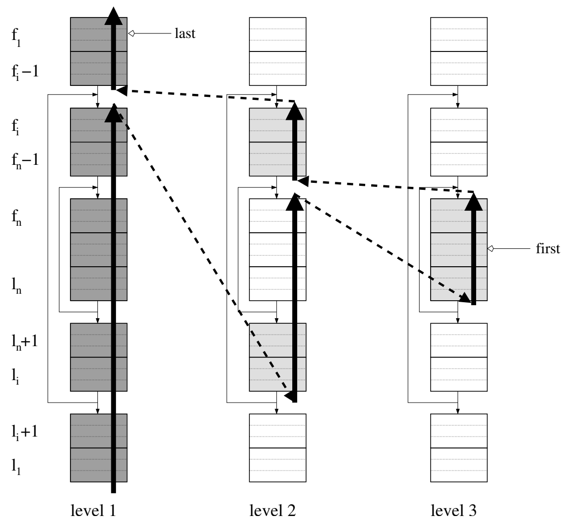


Figure 13: Stages Visit Order

The liveness analysis is done in Visit_Level(). $live$ first corresponds to the live-out set of current stage and cycle. After the first update of $live$, it corresponds to the actual live set for that stage and cycle. The number of variants in that set is stored (but not the live set itself). The second update computes the live-in set and will be used as the live-out for the next stage/cycle. In those updates, $DEF(s, c)$ and $USE(s, c)$ are the sets of non cross-iteration variants defined and used, respectively, by the operations scheduled in stage s and cycle c. Both sets are assumed to be empty if $c = T$.

4.3 Register Pressure

4.3.1 In The Outermost Level Pattern

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}:

$$\begin{aligned} LT_{olp}(c) &= LT_{cross}(c) \\ &+ \max_{i\in[1,S_n]}\left(\sum_{s=l_n-i}^{l_1} LT_{local}(s,c,last) + \sum_{s=f_1}^{l_n-1-i} LT_{local}(s,c,first)\right) \end{aligned} \tag{2}$$

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.

4.3.2 In The Inner Patterns

The formula to compute the register pressure in the inner patterns, LR_{inner}, is the sum:

$$\begin{aligned} LT_{inner}(c) &= LT_{cross}(T) + \sum_{s=l_n}^{l_1} LT_{local}(s,T,last) + \sum_{s=f_1}^{f_n-2} LR_{local}(s,T,first) \\ &+ \max_{l\in[2,n]}\left(\max_{i_0\in[0,S_l-1]}\left(\sum_{i=0}^{S_n-1} LT_{local}(f_l+(i_0+i)\%S_l,c,first)\right)\right) \end{aligned} \tag{3}$$

The first three terms correspond to the three types of stretched lifetimes shown in Figure 4: 7, 4, and 6 in that order. Their number is fixed for the entire execution of the inner patterns and equal to the number of lifetimes live at the exit of the OLP. Note the parameter T and not $T-1$ in all the terms on the first line of LT_{inner}. It is the number of live-out lifetimes at the end of the OLP, which might be different from the number of lifetimes at the last cycle of the OLP. In our example, there are one stretched lifetime of each type.

The fourth term of the formula , appearing on the second line of the formula, corresponds to the local lifetimes of the inner patterns(group 5 in Figure 4. For each inner level l, represented by the first max term, there are S_l different sequences of S_n stages, represented by the second max term. Given a loop level l, the sequences are all the possible combinations of S_n consecutive stages from the circular list of stages from l. For our example, at loop level 2, the list of stages is *d*-*c*-*b*, and there are $S_2 = 3$ possible sequences of consecutive stages: *d*-*c*, *c*-*b* and *b*-*d*. The exact formula to compute all the sequences given a loop level and a sequence index is the inner term of the sum on the second line of LT_{inner}.

4.3.3 MaxLive

$MaxLive$ is then the maximum between between the maximum register pressure of the OLP, $FatCover_{olp}$, and the maximum register pressure of the inner patterns, $FatCover_{inner}$:

$$FatCover_{olp} = \max_{\forall c\in[0,T-1]}(LT_{olp}(c)) \tag{4}$$

$$FatCover_{inner} = \max_{\forall c\in[0,T-1]}(LT_{inner}(c)) \tag{5}$$

$$MaxLive = max(FatCover_{inner}, FatCover_{olp}) \tag{6}$$

Note that, in the case of a single loop, the formula is still correct and $MaxLive$ is then equal to $FatCover_{olp}$.

4.4 Time Complexity

Both Compute_Cross_Iteration_LT and Compute_Local_LT runs in $O(S.II)$. Assuming that the loop variant sets are implemented using bit vectors, Visit_Level runs in $O(II)$ per stage. The routine is called once for each loop level l and visits S_l stages per call for a total of $\sum_{i=1}^{n} S_i$. Because $S > S_i, \forall i \in [1, n]$, the fast method algorithms run in $O(n.S.II)$.

The computation time of the formulas is bounded by the double maximum operators enclosing a sum in $LT_{inner}(c)$. Because $S_i \leq S, \forall i$, the computation time of the formula is bounded by $O(S^2.II.n)$. Therefore, the overall complexity of the fast method is bounded by: $O(S^2.II.n)$.

5 Comprehensive Register Pressure Computation

Unlike the fast method, the comprehensive method also takes into account live-in and live-out lifetimes. In order to do so, the schedule is unrolled and standard backwards data-flow liveness analysis is applied. However, to avoid redundant work, the minimum trip count for each loop level must be found. The algorithm proceeds by computing a minimum number of iterations for the outermost level, then applies liveness analysis to the corresponding unrolled schedule where the inner levels have a trip count of one or two.

5.1 Minimum Outermost Trip Count

As for single loop modulo scheduling, the final schedule can be decomposed into three phases: initialization, steady and conclusion. The initialization fills up the pipeline whereas the conclusion empties it. For our purpose, we will choose the size of the initialization and conclusion phase so that the initialization phase uses up the live-in variables and that the conclusion phase creates all the live-out variables. Note that that the initialization phase (conclusion phase, respectively) is composed of the prolog pattern (epilog pattern), but may also be composed of the OLP and the inner patterns. Figure 15 shows the different phases of the schedule from the loop nest proposed in Figure 5.

In the process of unrolling the schedule, our goal is to minimize the number of outermost iterations issued in each of the phases. The number of copies of the OLP in the steady phase is already fixed and equal to S_n. Therefore we must find a lower bound for the number of outermost iterations in the initialization and conclusion phases of the SSP schedule.

5.1.1 Initialization Phase

The number of outermost iterations issued during the initialization phase, denoted x, must satisfy the following conditions. First all the lifetimes of the live-in variables must have terminated. If $x_{LastInput}$ is

COMPUTE_MAXLIVE():
// *Initialize the live sets for each outer iteration*
$LIVE[N] \longleftarrow (\emptyset, \emptyset, \ldots, \emptyset)$
for each output variable v **do**
  for each i **from** N **to** $(N - alpha_{max}(v) + 1)$ **do**
    $LIVE[i] \longleftarrow LIVE[i] \cup \{v\}$

// *Backwards Data-Flow Liveness Analysis*
$MaxLive \longleftarrow \sum_{i=0}^{N-1} |LIVE[i]|$
$t \longleftarrow N + S - 1$
while $(t > omega_{max})$ **do**
  $t \longleftarrow t - 1$
  // *the outermost loop pattern (OLP)*
  Update_Live_OLP(t)
  // *the inner patterns (ILP or DFP)*
  if (the loop nest is not a single loop
    and $size_{prolog} + S_n \leq t - omega_{max} < N$
    and $(t - omega_{max} - size_{prolog}) \% S_n = 0$)
    Update_Live_Inner_Patterns(2,t)

UPDATE_LIVE_OLP(position t):
  for each cycle c backwards **from** $T - 1$ **to** 0 **do**
    for i **from** f_1 **to** l_1 **do**
      Update_Live(t-i,i,c,true)
    $MaxLive \longleftarrow max(MaxLive, \sum_{i=0}^{N-1} |LIVE[i]|$
    for i **from** f_1 **to** l_1 **do**
      Update_Live(t-i,i,c,false)

UPDATE_LIVE_INNER_PATTERNS(level l, position t):
  if $l == n$ **then**
    Update_Live_Inner_Level(n,t)
  else
    Update_Live_Inner_Patterns(l+1,t)
    Update_Live_Inner_Level(l,t)
    Update_Live_Inner_Patterns(l+1,t)
    Update_Live_Inner_Level(l,t)
    Update_Live_Inner_Patterns(l+1,t)

UPDATE_LIVE_INNER_LEVEL(level l, position t):
  for i from 0 to $S_l - 1$ **do**
    $firstStage \longleftarrow f_n - 1 - i$
    $firstIndex \longleftarrow t - (f_n - f_1)$
    for each cycle c backwards **from** $T - 1$ **to** 0 **do**
      for $iter$ from 0 to $S_n - 1$ **do**
        $index \longleftarrow firstIndex - iter$
        $stage \longleftarrow firstStage + iter$
        if $stage < f_l$ **then**
          $stage \longleftarrow stage + S_l$
        Update_Live(index,stage,c,true)
      $MaxLive \longleftarrow max(MaxLive, \sum_{i=0}^{N-1} |LIVE[i]|$
      for $iter$ from 0 to $S_n - 1$ **do**
        $index \longleftarrow firstIndex - iter$
        $stage \longleftarrow firstStage + iter$
        if $stage < f_l$ **then**
          $stage \longleftarrow stage + S_l$
        Update_Live(index,stage,c,false)

UPDATE_LIVE(index i, stage s, cycle c, boolean $before$):
  if $before$ **then**
    if $omega_{max} \leq i < N$ **then**
      for each definition d in stage s at cycle c **do**
        $LIVE[i] \longleftarrow LIVE[i] \cup \{d\}$
    for each use u in stage s at cycle c **do**
      if ($omega(u) = 0$ **and** $omega_{max} \leq i < N$)
       or ($omega(u) > 0$ **and** $0 \leq i - omega(u) < N$ **then**
        $LIVE[i - omega(u)] \longleftarrow LIVE[i - omega(u)] \cup \{u\}$
  else
    if $omega_{max} \leq i < N$ **then**
      for each definition d in stage s at cycle c **do**
        $LIVE[i] \longleftarrow LIVE[i] - \{d\}$
    for each use u in stage s at cycle c **do**
      if ($omega(u) = 0$ **and** $omega_{max} \leq i < N$)
        or ($omega(u) > 0$ **and** $0 \leq i - omega(u) < N$ **then**
        $LIVE[i - omega(u)] \longleftarrow LIVE[i - omega(u)] \cup \{u\}$

Figure 14: Comprehensive Method Algorithms

the last outermost iteration issued when the last live-in lifetime terminates, then we have: $x_{LastInput} = max_{u \in U_{input}} (stage(u) + omega(u) - 1)$,where U_{input} is the set of all uses of live-in variables in the schedule. For instance, if a live-in value is used in stage c with $omega = 2$, the index of the latest outermost iteration issued is $2 + omega - 1 = 3$. Indeed, in Figure 15, the last occurrence of c in the second outermost iteration, represented in grey, appears in the same OLP than the fourth occurrence of stage a. Second, the OLP should not be truncated in the steady phase. Therefore, at least S outermost iterations should be issued. Third and last, the schedule should be legal. Therefore the initialization phase should issue $size_{prolog} = f_n - 1$ iterations for the prolog and a multiple of S_n iterations for the necessary OLP patterns. Overall we must find the minimum integer p such that:

$$\begin{cases} x \geq x_{LastInput} \\ x \geq S \\ x = (f_n - 1) + p.S_n \end{cases}$$

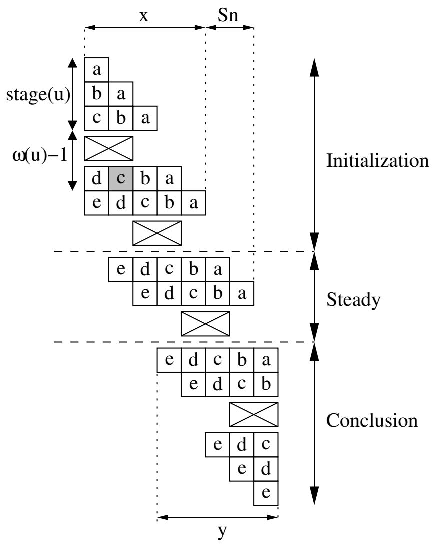


Figure 15: Schedule Phases

Which leads to the lower bound: $x = (f_n - 1) + \left\lceil \frac{max(S, x_{LastInput}) - (f_n - 1)}{S_n} \right\rceil .S_n$

5.1.2 Conclusion Phase

To compute y, the minimum number of outermost iterations in the conclusion phase, two conditions must be satisfied. First, all the live-out lifetimes must start from the conclusion phase. If $y_{FirstOutput}$ is the first outermost iteration to start a live-out lifetime, then we have $y_{FirstOutput} = max_{d \in D_{output}} (l_1 - stage(d) + alpha(d))$, where D_{output} is the set of all definitions of live-out variables in the schedule. Second, the schedule must finish properly without truncating any inner pattern. Therefore the conclusion phase should issue $size_{epilog} = l_1 - f_n + 1$ iterations for the epilog and a multiple of S_n iterations for the necessary OLP patterns. Overall we must find the minimum integer p such that:

$$\begin{cases} y \geq y_{FirstOutput} \\ y = (l_1 - f_n + 1) + p.S_n \end{cases}$$

Which leads to the lower bound $y = (l_1 - f_n + 1) + \left\lceil \frac{y_{FirstOutput} - (l_1 - f_n + 1)}{S_n} \right\rceil .S_n$

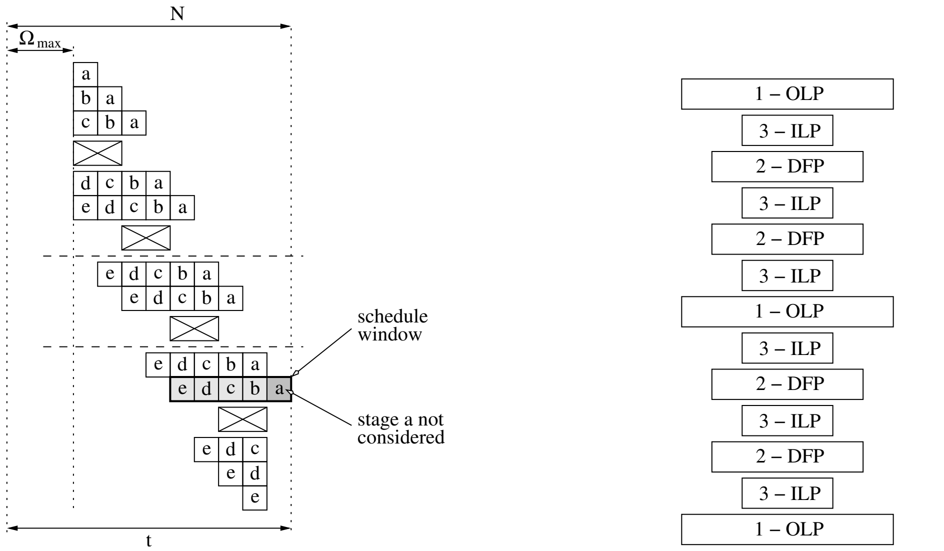
5.1.3 Minimum Outermost Trip Count

The total number of iterations that need to be considered is the sum of x, S_n and whatever iteration in y that has not yet been accounted for. The iterations counted by y can be subdivided into 3 groups. The first group includes the outermost iterations appearing in the inner patterns of the steady phase or before. It is equal to $size_{epilog} = l_1 - f_n + 1$. The second group includes the iterations issued in the OLP patterns of the steady phase and not counted yet $f_n - f_1 - 1 = f_n - 1$. The last group is the rest: $y - size_{epilog} - f_n - 1 = y - l_1$. Therefore the total number of necessary iterations is equal to $x + y + S_n - l_1$.

However, because live-in lifetimes virtually belongs to previous iterations, we must add extra iterations. If $omega_{max}$ is the maximum $omega$ value encountered in the schedule, then the minimum outermost trip count becomes $N = x + y + S_n - l_1 + omega_{max}$.

5.2 Register Pressure Computation

The computation of the register pressure uses the final schedule with N iterations, as computed in the previous section. Each outermost iteration is given its own live set initialized with the live-out variables defined by the iteration itself. A standard backwards data-flow liveness analysis algorithm is used for each of the outermost iteration. If an operation uses a variable defined in a previous outermost iteration, then the variable is added to the live set of that previous iteration. The algorithms are shown in Figure 14.



(a) Schedule Window Example

(b) Patterns Apparition Order

Figure 16: Comprehensive Method Algorithm Examples

The analysis proceeds up to S iterations at a time using a schedule window as shown on Figure 16(a) and equivalent to the SSP kernel. The position of the window, t, is determined by position of its first stage a from the first outermost iteration. Once the stages have been analyzed for the current position by *Update_Live_OLP()*, t is decreased by one. When the inner patterns are encountered (every S_n iterations after the prolog), *Update_Live_Inner_Patterns()* analyzes them. The liveness analysis update the live set of the outermost iteration being processed.

At each cycle, the total number of lifetimes is computed and the highest value stored in $MaxLive$. The liveness information is only temporarily stored for computation purposes and is therefore not available at the end of the algorithm.

5.2.1 Outermost Loop Pattern

The register pressure for the OLP is evaluated in *Update_Live_OLP()*. First the live set for the current cycle is computed by adding the definitions and uses of the operations scheduled for this cycle, using the $before$ flag of *Update_Live()*. Then the register pressure for the cycle is measured. If greater than the highest value recorded, we store the new value. The live set for the beginning of the cycle is then computed by another call to *Update_Live()* with the $after$ flag this time to add the uses but remove the definitions, following the conventions described in Section 3.1.

Note that *Update_Live()* updates the live set of the outermost iterations corresponding to live-in variables, only if a use with an $omega$ value greater than 0 is encountered.

5.2.2 Inner Patterns

When computing the register pressure for the inner patterns, we face two issues. First, all the lifetimes must be accounted for. Therefore each pattern must appear at least twice in each situation, the first appearance being used for initialization. Second, within each pattern the exact combination of S_n stages must be used.

The first issue is solved in *Update_Live_Inner_Patterns()* by using a schedule such as the one shown in Figure 16(b) for our 3-deep loop nest example. For each OLP, the second outermost DFP appears twice. If the loop nest was deeper, each DFP pattern would appear twice for each of DFP appearance of the upper level. The ILP is a special case and only need to appear once. The DFP and ILP patterns between two OLPs corresponds to the crossed boxes in Figure 16(a).

The second issue is solved by using the following formula. If l is the level considered, and i the iteration index within the group of S_n outermost iterations, the stages would be given by the formula: $l_l - (l_l - (f_n - 1) + i + j)\%S_l$ where $0 < j < S_l$ (there are S_l instances of the steady phase in a DFP of level l). In *Update_Live_Inner_Level()*, the result is stored in $stage$.

6 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 were loop nests extracted from the Livermore Loops [12], the NPB 2.2 benchmarks [17] and the SPEC2000 FP benchmark suite [27]. 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. Time and register pressure were measured for each of the method. As defined earlier, the register pressure corresponds to the maximum number of lifetimes at any point in the schedule for a given rotating register class, integer or floating-point. The compile-time measurement is the time required by the featured algorithm to compute $MaxLive$, in seconds.

A third method, not mentioned here, was also implemented. The method consists of unrolling the full schedule using lifetimes information already computed by the register allocator. Although brutal and much slower, it allowed to check the correctness of the two other implementations.

The main results are summarized below and explained in details in the next subsections:

- The fast method is 1 to 2 orders of magnitude faster than the comprehensive method, 3 to 4 orders of magnitude faster than the register allocator, and takes few hundreds of a second to execute.

- Despite the approximations made by the fast method, its computed register pressure is identical to the register pressure computed by the comprehensive method. No rule of thumb could be deduced to predict the register pressure by only considering the schedule parameters such as schedule size, 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 register pressure higher than the size of the register file.

- 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.

6.1 Compilation Time

The time measurements are presented in Figure 17 where the loop nests have been sorted by increasing depth, starting at 1 and delimited by tics on the horizontal axis. Moreover, loop nests of identical depth are sorted by increasing schedule size. Note that all the time measurements have a logarithmic scale for 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 schedule size. The shape of the graph (with a logarithmic vertical scale) confirms the quadratic running time of the fast method and the influence of the depth of the loop nest.

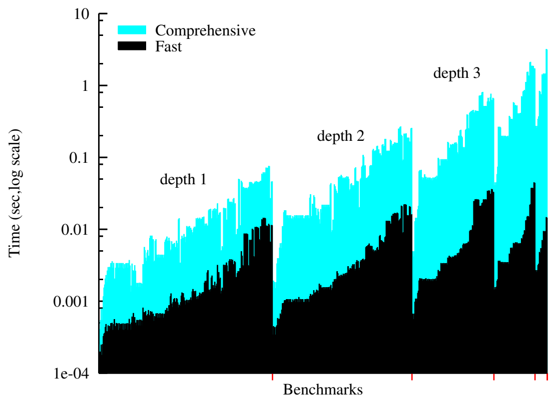
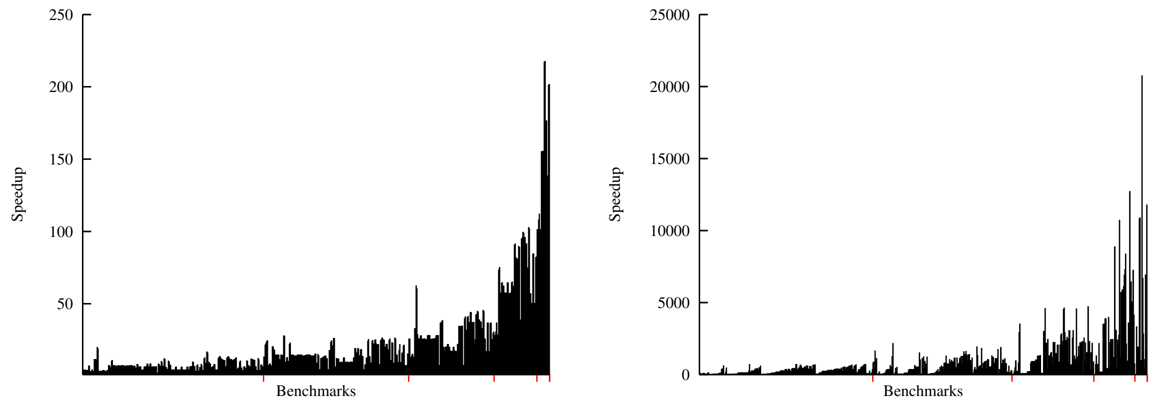


Figure 17: Absolute Running Time of each Method

The fast method, when compared to the comprehensive method in Figure 18(a), shows a net running-time improvement. On average, the fast method is 22.9 times faster with a maximum of 217.8. As the loop nest gets deeper, the speedup becomes exponentially more significant. Although the fast method does not consider live-in and live-out ranges, its computed register pressure matches the register pressure computed by the comprehensive method. Both methods can then be used to measure the register pressure of a loop nest scheduled with SSP and evaluate the quality of the solutions computed by any SSP register allocator. However, because of its speed, the fast method is the method of choice.



(a) vs. the Comprehensive Method

(b) vs. the Register Allocator

Figure 18: Fast Method Speedup

The running time of the fast method is measured against the running time of the register allocator from [23] in Figure 18(b), where the test cases have been again sorted by depth. On average, the fast method is 3 orders of magnitude faster than the register allocator with a maximum of 20000. More importantly, as the loop nest gets deeper, i.e. as the register pressure increases and the need for a quick method to evaluate the register pressure a priori becomes stronger, the speedup tends to increase. Therefore the fast method is a valid tool that can be used in the SSP framework to detect early in the compilation process if a schedule is infeasible and to avoid calling in vain the register allocator.

Although the fast method does not take into account live-in and live-out lifetimes, the computed register pressure was identical for the two other methods in all the benchmarks tested. Despite the relatively low number of such lifetimes, we can conclude that the number of live-in and live-out variables is relatively limited and therefore have no influence of the overall register pressure of the loop nest. Intuitively, it is also due to the fact that the corresponding lifetimes are most likely to appear in the prolog and epilog where only a subset of the loop variants are live.

6.2 Register Pressure

It is important to first note that the computed register pressure is an 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. Also $MaxLive$ makes the difference between the $first$ and $last$ occurrence of the same stage. In other words, $MaxLive$ assumes a 0% fragmentation level in the Itanium register stack as if copy instructions were inserted between each cycle of the schedule to compact the register stack.The actual register allocation solution computed by an ideal register allocator may allocate more registers than $MaxLive$.

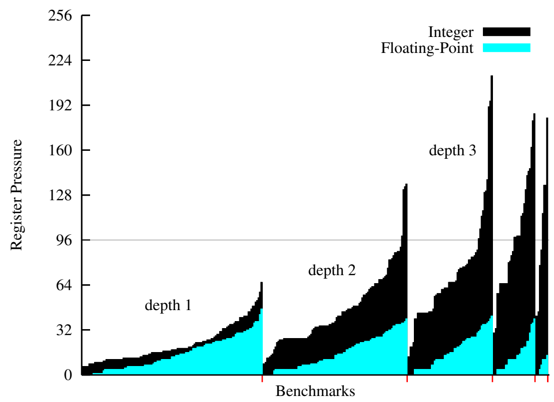


Figure 19: Register Pressure

The register pressure computed by the three methods is shown in Figure 19 for integer and floating-point loop variants. The benchmarks have been sorted by increasing depth, indicated by small tics on the horizontal axis, and by increasing register pressure. The average register pressure for integer and floating-points 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 floating-point registers in the Itanium architecture is never reached. However the 96 limit for integer 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 20.

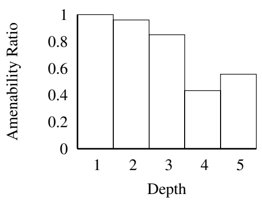


Figure 20: Ratio of Loop Nests Amenable to SSP

We note that the integer register pressure increases faster than the floating-point register pressure. Intuitively, the register pressure of the schedule increases as the nest gets deeper because more inner iterations are running in parallel. It is particularly true for integer 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 schedule. For floating-point values however, this is not the case. They are typically defined in the innermost loop only and have very short lifetimes. Even if many outermost iterations are running in parallel, only few floating-point are live at any point in time.

We also tried to approximate the register pressure by looking at the schedule parameters. However no rule of thumb could be derived by looking at one parameter such as S, S_n, the size of the schedule or the number of loop variants. Each parameter influences the register pressure but in a different manner for each benchmark. The register pressure was also compared to the actual number of registers allocated by the register allocator. Unfortunately, the number of allocated registers vary between 0% and 77% over $MaxLive$ for each depth, and no correlation could be found.

6.3 Floating-Point Register File Size

When looking at the total register pressure defined as the sum of the integer and floating-point register pressures, we note that it rarely exceeds 192 registers, the total number of rotating registers in the Itanium architecture as shown in Figure 21. The left and right vertical axis represent the total register pressure and the ratio between floating-point register pressure and integer register pressure respectively. The benchmarks are sorted by increasing ratio. It is worth noting that, although the floating-point register pressure can be twice higher than the integer register pressure, when the total register pressure is greater than 96, the floating-point/integer ratio remains lower than 0.5.

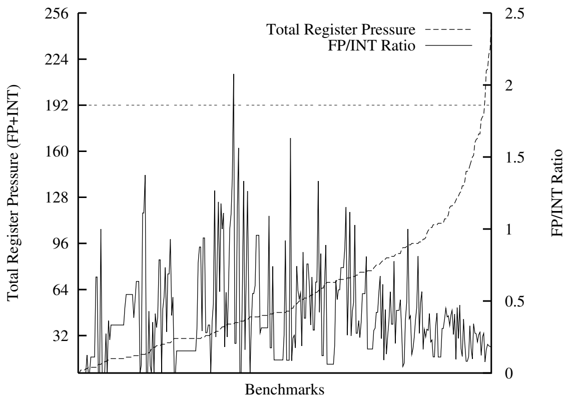


Figure 21: Total Register Pressure and FP/INT Register Pressure Ratio

Moreover, Figure 22 shows the floating-point register pressure as the same loop nest is scheduled at deeper levels. Very interestingly, the floating-point register pressure does not or barely increases when the same loop nest is scheduled at a deeper level. The maximum floating-point register pressure never exceeds 47 registers.

Therefore several conclusions that may be useful for future architecture designs with SSP in mind, can be drawn from these remarks. First, the integer register file may benefit from a smaller floating-point register file with a ratio of 2 for 1. The floating-point register size can either be decreased to save important chip real estate, or the integer 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 floating-point register file would be 64. It would not prevent any other loop nests from being register allocated while giving extra registers to the integer 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.

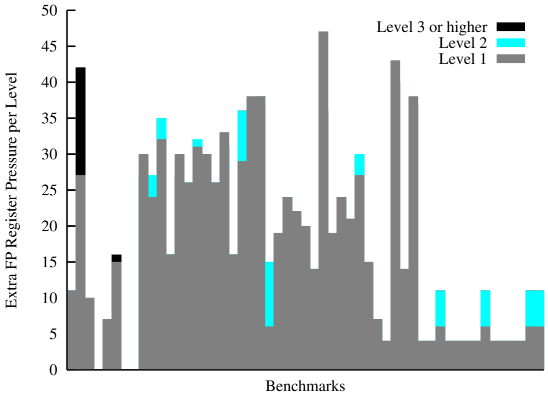


Figure 22: Floating-Point Register Pressure Progression

7 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, and would provide a valuable feedback to processor architects in their register files design decision.

We proposed in this paper two methodologies to compute the rotating register pressure of any SSP schedule and provides feedback to the loop selector when the register pressure is too high. One method, comprehensive, accounts for every lifetime but is slower. The fast method makes the assumption that the maximum register pressure will appear in the steady phase of the schedule.

Results showed that despite the approximations made, the computed register pressure by the fast method is accurate. It is at least 3 orders of magnitude faster than the register allocator on average, making it a valid tool to detect infeasible schedules early. The high register pressure measured in the experiments confirms the need for a fast method to evaluate the register pressure a priori. Finally, 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.

8 Acknowledgments

The authors would like to thank Jean-Christophe Beyler and anonymous reviewers for their helpful comments.

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 International Conference on System Sciences*, pages 183–192, 1996.

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

[4] A. Darte, R. Schreiber, B. R. Rau, and F. Vivien. Constructing and exploiting linear schedules with prescribed parallelism. *ACM Transactions on Design Automation of Electronic Systems*, 2001.

[5] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Minimum register requirements for a modulo schedule. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 75–84. ACM Press, 1994.

[6] L. J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. In *CC '92: Proceedings of the 4th International Conference on Compiler Construction*, pages 176–191. Springer-Verlag, 1992.

[7] R. A. Huff. Lifetime-sensitive modulo scheduling. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 258–267. ACM Press, 1993.

[8] S. Jain. Circular scheduling: A new technique to perform software pipelining. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 219–228, Toronto, Ontario, June 26–28, 1991. *SIGPLAN Notices,* 26(6), June 1991.

[9] 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, Atlanta, Georgia, June 22–24, 1988. *SIGPLAN Notices,* 23(7), July 1988.

[10] 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.

[11] J. Llosa, A. González, E. Ayguadé, and M. Valero. Swing modulo scheduling: A lifetime sensitive approach. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 80–86, october 1996.

[12] F. McMahon. The livermore fortran kernels: A computer test of the numerical performance range. Technical Report Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, December 1986.

[13] S.-M. Moon and K. Ebcioğlu. Parallelizing nonnumerical code with selective scheduling and software pipelining. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):853–898, 1997.

[14] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1997.

[15] K. Muthukumar and G. Doshi. Software pipelining of nested loops. *Lecture Notes in Computer Science*, 2027:165–??, 2001.

[16] K. Muthukumar and G. Doshi. Software pipelining of nested loops. In *Proceedings of the 10th International Conference on Compiler Construction, CC 2001*, volume 2027 in Lecture Notes in Computer Science, pages 165–181, 2001.

[17] NASA. NAS Parallel Benchmarks (NPB). http://www.nas.nasa.gov/Research/Tasks/pbn.html.

[18] Q. Ning and G. R. Gao. A novel framework of register allocation for software pipelining. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 29–42. ACM Press, 1993.

[19] Open64 Compiler and Tools. http://open64.sourceforge.net/, January 2005.

[20] Open Research Compiler for Itanium Processor Family. http://ipf-orc.sourceforge.net/, January 2005.

[21] B. R. 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. ACM Press, 1994.

[22] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 283–299. ACM Press, 1992.

[23] H. Rong, A. Douillet, and G. R. Gao. Register allocation for software pipelined multi-dimensional loops. In *PLDI '05: Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation*. ACM Press, 2005.

[24] 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. IEEE Computer Society, 2004.

[25] 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. IEEE Computer Society, 2004.

[26] J. Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown: optimal vs. heuristic methods in a production compiler. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 1–11. ACM Press, 1996.

[27] SPEC Standard Performance Evaluation Corp. SPEC CPU2000 Benchmark Suite. http://www.spec.org/cpu2000/, 2000.

[28] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Two-level hierarchical register file organization for vliw processors. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 137–146. ACM Press, 2000.