Register Allocation for Software Pipelined Multi-dimensional Loops

Hongbo Rong Alban Douillet Guang R. Gao

Department of Electrical and Computer Engineering, University of Delaware, Newark, DE 19716-3130

{rong, douillet, ggao}@capsl.udel.edu

Abstract

Software pipelining of a multi-dimensional loop is an important optimization that overlaps the execution of successive outermost loop iterations to explore instruction-level parallelism from the entire n-dimensional iteration space. This paper investigates register allocation for software pipelined multi-dimensional loops.

For single loop software pipelining, the lifetime instances of a loop variant in successive iterations of the loop form a repetitive pattern. An effective register allocation method is to represent the pattern as a vector of lifetimes (or a *vector lifetime* using Rau's terminology) and map it to rotating registers. Unfortunately, the software pipelined schedule of a multi-dimensional loop is considerably more complex, and so are the vector lifetimes in it.

In this paper, we develop a way to normalize and represent vector lifetimes in multi-dimensional loop software pipelining, which capture their complexity, while exposing their regularity that enables us to develop a simple, yet powerful solution. Our algorithm is based on the development of a metric, called $distance$, that quantitatively determines the degree of potential overlapping (conflicts) between two vector lifetimes. We show how to calculate and use the distance, conservatively or aggressively, to guide the register allocation of the vector lifetimes under a bin-packing algorithm framework. The classical register allocation for software pipelined single loops is subsumed by our method as a special case.

The method has been implemented in the ORC compiler and produced code for the Itanium architecture. We report the effectiveness of our method on 134 loop nests with 348 loop levels. Several strategies for register allocation are compared and analyzed.

Categories and Subject Descriptors D.3.4 [*PROGRAMMING LANGUAGES*]: Processors—Compilers, Optimization

General Terms Algorithms, Languages

Keywords Software Pipelining, Register Allocation

1. Introduction

Software pipelining of a multi-dimensional loop nest overlaps the execution of successive outermost loop iterations to explore instruction-level parallelism from the entire n-dimensional iteration space. The traditional approaches mainly focus on scheduling of the innermost loop, and extend the schedule toward an outer loop by hierarchical reduction [9, 11]. An alternative way is to perform innermost loop software pipelining after loop transformations [3].

PLDI'05 June 12–15, 2005, Chicago, Illinois, USA.

Single-dimension Software Pipelining (SSP) [14, 13] is a unique resource-constrained software pipelining framework that overlaps the iterations of an n-dimensional (n-D) loop. It simplifies the n-D loop scheduling problem into a 1-D (1-dimensional) loop scheduling problem, produces a 1-D schedule, and maps it back to the n-D iteration space to generate a parallelized loop nest. The classical modulo scheduling [1] is subsumed by SSP as a special case.

This paper investigates register allocation for software pipelined loop nests by SSP. To the best of our knowledge, no study has ever been reported on the topic. Existing methods for allocating registers for loop nests [6, 2] are extensions of the traditional graph coloring approach [4] and do not aim at software pipelining. However, lifetimes in a software pipelined loop have regular patterns [12], which should be taken advantage of by an efficient register allocator. Traditional software pipelining of loop nests [9, 11] centers around scheduling, with little discussion on register allocation.

For single loop software pipelining, the lifetime instances of a loop variant in successive iterations of the loop form a repetitive pattern. An effective register allocation method is to represent the pattern as a vector of lifetimes (or a *vector lifetime* using Rau's terminology) and map it to rotating registers [12, 5]. Unfortunately, the software pipelined schedule of a loop nest is considerably more complex, and so are the vector lifetimes in it, which leads to interesting and serious challenges for register allocation, e.g., *how to represent such vector lifetimes? How close and how far can two vector lifetimes be put together without conflict? How to identify legal and illegal registers for a vector lifetime? And what strategy should we use to minimize the number of allocated registers?*

In this paper, we develop a way to normalize and represent vector lifetimes, which capture their complexity in multi-dimensional loop software pipelining, while exposing their regularity that enables us to develop a simple, yet powerful solution. The problem is formulated as bin-packing of the multi-dimensional vector lifetimes on the surface of a cylinder, with time as axis and registers as the circle, such that there is no conflict between lifetimes and the circumference of the circle is minimized.

Our algorithm is based on the development of a metric, called $distance$, that determines quantitatively the degree of potential overlapping (conflict) between two vector lifetimes. We show how to calculate and use the distance, conservatively and aggressively, to guide the bin-packing of the lifetimes. The classical register allocation for software pipelined single loops [12, 5] is subsumed by our method as a special case.

This approach has been implemented in the ORC compiler for Itanium architecture. Experiments indicate impressive effectiveness of our approach in minimizing the number of allocated registers. Several allocation strategies are compared and analyzed.

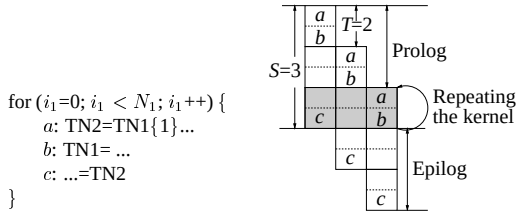
2. Register Allocation For Software Pipelined Single Loops

2.1 Software Pipelining

Software pipelining [1] exposes instruction-level parallelism by overlapping successive iterations of a loop. **Modulo scheduling** [9,

7, 8] is an important and probably the most commonly used approach of software pipelining.

In modulo scheduling, instances of an operation from successive iterations are scheduled with an **Initiation Interval** (II) of T cycles. The schedule length l is defined as the execution time of a single iteration. Then each iteration is composed of $S = \lceil \frac{l}{T} \rceil$ number of **stages**, with each stage taking T cycles. The schedule consists of three phases: the **prolog** to fill the pipeline, the **kernel** to be executed multiple times, and the **epilog** to drain the pipeline.



(a) Source (b) Modulo Schedule

Figure 1. A Software-Pipelined Loop

Example: Fig. 1(a) shows an example intermediate representation of a single loop, where a **Temporary Name** (TN) represents a variant. If a TN value is used i iterations after where it is produced, it has a **live-in distance** equal to i. The TN value is annotated with the live-in distance. For example, TN$\{1\}$ refers to the TN value defined in the previous loop iteration, with the live-in distance being 1. Suppose there are two function units, and operations a, b and c have a latency of 5, 1, and 1 cycles, respectively. Then a possible modulo schedule is shown in Fig. 1(b) with $T = 2$, $l = 6$ and $S = 3$. The kernel is highlighted in darker color.

2.2 Register Allocation

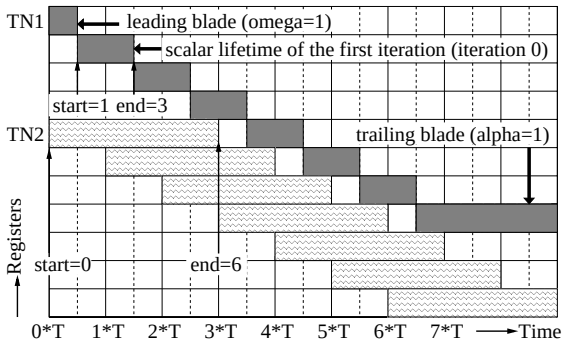
We briefly review the classical register allocation for software-pipelined single loops [12, 5], with target architecture support in the form of a rotating register file [12, 8].

A **scalar lifetime** is the lifetime of a loop variant for a given iteration of the loop. The variant has one operation to produce a value and one or more operations to consume the value. The scalar lifetime starts when the producer is issued and ends when all of the consumers have finished. All the scalar lifetimes of the loop variant over all the iterations of the loop compose the **vector lifetime** of the loop variant. The vector lifetime can be represented on a **space-time diagram**, where time is on the horizontal axis and the registers on the vertical axis. For example, the space-time diagram for TN1 and TN2 in Fig. 1 is shown in Fig. 2(a).

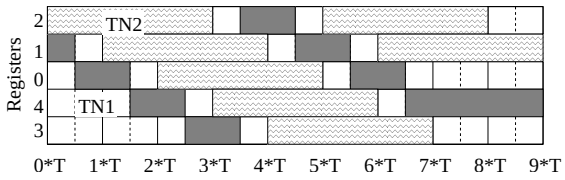
A vector lifetime is composed of a **wand** (the diagonal band), a **leading blade** in case of live-in values, and a **trailing blade** in case of live-out values. In Fig. 2(a), TN2 is made of a wand only. TN1 has a leading blade for it has a live-in value, and a trailing blade (Assume it has a live-out value).

Correspondingly, a vector lifetime is represented by a 4-tuple **(start, end, omega, alpha)**. The $start$ and end values refer to the start and end cycles of the scalar lifetime corresponding to the first iteration of the loop, i.e., iteration 0. Then, the scalar lifetime corresponding to iteration i starts at $start + i * T$ and ends at $end + i * T$. $Omega$ is the number of live-in values for the loop variant. It is also the maximum live-in distance of the loop variant. $Alpha$ represents the number of live-out values for the loop variant. For example, the vector lifetimes of TN1 and TN2 are represented as (1, 3, 1, 1) and (0, 6, 0, 0), respectively.

A physical register x is said to be **allocated** to a vector lifetime v if it is allocated to the scalar lifetime of v corresponding to the first iteration. The next physical register $x - 1$ is then allocated to the scalar lifetime corresponding to the second iteration, and so on. Due to the cyclic nature of the rotating register file, the register index wraps around to the highest index when it becomes -1.



(a) Space-Time Diagram for the Example in Fig. 1(a) (with $N_1 = 7$)



(b) An Optimal Bin-Packing (Circumference=5)

Figure 2. Register Allocation

For the same reason, a space-time diagram can be seen as a **cylinder** with time as the axis and registers as the circle. The **circumference** of the circle is the total number of registers required to allocate to the loop. The register allocation problem consists of packing the vector lifetimes on the surface of the cylinder, such that there is no conflict and the circumference is minimized. **Conflict** refers to the fact that two scalar lifetimes that overlap in time are allocated to the same register. An optimal register allocation for the space-time diagram in Fig. 2(a) is displayed in Fig. 2(b), where TN1 is allocated physical register 0, and TN2 register 2, with a circumference of 5 after the diagram is wrapped into a cylinder.

To achieve such register allocation, the algorithm sorts the vector lifetimes and inserts them one by one on the surface of the space-time cylinder without backtracking. Three sorting heuristics can be used: **start time ordering**, where the earliest vector lifetime is inserted first; **adjacency ordering**, where the vector lifetime to be inserted minimizes the horizontal distance with the previously inserted lifetime; and **conflict ordering**, which is to vector lifetimes what graph coloring [4] is to scalar lifetimes. The insertion of the chosen lifetime is then decided by one of three **strategies**: best, first, and end fits. **Best fit** finds a register that minimizes the current register usage. **First fit** chooses the first compatible register starting from register 0, while **end fit** starting from the register allocated to the vector lifetime inserted at the last step.

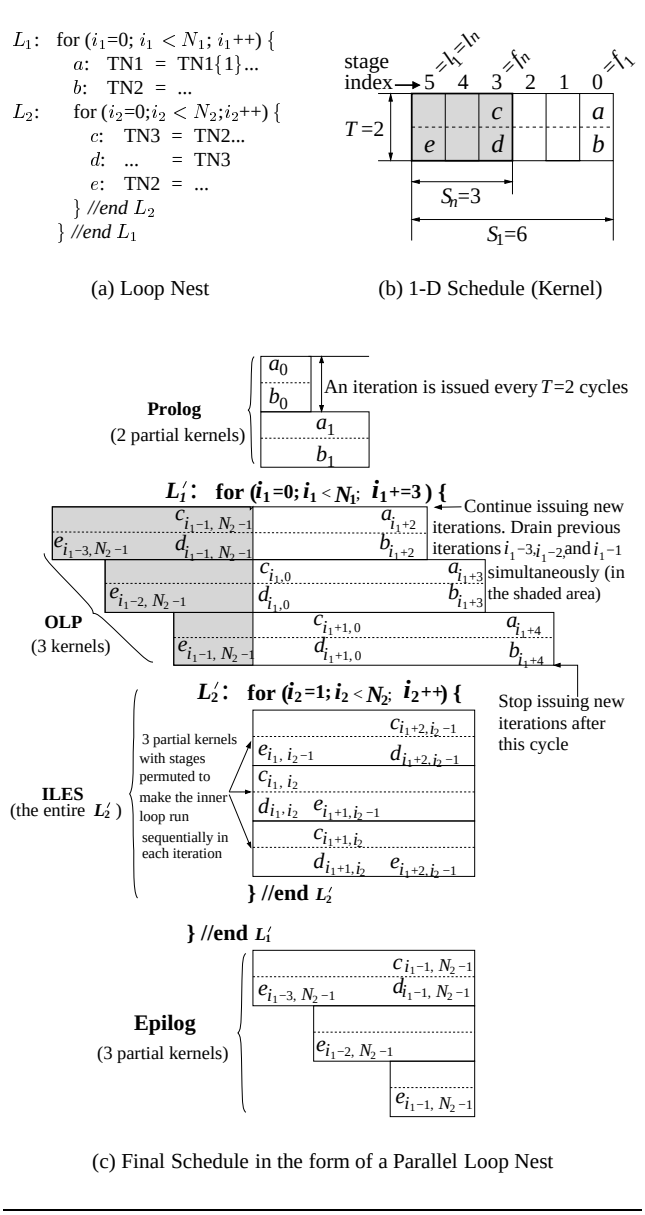
Note that the register allocation problem can be formulated as a Traveling Salesman Problem (TSP) as well, which is known to be NP-Complete [12].

3. Register Allocation For Software Pipelined Loop Nests

This section briefly reviews SSP, characterizes the vector lifetimes in an SSP schedule, formulates the corresponding register allocation problem, and finally proposes key observations to solve it.

3.1 Single-dimension Software Pipelining

Single-dimension Software Pipelining (SSP) [14, 13] is a resource-constrained scheduling method to software pipeline a loop nest. In



(a) Loop Nest (b) 1-D Schedule (Kernel)

(c) Final Schedule in the form of a Parallel Loop Nest

Figure 3. A Loop Nest Scheduled by SSP

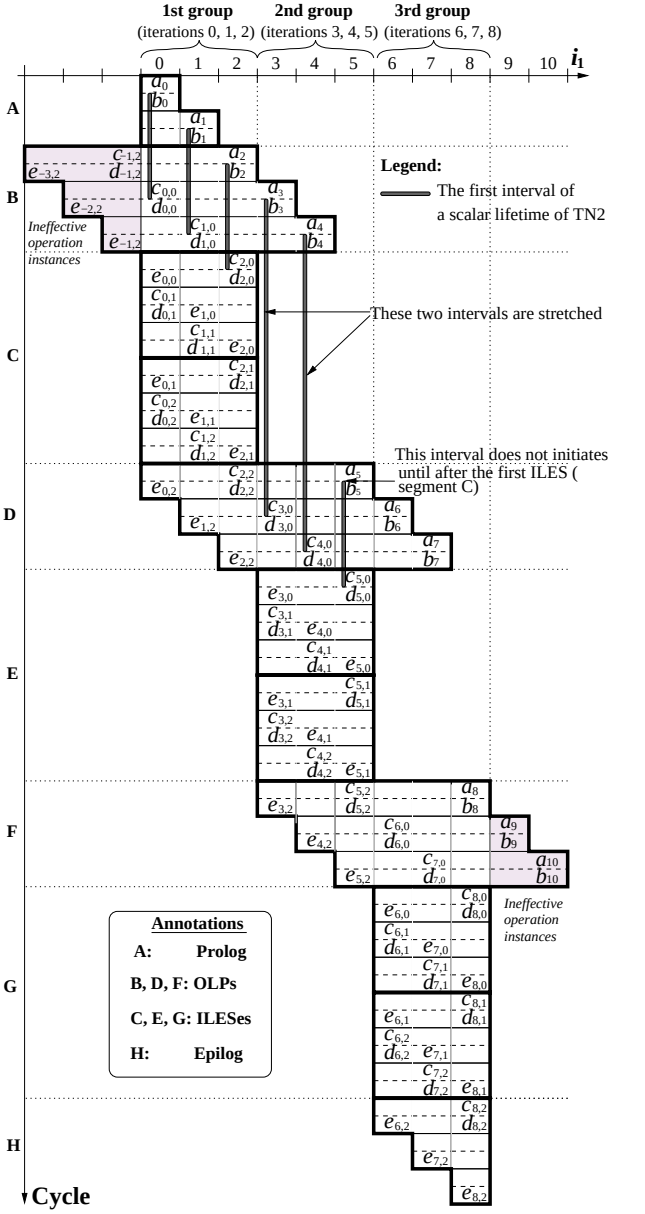


Figure 4. Final Schedule (with $N_1 = 9$, $N_2 = 3$)

contrast to traditional innermost-loop-centric approaches [9, 11], SSP chooses the most profitable loop level from the entire loop nest. Here profitability can be measured in terms of instruction-level parallelism, data reuse, or any other optimization criteria. SSP retains the simplicity of the classical modulo scheduling technique for single loops, yet under the same condition, achieves the shortest computation time that could be achieved by traditional innermost-centric approaches [14]. The classical modulo-scheduling is subsumed by SSP as a special case.

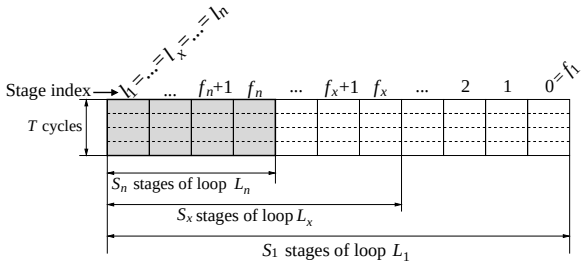
SSP consists of three steps: (1) **Loop selection**. A loop in the loop nest is chosen to be software pipelined. Only this loop will be parallelized. Its outer loops, if any, remain intact. Note that this selected loop may have its own inner loops and thus be a loop nest itself. (2) **Dependence simplification and 1-D schedule construction**. The n-dimensional ($n \geq 1$) data dependence graph (DDG) of the selected loop is simplified into a 1-dimensional DDG, and based on that, a **1-dimensional schedule** is computed, represented by a **kernel**. No matter how many inner loops the selected loop has, it is scheduled as if it were a single loop. Any traditional modulo scheduling technique may be applied to compute this 1-D schedule. (3) **Final schedule computation**. The 1-D schedule is mapped back to the n-dimensional iteration space to form the **final schedule**, semantically equivalent to the selected loop.

This approach is referred to as Single-dimension Software Pipelining because the problem of multi-dimensional scheduling is simplified to 1-dimensional scheduling and mapping.

Intuitively, in the final schedule, the iterations of the selected loop are issued in parallel, whereas the inner loops within each of the iterations run sequentially. Let S_n be the total number of stages corresponding to the innermost loop in the kernel, and T be the initiation interval of the kernel. Every T cycles, an iteration of the selected loop is issued, until the processor resources become insufficient to support any new iteration. Then a single **group** of S_n iterations, already issued, execute their inner loops in parallel. Until this group is going to finish and frees the resources, all the

```
L1: for (i1=0; i1 < N1; i1++) {
        OPSET1
L2:     for (i2=0; i2 < N2; i2++) {
            OPSET2
            ...
Ln:             for (in=0; in < Nn; in++) {
                    OPSETn
                } //end Ln
            ...
        } end L2
    } //end L1
```

(a) Generic Source Loop Nest



(b) Generic Kernel

```
    Prolog
L1': for (i1=0; i1 < N1; i1+= Sn) {
        OLP
        ILES
    }
    Epilog
```

(c) The General Form of a Final Schedule

Figure 5. Generic Loop Nest Scheduled by SSP

other iterations stall. In this paper, such a stall period is referred to as an **inner loop execution segment (ILES)**.

Example: Fig. 3(a) shows the intermediate representation of a double loop nest. Assume that L_1 is selected by SSP, 3 function units are available, and operations a, b, c, d, e have latencies of 2, 5, 1, 4, and 1 cycles, respectively. Fig. 3(b) shows a possible 1-D schedule (the kernel) constructed. From this kernel, a final schedule is computed and the loop nest can be rewritten accordingly, as shown in Fig. 3(c), where o_{i_1,i_2} refers to the instance of operation o with i_1 the outer loop index and i_2 the inner loop index. An L_1 iteration is issued every $T = 2$ cycles, and executes sequentially. The issuing continues until the 10th cycle. Otherwise, there would be resource conflicts with the already running L_1 iterations. Then, only the first group of $S_n = 3$ L_1 iterations continue executing their inner loops. Once done, the group starts draining the pipeline and freeing processor resources. Then other L_1 iterations continue to issue and execute. To facilitate understanding, the final schedule in Fig. 3(c) is illustrated cycle by cycle in Fig. 4, where ineffective operation instances whose i_1 indexes are beyond the L_1 iteration range, $[0, N_1 - 1]$, are masked from execution using predication [13], as shown in gray color in Fig. 4.

Due to the regular stalls and issuing, repeating patterns naturally appear in the final schedule. The final schedule is composed of a **prolog**, the repetition of an **outermost loop pattern (OLP)** and an ILES, and an **epilog**. Each of them consists of multiple copies of the kernel, as illustrated in Fig. 3(c).

3.2 Assumptions and Conventions

Throughout the paper, we will assume that a loop nest is composed of n loops: L_1, L_2, ..., L_n, as shown in Fig. 5(a), where $OPSET_x$ represents a set of non-branch operations between the beginnings of two adjacent loops. The loop nest is a **single loop** if $n = 1$.

Without loss of generality, we assume that SSP selects the outermost loop L_1 for scheduling, and constructs a 1-D schedule, which is represented by a kernel. The stages in the kernel are numbered from right to left as $0, 1, \cdots$, as shown in Fig. 5(b). Each stage takes T cycles to execute. If an operation is scheduled at cycle c of stage s, where cycle c is relative to the beginning of the stage and $0 \leq c < T$, its **1-D schedule time** is equal to $s * T + c$. The operations from the same loop L_x, including its inner loops, are scheduled into contiguous stages. The first such stage is referred to as f_x, and the last one l_x. The total number of stages for loop L_x is termed S_x, which equals $l_x - f_x + 1$.

The kernel is then used as a template to build the final schedule. The general form of the final schedule is shown in Fig. 5(c).

Unless stated otherwise, the term **iteration** always refers to an iteration of the outermost loop L_1. A **loop variant** refers to a temporary name that has a definition in loop L_1, including its inner loops. To be simple, we assume that at each loop level, the variant is either not defined or defined only once.

3.3 Features of the Vector Lifetimes

Fig. 6 shows the space-time diagram for each of the loop variants in our example, corresponding to the final schedule in Fig. 4, with segments A through H marked accordingly. They include all the typical features of a vector lifetime in any SSP final schedule.

Similarly to the single loop case, a vector lifetime may have a leading blade when there are live-in values, and a trailing blade when there are live-out values. For example, in Fig. 6, TN1 has a leading blade, and TN2 a trailing blade (Assume it has two live-out values). The wand, however, has unusual features specific to a multi-dimensional loop, due to the following two reasons:

First, an iteration has inner loops within it. This leads to the following features:

1. Multiple intervals in a scalar lifetime. A variant can be defined and killed multiple times during the execution of an inner loop. See TN2 and TN3 in Fig. 6 for example.

2. Unknown length of an interval at compile time. A variant has an interval **live through** an inner loop if the variant is live, but never redefined, during the execution of this loop. The length of the interval depends on the number of iterations of this loop and those of its inner loops, which may be unknown at compile-time.

Second, an iteration may be stalled and resumed. This leads to two other interesting features:

3. Stretched intervals. An interval may be "stretched" longer than usual in the period of an ILES. For example, Fig. 4 illustrates the first interval, produced by operation b and consumed by operation c, in each scalar lifetime of TN2 corresponding to iterations 0 to 5. All the intervals have the same length except the two intervals from iterations 3 and 4, which start before but end after the first ILES, segment C.

4. Delayed initiation of intervals. For example, in Fig. 4, the intervals from successive iterations initiate every $T = 2$ cycles, except that the interval from iteration 5 is delayed until after the first ILES.

The above 4 features present novel challenges to the register allocation. Fortunately, there is also one useful and important feature:

5. Repetition of the OLP and ILES segments, as indicated by the general form of a final schedule in Fig. 5(c). When an OLP (ILES) repeats, any interval in it repeats as well. This implies that the part of the vector lifetime within an OLP (or ILES) must be identical to that in any other OLP (or ILES). The only exception is that the first and last OLPs, and the last ILES may contain a subset of intervals of other OLPs and ILESes due to the ineffective operations at the beginning and the end of the final schedule (See Fig. 4).

3.4 Problem Formulation

The problem to be addressed in this paper can be formulated as follows: *Given a loop nest composed of n loops $L_1, L_2, \ldots, L_n$, sup-*

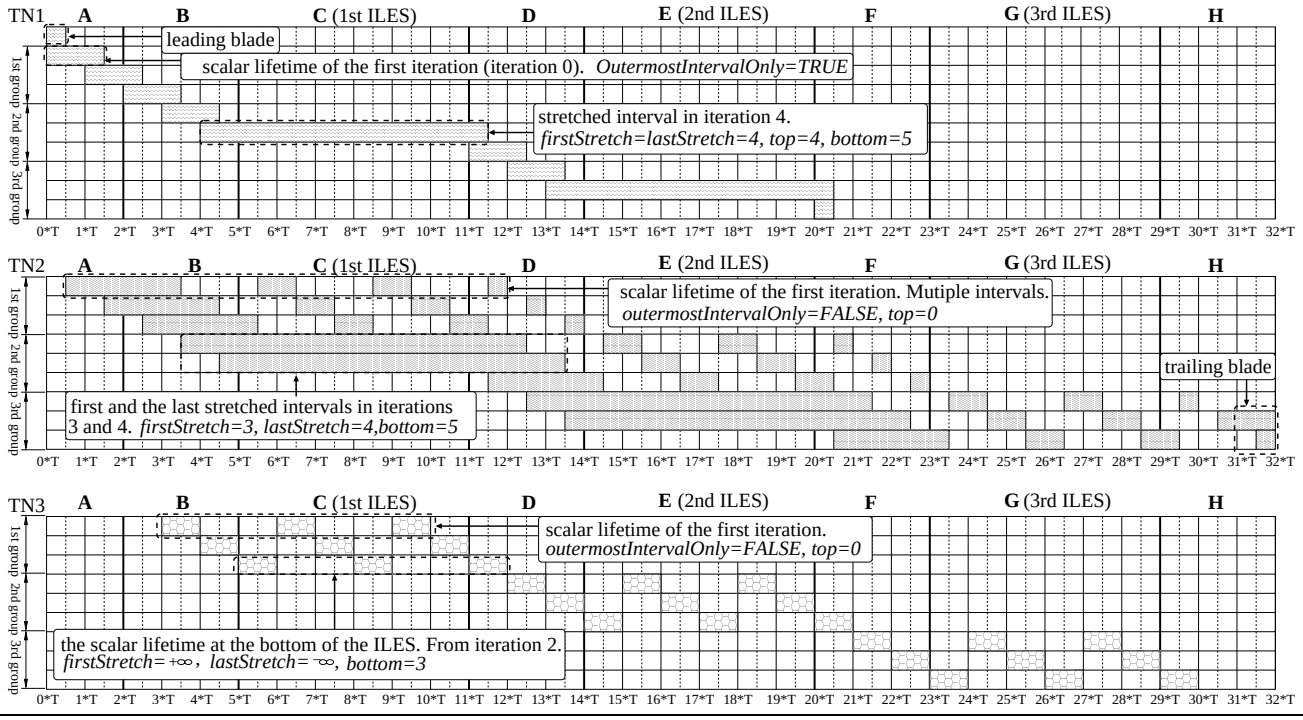


Figure 6. The Final Form of the Vector Lifetimes

pose the outermost loop L_1 is software-pipelined by SSP. Allocate minimum number of registers to the loop variants, in the presence of the rotating register file. Considering the cyclic structure of a rotating register file, the task is to pack the vector lifetimes on the surface of a cylinder, where time is the axis and the number of registers is the circumference, such that there is no conflict between any two scalar lifetimes and the circumference is minimized.

3.5 Key Concepts and Intuition

The key to the entire register allocation problem is to efficiently abstract the vector lifetimes, such that their complexity is captured, and their regularity is exposed to enable us to develop a simple solution. Based on the abstract representation of the vector lifetimes, the next key is to accurately measure how close and how far two vector lifetimes can be packed together on the space-time diagram.

We propose a dynamic view of a vector lifetime, which greatly simplifies both lifetime representation and the measurement of the distance between two vector lifetimes. We further propose a conservative and an aggressive ways for the measurement.

3.5.1 The Dynamic View of a Vector Lifetime: the Simplest, Ideal, and Final Forms

A vector lifetime can be viewed in a dynamic way, from its simplest form, to its ideal form, and eventually to its final form.

The **simplest form** of a vector lifetime corresponds to the special case that the number of iterations of any inner loop equals 1. The loop nest is then equivalent to a single loop. Therefore it is easy to represent and handle the vector lifetimes using the classical register allocation for software pipelined single loops presented in Section 2.2. For our example, the simplest form of the three vector lifetimes are shown in Fig. 7(a). The simplest form is composed of prolog, OLPs, and epilog, but without ILESes.

The **ideal form** of a vector lifetime corresponds to the ideal case that all the scalar lifetimes are evenly issued every T cycles. Each iteration runs without stopping as if there were no resource constraint. In this situation, there is no stall in the final schedule. Therefore there are no stretched intervals. The ideal form can be reached by adding to the simplest form the intervals produced during the execution of the inner loops. The ideal form of the vector lifetimes for our example is shown in Fig. 7(b).

The **final form** corresponds to the final schedule, which does have stalls to account for resource constraints. The final form of the vector lifetimes for our example is represented in Fig. 6. Every part of each vector lifetime is represented, without any omission.

With this dynamic view, it is much easier to mathematically abstract any vector lifetime. First, the simplest form is used to abstract the start and end of the scalar lifetimes. Then the ideal form is used to add the intervals defined at the inner loop levels. Finally, the final form is used for abstraction of the stretched intervals.

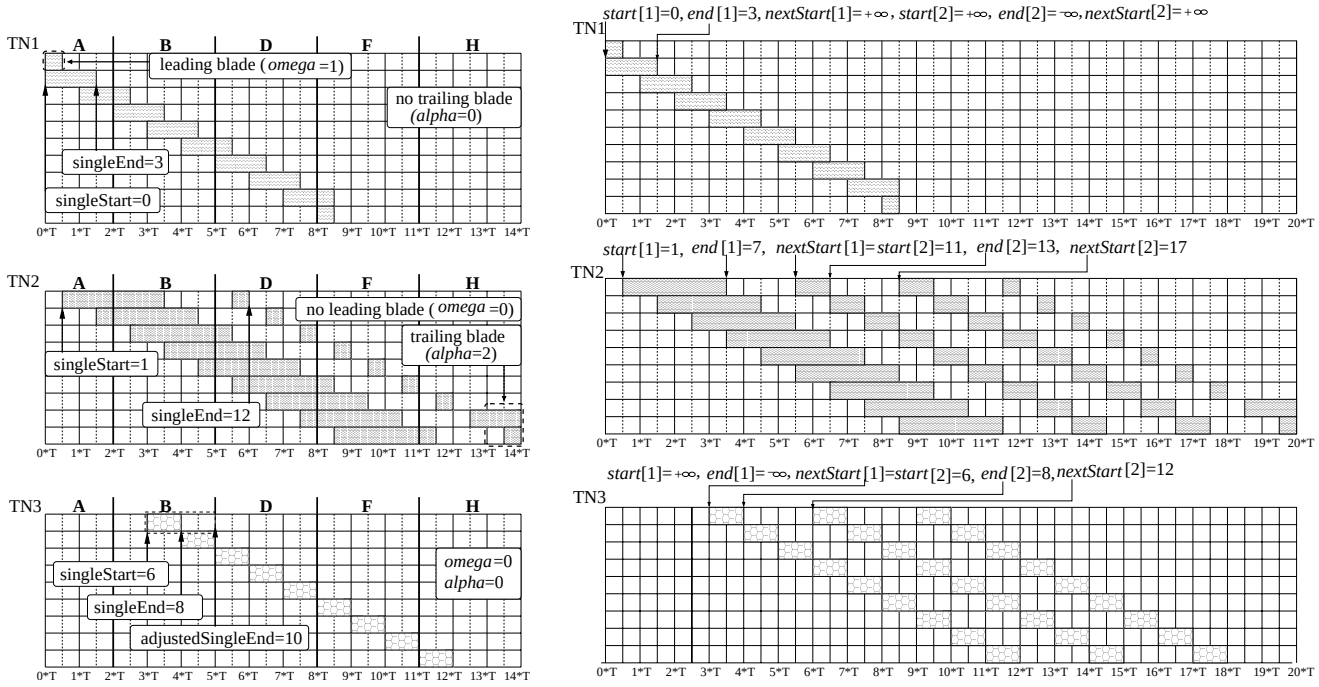
Repetition exists in all the three forms. (1) In the simplest and ideal forms, all scalar lifetimes are identical, except for the leading and trailing blades. (2) For a scalar lifetime in the ideal form, the interval defined at an inner loop level repeats itself with the sequential execution of this inner loop. (3) In the final form, the ILESes repeat themselves as well, as discussed in Section 3.3.

Due to the repetition, in representing and handling a vector lifetime, we can simply focus on the scalar lifetime corresponding to the first iteration, the first instance of the interval defined at each inner loop level, and the first ILES, and take them as a point of reference.

3.5.2 The Distances between Two Vector Lifetimes

There are two ways to pack two vector lifetimes on the space-time diagram: with and without interleaving. Fig. 8(a) shows a register allocation for our example without interleaving. The diagram has already been wrapped around into a cylinder. The circumference is 9. Fig. 8(b) shows another register allocation where vector lifetimes are interleaved when possible. In this case, the second scalar lifetime of TN3 is interleaved with the first scalar lifetime of TN2, and so on. The interleaving results in a smaller circumference of 7. One may observe that this is an optimal solution: the circumference cannot be further reduced.

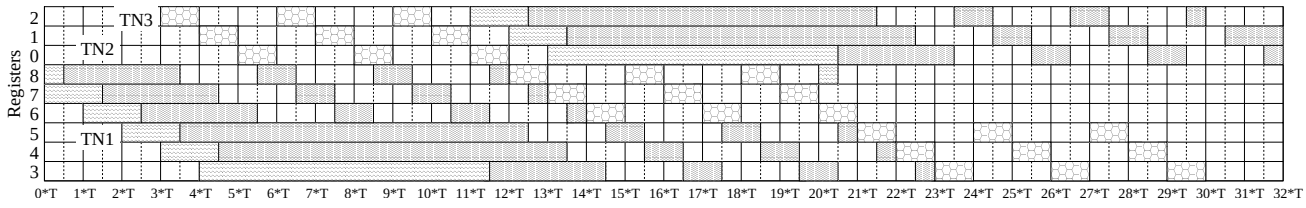
Correspondingly, we propose **conservative distance** and **aggressive distance**, which measure how close and how far, in number of registers, two vector lifetimes can be on the space-time diagram without conflict. The conservative distance does not allow for interleaving, whereas the aggressive distance does. The two dis-



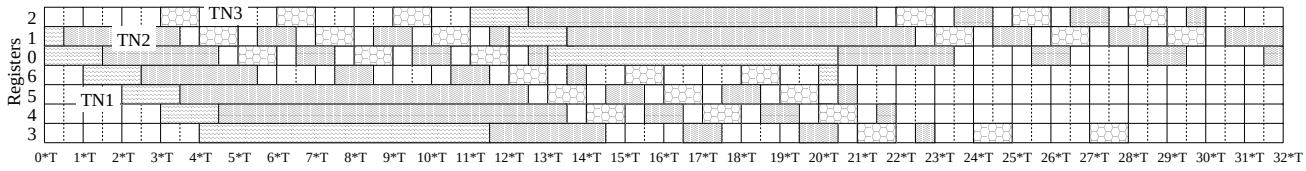
(a) The Simplest Form: Prolog+OLPs+Epilog. No ILES

(b) The Ideal Form: No Stall or Resource Constraints

Figure 7. The Simplest and Ideal Forms of the Vector Lifetimes



(a) Register Allocation Using the Conservative Distance. Circumference=**9**



(b) Register Allocation Using the Aggressive Distance. Circumference=**7**

Figure 8. Register Allocation Examples

tances play a key role in effectively minimizing register usage. The aggressive distance enforces finer control upon the selection of a register for a vector lifetime, as to be confirmed by our experiments. The three forms of the vector lifetimes will be used appropriately to compute the distances.

4. Solution

Our approach consists of the following steps: **(1) Lifetime normalization**. First, the vector lifetimes are normalized such that any interval has a length known at compile time. **(2) Lifetime representation**. The normalized vector lifetimes are then abstracted by a set of parameters. **(3) Distance Calculation**. Using those parameters, a conservative and an aggressive distances between any two

vector lifetimes, including the distances between a vector lifetime and itself, are computed. Then either the conservative or the aggressive distance can be used in the following steps: **(4) Sorting**. The distance is used to order the vector lifetimes. (5)**Bin-packing**. The ordered vector lifetimes are inserted one by one onto the surface of the space-time cylinder with one of the strategies (best, first, and end fits), assuming maximum circumference, with the distance between any pair of vector lifetimes respected. **(6) Circumference minimization**. The last step minimizes the circumference of the cylinder. The main algorithm is shown in Fig 9.

So far, we use a sorting process similar to that in the traditional register allocation for single loops [12, 5], as to be shown in Section 6. However, any other sorting heuristic may also be feasible. The other steps will be presented in detail below.

ALLOCATE_REGISTERS($strategy$):
1: $VLT \longleftarrow \emptyset$ *//an empty list for vector lifetimes*
2: **for each** loop variant tn **do**
3: $REFS \leftarrow$Normalize(tn) *//gather and normalize tn references*
4: $vlt \leftarrow$ Represent($REFS$) *//abstract lifetime from REFS*
5: $VLT \leftarrow VLT \cup \{vlt\}$ *//append the lifetime to the list*
6: **for each** vector lifetime $A \in VLT$ **do**
7: **for each** vector lifetime $B \in VLT$ **do**
8: $CONS[A, B] \leftarrow$Compute_Conservative_Distance(A,B)
9: $AGGR[A, B] \leftarrow$Compute_Aggressive_Distance(A,B)
10: $DIST \leftarrow CONS$ or $AGGR$
11: $VLT \leftarrow$Sort_Lifetimes($VLT, DIST$)
12: Insert_Lifetimes($VLT, DIST, strategy$) *//bin-packing*
13: Minimize_Circumference($VLT, DIST$)

Figure 9. Register Allocation Algorithm

4.1 Lifetime Normalization

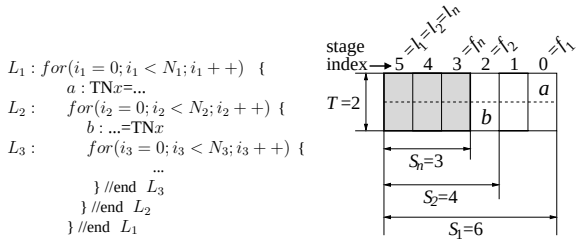
An obstacle to lifetime representation is that the length of an interval may be unknown at compile time, if it is live through a loop, as discussed in Section 3.3. Without this information, it is impossible to characterize the interval accurately.

In order to represent the vector lifetimes uniformly, we first normalize them such that after normalization, any interval has a length known at compile time. To achieve this purpose, any interval that is live through any loop needs to be cut in the middle so that it is not live through the loop any more. Due to the nesting structure of the loops, if the interval is live through any outer loop, it must be live through the innermost one. Therefore, preventing it from being live through the innermost loop is sufficient to prevent it from being live through any other loop. Conceptually, this can be done by inserting a dummy copy instruction TN=TN in the innermost loop, where TN is the variant corresponding to the vector lifetime under discussion.

Example: Fig. 10(a) shows another example loop nest, where TNx is defined at L_1 level, live in L_2 and L_3 without being redefined. A 1-D schedule is shown in Fig. 10(b). Then its first scalar lifetime of TNx in the ideal form would be like that shown in Fig. 10(c). The scalar lifetime has a single interval, which has indefinite length. The normalization is equivalent to inserting a dummy copy instruction in the innermost loop. Fig. 10(d) conceptually illustrates this effect. Fig. 10(e) shows the intervals to be observed by lifetime representation in the following step, due to the normalization. An interval is highlighted within a dotted box. The normalization cuts the long interval into repetitive smaller intervals, with the execution of the inner loops L_3 and L_2. We will further study the representation of the intervals later.

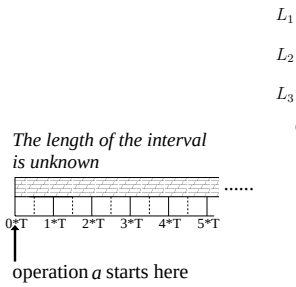
Fig. 11 shows the algorithm of lifetime normalization. All the references of the variant, definitions and uses, are gathered into a set, $REFS$. A **reference** is expressed as a 4-tuple ($time$, USE/DEF, $omega$, $level$), where $time$ is the 1-D schedule time of the operation that refers to the variant; USE/DEF indicates whether the reference is a definition or use; $omega$ is the live-in distance of the reference in the operation; and $level$ is the nesting level of the loop that immediately encloses the operation.

If the loop variant is live through the innermost loop L_n, a dummy use and a dummy definition with $time = fn * T$ are added into the references set. This is equivalent to inserting a copy operation TN=TN at the first cycle of the innermost loop stages in the 1-D schedule, although we never really do such insertion. Inserting the copy instruction at any other cycle of the innermost loop stages is also feasible. Here we choose the first cycle arbitrarily.

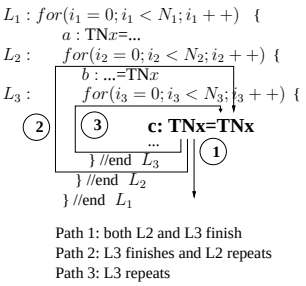


(a) The 2nd Example

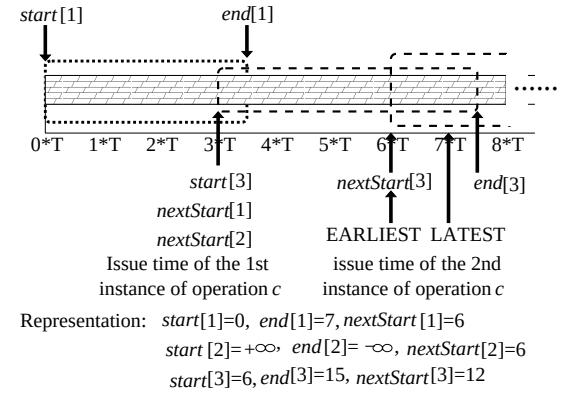
(b) The Kernel



(c) The First Scalar Lifetime of TNx in the Ideal Form



(d) Conceptual Illustration of the Effect of Normalization



(e) Representation after Normalization

Figure 10. Normalize a Vector Lifetime to Simplify the Representation

4.2 Lifetime Representation

A vector lifetime can be fully abstracted by characterizing its simplest and ideal forms only, with some **core parameters**. Then the stretched intervals in the final form can be represented by some **derived parameters**, deduced from the core parameters.

NORMALIZE(tn):
1: $REFS \longleftarrow \emptyset$ *//an empty set for tn references*
2: **for each** operation op **do**
3: $time \longleftarrow$ 1-D schedule time of op
4: $level \longleftarrow$ the level of the loop immediately enclosing op
5: **for each** source operand of op equal to tn **do**
6: $omega \longleftarrow$ live-in distance of the operand
7: $ref \longleftarrow (time, \text{USE}, omega, level)$
8: add ref into $REFS$
9: **for each** result operand of op equal to tn **do**
10: $ref \longleftarrow (time, \text{DEF}, 0, level)$
11: add ref into $REFS$
12: **if** tn is live in L_n but not defined in it **then**
13: $ref1 \longleftarrow (f_n * T, \text{USE}, 0, n)$
14: $ref2 \longleftarrow (f_n * T, \text{DEF}, 0, n)$
15: add $ref1$ and $ref2$ into $REFS$
16: return $REFS$

Figure 11. Lifetime Normalization Algorithm

4.2.1 Core Parameters: Characterizing the Simplest and Ideal Forms of a Vector Lifetime

First, let us characterize the simplest form, in the same way as the traditional register allocation for software pipelined single loops does [12]. **SingleStart** and **singleEnd** are the start and end time of the scalar lifetime corresponding to the first iteration. **Omega** and **alpha** are the total live-in values and total live-out values, respectively.

Next, we characterize the ideal form. Two elements in a scalar lifetime need to be considered: "interval", and "hole" between two adjacent intervals. For the interval defined at a loop level and the hole following it, we represent them by the first instance of the interval and the first instance of the hole in the scalar lifetime corresponding to the first iteration. *The representation maximizes the length of the interval, but minimizes that of the hole*. This is necessarily for two reasons: (1) The definition of the interval may have last uses, and successor definitions, on different control flow paths, due to the multiple loop-back edges of the loop nest. Therefore, the lengths of the interval and the hole may vary. (2) This is required by distance calculation. For example, in calculating the aggressive distance of two vector lifetimes A and B, an interval of B is put into a hole of A. If, in the worst case that the interval is the longest possible and the hole is the shortest possible, the hole is still long enough to contain the interval without conflict, then any instances of the interval and the hole would fit without conflict, whatever paths are followed by the interval and the hole. Similar request is from calculating the conservative distance as well.

The intervals defined at every loop level, and the holes following them, are represented by 3 arrays, $start$, end, and $nextStart$. For each level i, we ensure that $end[i] - start[i]$ is the maximal length of the interval defined at this level, and $nextStart[i] - end[i]$ is the smallest possible size of the hole following it.

Start[i] is the issue time of the definition of the variant at level i, which is equal to its 1-D schedule time. **End[i]** is one plus the *latest* issue time of all possible uses of the definition. There are two special cases: (1) When the definition has no use, $end[i]$ is the completion time of the definition, i.e., one plus the issue time of the definition. (2) When there is no definition of the variant at level i at all, $(start[i], end[i])$ are set as $(+\infty, -\infty)$, indicating that no interval is really defined at this level since the length of the interval is $end[i] - start[i] < 0$. For this reason, the interval will be referred to as a **phantom interval**. This setting of $(start[i], end[i])$ naturally enables us to treat the phantom interval the same way as a "real" interval, without any special consideration, later in calculating the distances.

NextStart[i] is the *earliest* issue time of all possible definitions that are following this interval in the same iteration. $NextStart[i]$ is naturally defined to be $+\infty$ if there is no such definition.

In short, a vector lifetime can be fully represented as a 7-tuple: **(singleStart, singleEnd, omega, alpha, start, end, nextStart)**. The tuple is defined based on the references set $REFS$, which is the output of the lifetime normalization. Formal definitions are given in the Appendix.

Example: Fig. 7(a) and 7(b) show how the vector lifetimes for our example in Fig. 6 are characterized in the simplest and ideal forms.

Fig. 10(e) shows another example how the ideal form is described. The variant has no definition at L_2 level. So *(start[2], end[2])*= $(+\infty, -\infty)$. This phantom interval has a next interval defined at L_3 level by operation c (See Fig. 10(d)). Thus $nextStart[2] = start[3]$.

For that next interval, its start, $start[3]$, is equal to the issue time of the first instance of operation c. The interval may terminate along three different paths, as shown in Fig. 10(d). Among the three paths, the second one leads to the longest interval, and the third one the smallest hole. Both paths reach the second instance of operation c. The instance has the latest issue time if it is along the second path, and the earliest issue time along the third one, which determine $end[3]$ and $nextStart[3]$, respectively. Note that the hole following the interval has a negative size, i.e., $nextStart[3] - end[3] < 0$. However, it can be handled in the same way as a normal hole with positive size, without any special concern, later in calculating the distances.

4.2.2 Derived Parameters: Characterizing the Final Form of a Vector Lifetime

The derived parameters describe the stretched intervals in the final form of a vector lifetime.

The length of a stretched interval cannot be accurately characterized, even after lifetime normalization. Unlike the indefinite-length intervals due to the liveness of the variant, which can be normalized to have finite-length by "breaking" the liveness in the middle of the intervals, stretched intervals are caused by stalls in the final schedule due to limited processor resources, which is unavoidable. For example, the vector lifetimes in Fig. 7(b) are normal, however, its final form in Fig. 6 still has stretched intervals, whose lengths remain indefinite at compile time.

Thus, we cannot directly describe a stretched interval "horizontally", i.e., from the view point of "time" in its space-time diagram, as we did for the scalar lifetimes in the simplest and ideal forms. Fortunately, it is feasible and sufficient for register allocation to characterize the stretched intervals "vertically", i.e., from the viewpoint of "space" in the space-time diagram. This leads to the following parameters derived from the core parameters: **(outermostIntervalOnly, firstStretch, lastStretch, top, bottom)**.

The boolean variable **outermostIntervalOnly** is true when the loop variant is defined only at the outermost level.

FirstStretch and **lastStretch** are the iteration indexes of the first and last stretched intervals of the variant, respectively, that appears in the first ILES. If there is no stretched interval at all, we set $(firstStretch, lastStretch) = (+\infty, -\infty)$.

Top is the iteration index of the intervals at the top of the first ILES. **Bottom** is one plus the iteration index of the intervals at the bottom of the first ILES. When there is no interval in an ILES, we set $(top, bottom) = (+\infty, -\infty)$. The difference $bottom - top$ represents the vertical thickness of the vector lifetime in an ILES.

Example: Fig. 6 shows the derived parameters of the lifetimes.

The derived parameters are formally defined in the Appendix.

4.3 Distance Calculation

For any two vector lifetimes A and B, a metric, called **distance**, is used to measure how close and how far A and B can be packed together on the space-time diagram without conflict. There are two types of distances: conservative and aggressive distances.

Let r_A and r_B be the registers to be allocated to A and B, respectively. Each of the **conservative distance**, denoted *CONS[A, B]*, and the **aggressive distance**, denoted $AGGR[A, B]$, defines a legal range of $r_B - r_A$, within which A and B do not conflict in the space-time diagram. The conservative distance does not allow the vector lifetimes to interleave, while the aggressive distance does. For convenience, let $INTL[A, B]$ be the legal range of $r_B - r_A$ when the vector lifetimes interleave, called **interleaving distance**.

Then

$$AGGR[A,B] = CONS[A,B] \cup INTL[A,B]$$

More specifically, for the conservative distance, B is to the right of any intervals of A. For the interleaving distance, an interval of B is to the right of the corresponding interval of A defined at the same loop level. For convenience, in either case, we will say that B **is to the right of** A, or B **is after** A. Equivalently, we may also say A **is to the left of** B, or A **is before** B.

Note that the relative positions of A and B are different in $AGGR[A,B]$ and $AGGR[B,A]$. Thus in general, they are not equal. That is, $AGGR$ is not symmetric. Similarly, $INTL$ and $CONS$ are not symmetric, either.

4.3.1 Conservative Distance

$$adjusted\text{-}SingleEnd(A) = \begin{cases} singleEnd(A) \text{ , if } outermost\text{-}IntervalOnly(A) \\ max(singleEnd(A), l_n * T) \text{ , otherwise} \end{cases}$$

$$d_1 = \left\lceil \frac{adjustedSingleEnd(A) - singleStart(B)}{T} \right\rceil \quad (1)$$

$$d_2 = \begin{cases} d_1 & \text{if } omega(B) = 0 \\ max(d_1, omega(A)) & \text{otherwise.} \end{cases} \quad (2)$$

$$d_3 = \begin{cases} d_2 & \text{if } alpha(A) = 0 \\ max(d_2, alpha(A)) & \text{otherwise.} \end{cases} \quad (3)$$

$$d_4 = max(d_3, bottom(B) - top(A)) \quad (4)$$

$$\Longrightarrow CONS[A,B] = [d_4, +\infty]$$

Figure 12. Conservative Distance Computation

First, $singleEnd(A)$ is adjusted, which will be explained later.

To compute the conservative distance between two vector lifetimes A and B, we consider the simplest form of the vector lifetimes. The problem is then equivalent to that of the single loop case [12] and the following conditions must be verified: the wand of B must be after the wand of A, and the leading/trailing blades of B must be above the leading/trailing blades of A. Formally,

$singleStart(B) + (r_B - r_A) * T \geq adjustedSingleEnd(A)$

$r_B - r_A \geq omega(A)$ if $omega(B) > 0$

$r_B - r_A \geq alpha(A)$ if $alpha(A) > 0$

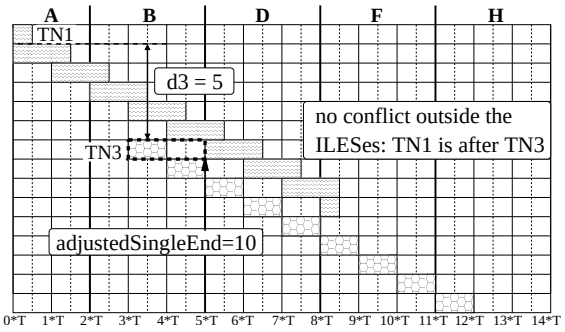
Solving these inequalities results in formulas 1, 2, 3 in Fig. 12.

Next, we further consider the vector lifetimes in their final form by putting ILESes into their simplest form. Because interleaving is not allowed, if A and B have intervals in those segments, a segment of B should be above the corresponding segment of A, i.e., $r_B - bottom(B) \geq r_A - top(A)$. This leads us to formula 4.

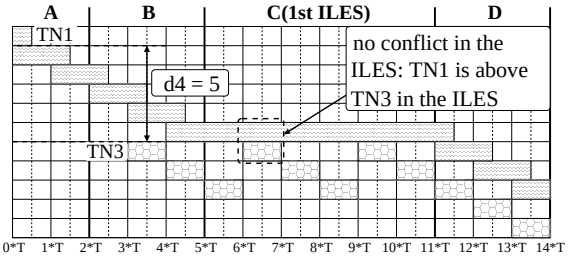
Example: Fig. 13 illustrates the computation of *CONS*$[TN3, TN1]$ for the example in Fig. 6 and 7. D_3 and d_4 ensure that there is no conflict outside and inside the ILESes, respectively. In this example, $d_3 = d_4 = 5$, and thus $CONS[TN3, TN1] = [5, +\infty]$.

Why $singleEnd(A)$ ***needs adjustment?*** The simplest form of a vector lifetime is composed of the prolog, OLPs, and epilog. In these segments, there should not be any conflict before and after expanding the simplest form into the final form. Unfortunately, since ILESes are not included in the simplest form, there is one and only one exception: when an interval's definition is before, but its consumer is in, an ILES, since the ILES is not in the simplest form and neither the consumer, the interval would be shorter than usual. This may leads to a value of d_1 smaller than necessary.

Fig. 14(a) shows a simple loop nest. With the 1-D schedule in Fig. 14(b), the vector lifetimes are shown in Fig. 14(c). From the figure, the lower bound of $CONS[TNz, TNy]$ should be 3. However, in the simplest form (Fig. 14(d)), the first instance of operation c, $c_{0,0}$, produces a value without the consumer. The consumer is in the first ILES, which is not in the simplest form. Thus the interval lasts for only one cycle. This leads to $d_1 = 2$.



(a) In the simplest form of the vector lifetimes



(b) In the final form of the vector lifetimes

Figure 13. Illustration of Conservative Distance Computation

Then in the final form, since TNy has no interval in the ILESes and thus $bottom(TNy) = -\infty$, we have $d_4 = d_1 = 2$, which is obviously wrong: There is conflict in the first OLP, because the interval starting from $c_{0,0}$ now lasts into the first ILES. Due to the repetitiveness of OLP and ILES, there would be conflict in other OLPs as well. To avoid such conflict, in the simplest form, we only need to elongate that interval to reach the end of the first OLP so as to take that piece of space exclusively.

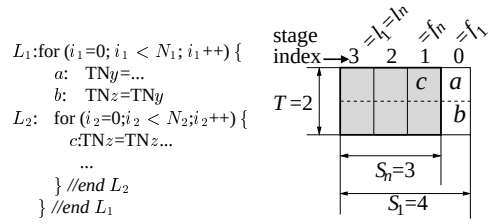
Having a consumer in an ILES means that there is a use at an inner loop level. Since after normalization, a loop cannot have only uses in it without a definition, that further means there is a definition at an inner loop level. That is, $outermostIntervalOnly$ is false. In general, in calculating $CONS[A,B]$, we adjust $singleEnd(A)$ to the end of the first OLP, if it is before that end and *outermostIntervalOnly(A)* is false. This will sufficiently prevent any future conflict outside the ILESes. In the above example, $singleEnd$ of TNz needs to be adjusted to the end of the first OLP (cycle $3 * T$, or in general, $l_n * T$). That leads to $d_1 = 3$. In fact, Fig. 14(c) is the result using the adjusted $singleEnd$.

Example: In Fig. 7(a), $singleEnd$ of TN3 is adjusted to cycle 10. It remains the same for the other two variants.

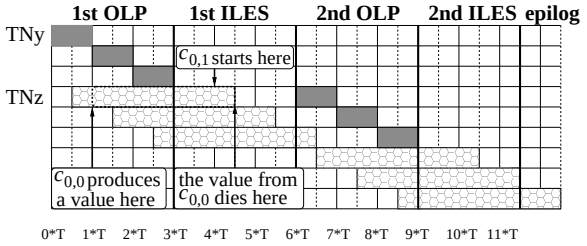
4.3.2 Aggressive Distance

The aggressive distance is the union of the conservative and interleaving distances. Below we introduce how to compute the interleaving distance.

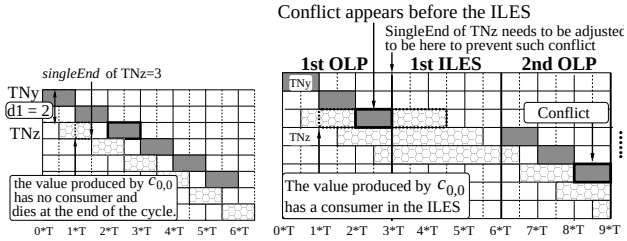
First, we consider the ideal form of the vector lifetimes. For each loop level i, let I_A and I_B be the intervals from vector lifetimes A and B defined at this level, respectively. To interleave them, I_B must be put to the hole behind I_A. There is one special case to be considered: if I_B is a phantom, the interval following it in the same



(a) The 3rd Example (b) The Kernel



(c) The Final Form of the Vector Lifetimes (with $N_1 = 6, N_2 = 2$)



(d) Pack the Simplest Form (e) Expand to the Final Form

Figure 14. Adjusting SingleEnd

scalar lifetime of B should be naturally after I_A. That is,

$$\begin{aligned} start(I_B) + (r_B - r_A) * T &\geq end(I_A) \\ end(I_B) + (r_B - r_A) * T &\leq nextStart(I_A) \\ nextStart(I_B) + (r_B - r_A) * T &\geq end(I_A) \text{ if } I_B \text{ is a phantom} \end{aligned}$$

where $start(I_v)$, $end(I_v)$, and $nextStart(I_v)$ refer to $start[i]$, $end[i]$, and $nextStart[i]$ of the vector lifetime v, with $i \in [1, n]$ being the loop level under discussion.

This leads to the $[l, u]$ range defined by equations 5 and 6 in Fig. 15. The intersection of all the ranges calculated from the (I_A, I_B) pairs of all loop levels is the range within which no conflict would occur between any interval of A and B. This intersection is defined by equations 7 and 8 for the lower and upper bounds, respectively.

Then we consider the stretched intervals in the final form. Intuitively, the stretched intervals of B should be between the stretched intervals of A and the intervals of A defined at the inner loop levels. More formally,

$$\begin{aligned} r_B - bottom(B) &\geq r_A - firstStretch(A) \\ r_B - firstStretch(B) &\leq r_A - S_n \text{ if not } outermost - \\ &\qquad IntervalOnly(A) \end{aligned}$$

These translate into formulas 9 and 10.

$$l(I_A, I_B) = \begin{cases} \left\lceil \frac{end(I_A) - nextStart(I_B)}{T} \right\rceil & \text{if } I_B \text{ is} \\ & \text{a phantom} \\ \left\lceil \frac{end(I_A) - start(I_B)}{T} \right\rceil & \text{otherwise} \end{cases} \quad (5)$$

$$u(I_A, I_B) = \left\lfloor \frac{nextStart(I_A) - end(I_B)}{T} \right\rfloor \quad (6)$$

$$d_5 = max \left\{ l(I_A, I_B) \mid \begin{array}{l} \forall \text{interval pair } (I_A, I_B) \\ \text{from } A \text{ and } B \end{array} \right\} \quad (7)$$

$$d_6 = min \left\{ u(I_A, I_B) \mid \begin{array}{l} \forall \text{interval pair } (I_A, I_B) \\ \text{from } A \text{ and } B \end{array} \right\} \quad (8)$$

$$d_7 = max\,(d_5, bottom(B) - firstStretch(A)) \quad (9)$$

$$d_8 = \begin{cases} d_6 \text{ if } outermostIntervalOnly(A) \\ \min(d_6, firstStretch(B) - S_n) \text{ otherwise} \end{cases} \quad (10)$$

$$d_9 = \begin{cases} d_7 & \text{if } omega(B) = 0 \\ max(d_7, omega(A)) & \text{otherwise.} \end{cases} \quad (11)$$

$$d_{10} = \begin{cases} d_9 & \text{if } alpha(A) = 0 \\ max(d_9, alpha(A)) & \text{otherwise.} \end{cases} \quad (12)$$

$$\Longrightarrow INTL[A, B] = [d_{10}, d_8]$$
$$AGGR[A, B] = CONS[A, B] \cup INTL[A, B]$$

Figure 15. Aggressive Distance Computation

Finally, similarly to formulas 2 and 3, the constraints from the live-in and live-out values are added through formulas 11 and 12.

Example: Fig. 16 illustrates the major steps in computing the interleaving distance $INTL[TN3, TN2]$. For clarity, we use only the final form, and add the phantom intervals of TN3 to show how the extreme values, $+\infty$ and $-\infty$, are used in the formulas naturally without any special concern.

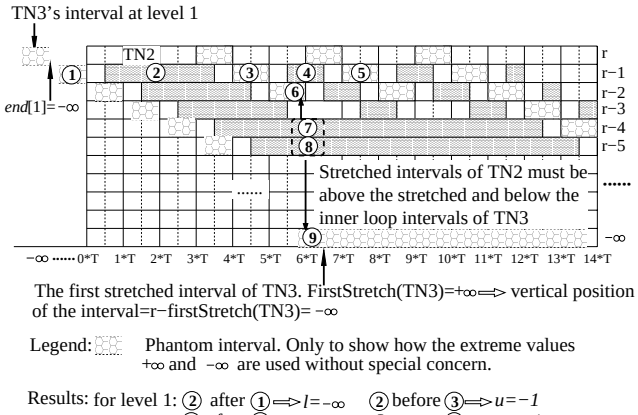


Figure 16. Illustration of Aggressive Distance Computation

4.4 Bin-packing on the Cylinder

Once the distance between any two vector lifetimes is computed, the vector lifetimes are inserted one after the other on the surface of a space-time cylinder of a circumference equal to the maximum number of available registers, R. The algorithm is shown in Fig. 17.

INSERT_LIFETIMES($VLT, DIST, strategy$):
1: **for each** vector lifetime $A \in VLT$ **do**
2: $illegal[A] \longleftarrow \emptyset$ *//empty set of illegal registers*
3: **for each** unallocated vector lifetime $A \in VLT$ **do**
4: $legal \leftarrow [0, R-1] - illegal[A]$
5: Allocate to A a register $r_A \in legal$ using $strategy$
6: **for each** other unallocated vector lifetime $B \in VLT$ **do**
7: $illegal[B] \longleftarrow illegal[B] \cup newIllegal(B, A, R)$

where

$$newIllegal(B, A, x) = \{i \bmod x | \forall i \in \mathbb{N}, i \notin legalSet(B, A)\}$$

$$legalSet(B, A) = (r_A - DIST[B, A]) \cup (r_A + DIST[A, B])$$

Figure 17. Lifetime Insertion Algorithm

For any given vector lifetime A, $illegal[A]$ is the set of registers that cannot be allocated to A to avoid conflict with already-placed vector lifetimes. Its difference with the full set of available registers $[0, R-1]$, i.e., $([0, R-1] - illegal[A])$, is the set of candidate registers allocatable to A.

Assume that A is allocated register r_A. For any other unallocated vector lifetime B, its illegal set needs to be updated. B can be placed either to the right of A, or to the left of A. If B is to be placed to the right of A, $r_A + DIST[A, B]$ is the set of registers that can be allocated to B without conflict with A. If B is placed to the left of A, $r_A - DIST[B, A]$ is. The union of the sets in both cases is noted $legalSet(B, A)$. Here $DIST$ is the distance, $CONS$ or $AGGR$, and $x \pm set$ denotes a new set that is $\{x \pm y | \forall y \in set\}$. The complement of $legalSet(B, A)$ is the set of illegal registers for B, due to conflicts with A.

Fig. 18 illustrates $legalSet(B, A)$ with thin arrows when $DIST$ is $CONS$, and bold arrows when $DIST$ is $AGGR$. Note that the former is a subset of the latter.

Because the number of physical registers is limited to R, the set of illegal registers wraps around the cylinder and is therefore considered modulo R into $newIllegal(B, A, R)$. A legal register before wrapping around can become illegal after that, if any illegal register, after wrapping around, overlaps with it, as shown by the annotation for parts 1 and 2 in Fig. 18.

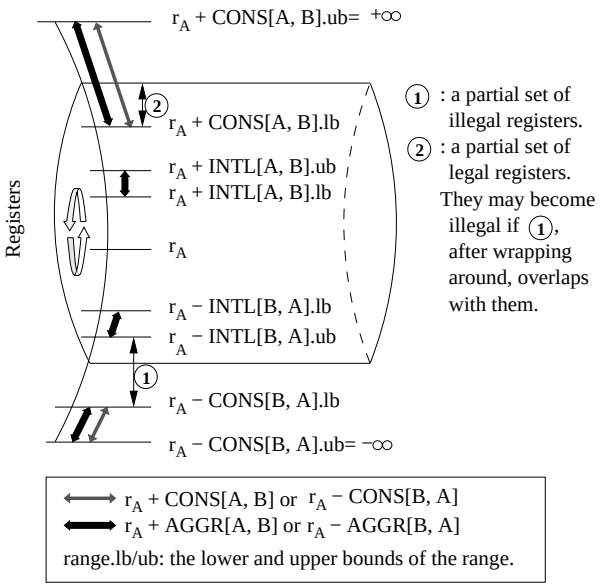


Figure 18. Lifetime Insertion on the Space-Time Cylinder

4.5 Circumference Minimization

So far the solution uses the maximum number of physical registers, R, as the circumference of the cylinder. The last step tries to compress the cylinder to decrease the actual number of registers required.

First, the circumference is initialized as the distance between the smallest and the biggest registers allocated. Then the allocation is translated such that the smallest register allocated equals 0.

A vector lifetime A may conflict with itself after wrapping around, if the circumference is not within $DIST[A, A]$, where $DIST$ is either $CONS$ or $AGGR$. It also conflicts with a different vector lifetime B, if B is allocated an illegal register with respect to A under the current circumference. If there is no conflict between any pair of vector lifetimes, including a vector lifetime with itself, the search is over. Otherwise, we increment the circumference until a legal solution is found. Note that a circumference of R is always valid, and thus the search will definitely terminate. The algorithm is shown in Fig. 19.

MINIMIZE_CIRCUMFERENCE($VLT, DIST$):
1: $minReg \longleftarrow$ the smallest register allocated
2: $maxReg \longleftarrow$ the biggest register allocated
3: $c \longleftarrow maxReg - minReg + 1$ *//c: circumference*
4: **for each** vector lifetime $A \in VLT$ **do**
5: $r_A \longleftarrow r_A - minReg$ *//r_A: A's register*
6: Next:
7: **for each** vector lifetime $A \in VLT$ **do**
8: **if** $c \notin DIST[A, A]$ **then**
9: $c \longleftarrow c + 1$
10: goto Next
11: **for each** vector lifetime $B \in VLT$, s.t. $B \neq A$ **do**
12: **if** $r_B \in newIllegal(B, A, c)$ **then** *//r_B:B's register*
13: $c \longleftarrow c + 1$
14: goto Next

Figure 19. Circumference Minimization Algorithm

5. Properties and Time Complexity

THEOREM 1. *Given two vector lifetimes A and B, if the loop nest is a single loop ($n = 1$), then $CONS[A, B] = AGGR[A, B] = [x, +\infty)$, where x is the lower bound calculated by the classical register allocation for software pipelined single loops [12]. In this sense, the approach in this paper subsumes the classical approach as a special case.*

Intuitively, a vector lifetime in a single loop has only one interval for each scalar lifetime. Thus putting the vector lifetime B to the right of A is equivalent to interleaving B to the right of A. So $CONS[A, B] = INTL[A, B] = AGGR[A, B]$.

PROOF. According to the definitions in Section 4.2, for any vector lifetime in a single loop, we have $outermostIntervalOnly$ is true, $singleStart = start[1]$ and is equivalent to $start$ in the traditional case, $singleEnd = end[1]$ and is equivalent to end in the traditional case, $nextStart[1] = +\infty$. Since there is no ILES, $(firstStretch, lastStretch) = (+\infty, -\infty)$, and $(top, bottom) = (+\infty, -\infty)$. Apply them to the formulas in Fig. 12 and 15. We obtain the same formulas used in Rau's method. □

Next we analyze the time complexity of our solution. Let l be the total vector lifetimes, and o the number of operations. Lifetime normalization and representation has a time complexity of $O(o)$ per variant, and $O(l * o)$ for all the variants. The total time spent in computing distances is $O(l^2)$ for all pairs of variants. Sorting can be done in $O(l^2)$ time. Bin-packing uses the same strategies as those for the single loops [12], given the distances. The time complexity for first and end fits are quadratic($O(l^2)$), and cubic for

best fit($O(l^3)$) [12]. Circumference minimization loops at most R times, with each time costing $O(l^2)$ time. In summary, the overall time complexity is $O(l^x + l * o)$, where $x = 2$ for first and end fit strategies, and $x = 3$ for best fit.

6. Experiments

The register allocation method proposed in this paper, as well as the rest of the SSP framework, was implemented in the ORC 2.1 compiler. The input of the register allocator is the kernel produced by the scheduling phase [14], and the output is a register-allocated kernel that is sent to the code generator [13].

6.1 Experimental Framework

The vector lifetimes are sorted in increasing order by **adjacency**. Ties are broken using $singleStart$, then $adjustedSingleEnd$ modulo T. The adjacency represents the number of cycles a register is idle. It is defined as $singleStart(B) - adjustedSingleEnd(A) + d_3(A, B) * T$, where A is the vector lifetime previously selected at the last step, B the vector lifetime to be selected, and $d_3(A, B)$ is the d_3 value in calculating $CONS[A, B]$ in Fig. 12. The heuristic is an extension to the (adjacency, start time) heuristic used in the traditional register allocation for single loops [12].

Seven different strategies were used for inserting the vector lifetimes on the space-time cylinder. The first, referred to as **Simple**, ignores the specific shape of a vector lifetime A and exclusively allocates to it $S_1 + omega(A)$ physical registers, which is the maximum number of instances of a loop variant that can be live simultaneously in the final schedule [13]. As described in Section 4, bin-packing and circumference minimization can use either the conservative or the aggressive distance. The two choices, combined with the first, best, and end fits introduced in Section 2.2, lead to 6 strategies: **ConsFirst**, **ConsBest**, **ConsEnd**, **AggrFirst**, **AggrBest**, and **AggrEnd**. Besides, a theoretical lower bound, named **MaxLive**, is used for measuring optimality. It is the maximum number of scalar lifetimes live simultaneously at any cycle in the final schedule.

A total of 134 loop nests were gathered from NAS, Livermore and SPEC2000 benchmark suites. To test the register pressure related with different levels, SSP was applied to each feasible level, leading to a total of 348 loop levels tested. Note that even for the same variant in the same source code, software pipelining of two different levels results in completely different vector lifetimes. The distribution of the nesting depths is shown in Table 1. Here "nesting depth" is 1 plus the total number of inner loops of the loop level under consideration. A maximum circumference of 1024 was assumed. When the number of registers allocated did not exceed 96 rotating integer and floating-point registers, the total rotating registers in Itanium architecture, the parallelized loop nest was run on an Itanium2 machine with 1.4GHz/1GB RAM, and correctness was validated by comparing its output with that of the same loop nest compiled with GCC or the original ORC2.1 binary.

Some statistics about the loops are shown in Fig. 20(a). Overall, 60% of the loops have 47 operations or less , but 12% of the loops have more than 200. 64% of the loops have an II of less that 10 cycles, and 8.4% of them have an II larger than 40 cycles. Note that, because a smaller II may be related to a higher number of stages, it tends to increase the register pressure. 80% of the loops have 56 integer loop variants or less and the maximum is 174. For floating-point loop variants, 80% of the loops has less than 45 with a maximum of 96. The total number of stages never exceeds 11 and the number of live-in values never goes above 7.

Nesting Depth	1	2	3	4	5
Number	127	108	68	33	12

Table 1. Number of Loops for Each Nesting Depth

6.2 Results

1. The conservative and aggressive distances, used with the best or first fit strategies, greatly improve the success rate of the register allocation, close to the limit set by an ideal register allocator. By *success*, we refer to the fact that the total number of registers allocated is within 96. A register allocator is *ideal* if it allocates exactly $maxLive$ number of registers. Fig. 20(b) and 20(c) show the distribution curves for loop nests of depths 2 to 5. For integer registers, of all the loops that can be successfully handled by an ideal register allocator, the simple heuristics can successfully handle only 30%. The rate is improved up to 53% using the conservative distance, and up to 91.7% using the aggressive distance. This confirms the *importance of characterization and exploitation of the specific patterns of the software pipelined vector lifetimes to achieve efficient register allocation*. Considering the fact that $maxLive$ is not the optimal lower bound [1], this result is encouraging and impressive. Overall, with only 96 integer registers, our method allows us to successfully compile 76.5% of all the loop nests.

We analyzed several loops to further investigate why the aggressive distance can be so effective. Fig. 21 shows an excerpt of the integer space-time cylinder of a double loop nest from the initialization procedure of the Livermore benchmark, automatically generated by our compiler (except for the annotations). A symbol in the figure represents a vector lifetime. Surprisingly, intervals from four different vector lifetimes, h, i, g, and 9, are interleaved. The allocation is impressively compact. Further investigation of some other loops shows that this phenomenon is not uncommon.

2. The first fit strategy is found to be almost as effective as the best fit strategy, with significantly less compile time. Their cumulative distribution curves on the number of registers allocated in Fig. 20(b) and 20(c) are close and cannot be distinguished. They both outperform the end fit strategy by succeeding in compiling at least 15% more of the loops. However, first fit takes 0.009 seconds on average, while best fit 11 seconds. Table 2 shows the average time distribution in each phase of register allocation. AggrBest consumes significantly more time in bin-packing than AggrFirst. There is no meaningful difference in the time of any other phase. Fig. 20(d) shows the average registers needed for each loop level for $AggrFirst$. For a single loop, the register requirement is almost minimized. For a multi-dimensional loop, as the nest deepens, the register requirement is relatively closer to $maxLive$.

	Normalization	Representation	Computing Distance	Sorting	Bin-packing	Circumference Minimization
AggrFirst	0.004	0.004	0.019	0.006	0.009	0.006
AggrBest	0.004	0.004	0.028	0.008	11.19	0.006

Table 2. Average Compile Time Distribution (in Seconds)

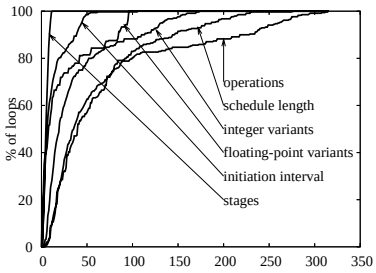
3. The experiments confirm that, in the single loop case, our method subsumes the traditional register allocation as a special case, and achieves near-optimality. Fig. 20(e) and Fig. 20(f) show that there is no difference between using the conservative distance or the aggressive distance, as proved in Section 5.

4. The improvement of register allocation does make a difference in performance. 18 loop nests, depth 2 to 5, were considered. For all of them, $AggrFirst$ succeeds, while $Simple$ and $ConsFirst$ fails. The loop nests were rescheduled with higher IIs so that $Simple$ and $ConsFirst$ may succeed. However, $simple$ cannot succeed for 11 loops even II is increased to infinitely big, and $ConsFirst$ for 5 loops. This is because the register pressure stops decreasing when II is above certain threshold. Similar situation is also noticed in software pipelining of single loops [10]. For all the other loops, we divided the final II (execution time) of each loop against that of $AggrFirst$, and get the II ratio (execution time ratio). On average, the IIs of $Simple$ and $ConsFirst$ were then 81% and 25% larger, respectively. And the execution time was 69% and 28% longer, respectively.

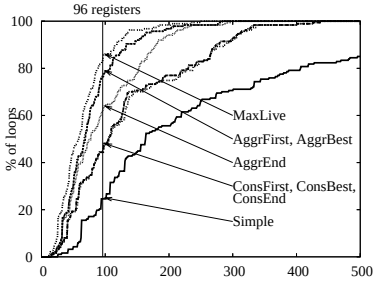
7. Conclusion

In the context of multi-dimensional software-pipelining, vector lifetimes with complex shapes present unusual challenges to regis-

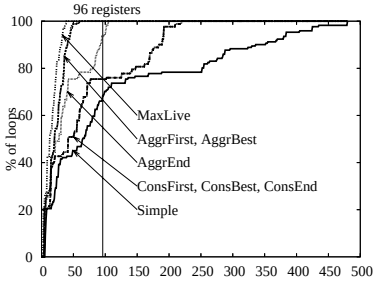
[1] For instance, in Fig. 8(b), the optimal number of registers required is 7, but the maximum number of scalar lifetimes live at each cycle is 6.



(a) Cumulative Distribution of the Properties of the Input Loop Nests (Depth 1-5)



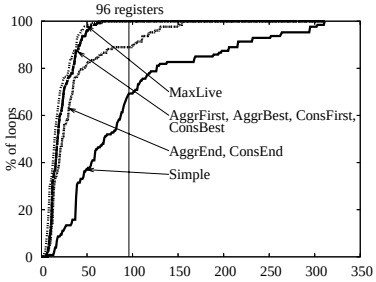
(b) Cumulative Distribution of the Number of Integer Registers Allocated(Depth 2-5)



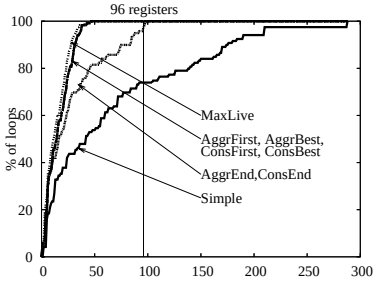
(c) Cumulative Distribution of the Number of Floating-Point Registers Allocated (Depth 2-5)

	Integer		Floating-point	
Depth	MaxLive	AggrFirst	MaxLive	AggrFirst
5	135.00	161.00	14.00	20.67
4	99.42	119.48	15.75	20.50
3	74.70	90.75	19.02	24.68
2	46.92	54.61	17.37	22.86
1	21.78	22.40	15.79	16.19

(d) Average Register Requirement



(e) Cumulative Distribution of the Number of Integer Registers Allocated (Depth 1)



(f) Cumulative Distribution of the Number of Floating-Point Registers Allocated (Depth 1)

Figure 20. Cumulative Distribution Statistics about the Loop Nests

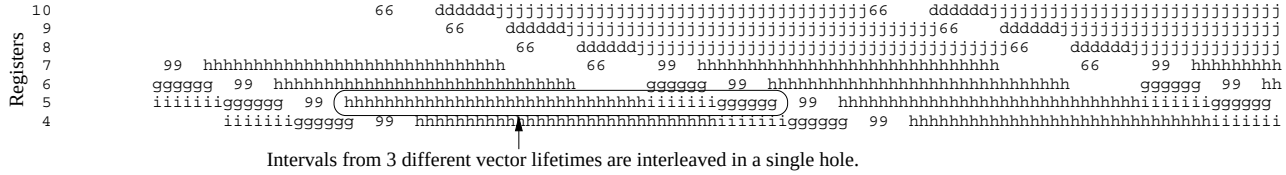


Figure 21. Excerpt of the Integer Space-time Cylinder for a Double Loop Nest

	Average II ratio	Average execution time ratio	#loops failed
Simple	1.81	1.69	11
ConsFirst	1.25	1.28	5

Table 3. Impact of Register Allocation Strategies upon Performance

ter allocation. This paper has presented a systematic solution. The essential problems of lifetime representation and normalization were addressed to precisely abstract the lifetimes. The conservative and aggressive distances were proposed to guide bin-packing and circumference minimization free of conflict, with multiple fit strategies. The method subsumes the classical register allocation for software pipelined single loops as a special case. Experiments indicate that first fit strategy aided with the aggressive distance effectively minimizes register usage with insignificant compile time.

A. Appendix: Formal Definitions for Lifetime Representation

The core and derived parameters are defined from the references set $REFS$. For convenience, we refer to the elements in a reference ref as $time(ref)$, $type(ref)$ (USE or DEF), $omega(ref)$, and $level(ref)$, respectively.

A.1 SingleStart, SingleEnd, Omega, and Alpha

$$singleStart = min\{time(r), \forall r \in REFS \text{ s.t. } type(r) = \text{DEF}\}$$
$$singleEnd = max\{time(r) + 1 + omega(r) * T, \forall r \in REFS\}$$
$$omega = max\{omega(r), \forall r \in REFS\}$$

$SingleStart$ and $singleEnd$ are equivalent to the $start$ and end time used in traditional register allocation for single loops [12], described in Section 2.2. In defining $singleEnd$, not only uses, but also definitions must be considered: in case a variant is defined, but not used, the end time of that variant should be the completion time of the last definition.

In a healthy program, only a reference at the outermost loop level, which is software pipelined, may have a non-zero $omega$. The loop nest in Fig. 3(a) is a typical example. Also, $omega$ of a definition is always 0. $Alpha$ depends on the code outside the loop nest, and cannot be computed from the references. We assume it has already been given by an earlier phase of the compiler.

A.2 Start and End

Given a loop level $i \in [1, n]$, if level i has no definition in $REFS$, we set $(start[i], end[i]) = (+\infty, -\infty)$. Otherwise, let the definition be $d \in REFS$, then

$$\begin{aligned} start[i] &= time(d) \\ end[i] &= max(time(d)+1, time(u)+1+omega(u)*T \\ &\quad +loopbackOffset(d,u), \forall u \in USES(d)) \end{aligned}$$

where $USES(d)$ is the set of all possible uses of d. Note that a value produced at an inner loop level cannot be consumed at the outermost loop level. Otherwise, the dependence (called *negative dependence*) should have prevented the outermost loop level to be software pipelined [14]. Therefore, $USES(d)$ excludes any use at L_1 level if d is at an inner loop level, even if d can reach the use from the standard data flow analysis. Formally,

$$\begin{aligned} USES(d) = \{u | type(u) = \text{USE} \text{ and } d \text{ reaches } u \\ \text{and } (level(u) \neq 1 \text{ or } level(d) = 1)\} \end{aligned}$$

In the term of data flow analysis, a point x in the control flow graph *reaches* another point y if there is a path from x to y without any definition of the variant under consideration between them. The control flow graph in this paper is that for the original loop nest (before software pipelining) defined in Fig. 5(a), which contains n back edges, one for each loop.

$LoopbackOffset(x, y)$ is the offset due to the execution of an inner loop L_j. Since the inner loop runs sequentially, the offset equals $S_j * T$. Formally, if x reaches y without going along the back edge of any inner loop, $loopbackOffset(x, y) = 0$. Otherwise, $loopbackOffset(x, y) = max(S_j * T)$ for any inner loop $L_j (2 \leq j \leq n)$, along whose back edge x can reach y. See the example in Fig. 10(d).

A.3 NextStart

Given a loop level $i \in [1, n]$, let x be a point at this level. If level i has a definition $d \in REFS$, let $x = d$. Otherwise, let x be the starting point of loop L_i, i.e., x is at the first cycle of the first stage of L_i. Then

$$nextStart[i] = \begin{cases} min\{\ time(d') + loopbackOffset(x, d'), \\ \qquad \forall d' \in NEXTDEFS(x)\} \\ \qquad \text{If } NEXTDEFS(x) \text{ is not empty} \\ +\infty \quad \text{Otherwise} \end{cases}$$

where $NEXTDEFS(x)$ is the set of all possible next definitions in the same iteration as point x. The back edge of the outermost loop should not be followed in calculating this set. Otherwise, the set may contain a definition from the next iteration. Formally,

$$\begin{aligned} NEXTDEFS(x) = \{y | y \in REFS \text{ and } type(y) = \text{DEF} \text{ and } x \\ \text{can reach } y \text{ without going along the} \\ \text{back edge of the outermost loop } \} \end{aligned}$$

A.4 FirstStretch and LastStretch

Take the first ILES as a point of reference. Let $x \geq 0$ be the iteration index of a stretched interval at loop level i. The interval is stretched if it is defined before the first cycle of the first ILES, but its last use reaches or goes beyond that cycle, and is not within the first group of iterations. For example, in Fig. 4, the interval of TN2 defined by operation b_3 spans across the first ILES, and the use $c_{3,0}$ is in the second group. Formally, the conditions are:

$$\begin{aligned} start[i] + x * T < l_n * T, \\ end[i] + x * T > l_n * T, \\ x + omega \geq S_n \text{ when } i = 1, \\ x \geq S_n \text{ when } i > 1, \\ \text{and } x \geq 0. \end{aligned}$$

Let $first$ and $last$ be the smallest and largest solutions of x to the above inequalities. That is,

$$\begin{aligned} first &= \min_{\forall i \in [1,n]} max(0, S_n - span, l_n - \left\lfloor \tfrac{end[i]-1}{T} \right\rfloor) \\ last &= \max_{\forall i \in [1,n]} (l_n - \left\lceil \tfrac{start[i]+1}{T} \right\rceil) \end{aligned}$$

where $span = omega$ if $i = 1$, or $span = 0$ otherwise.

Note that when $n = 1$ or $first > last$, there is no stretched interval at all. Thus we define

$$(firstStretch, lastStretch) = \begin{cases} (+\infty, -\infty) & \text{if } n = 1 \text{ or} \\ & \quad first > last \\ (first, last) & \text{otherwise} \end{cases}$$

A.5 Top and Bottom

Again take the first ILES as a point of reference. If *outermostIntervalOnly* is true, the segment is made of only stretched intervals, if any. Otherwise, there are S_n scalar lifetimes from the first group, followed by stretched intervals, if any. For example, in Fig. 6, TN1 has only a stretched interval in the first ILES, while TN2 has $S_n = 3$ scalar lifetimes and two stretched intervals. Therefore,

$$(top, bottom) = \begin{cases} (firstStretch, lastStretch + 1) & \text{if } outer- \\ & \quad mostIntervalOnly \\ (0, max(S_n, lastStretch + 1)) & \text{otherwise} \end{cases}$$

Note that if $outermostIntervalOnly$ is true, and no interval is stretched (i.e., $(firstStretch, lastStretch) = (+\infty, -\infty)$), the ILES would have no interval in it. In this case, according to the formulas, $(top, bottom)$ would be $(+\infty, -\infty)$, as expected.

Acknowledgments

We are grateful to Chan Sun, Ross Towle, Shuxin Yang, Jose Nelson Amaral, R. Govindarajan, Jean C. Beyler, and Michel Strasser for their valuable comments.

References

[1] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995.

[2] D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. In *Proc. of PLDI'91*, pages 192–203, 1991.

[3] S. Carr, C. Ding, and P. Sweany. Improving software pipelining with unroll-and-jam. In *Proc. of HICSS'96*, pages 183–192, 1996.

[4] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proc. of CC'82*, pages 98–101. ACM Press, 1982.

[5] J. C. Dehnert and R. A. Towle. Compiling for the Cydra 5. *J. Supercomput.*, 7(1-2):181–227, 1993.

[6] L. J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. In *Computational Complexity*, pages 176–191, 1992.

[7] R. A. Huff. Lifetime-sensitive modulo scheduling. In *PLDI'93*, pages 258–267, 1993.

[8] Intel. *Intel IA-64 Architecture Software Developer's Manual, Vol. 1*. Intel, 2001.

[9] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. of PLDI'88*, pages 318–328, 1988.

[10] J. Llosa, M. Valero, and E. Ayguad. Heuristics for Register-Constrained Software Pipelining. In *Proc. of MICRO'96*, pages 250–261, 1996.

[11] K. Muthukumar and G. Doshi. Software pipelining of nested loops. *LNCS*, 2027:165–181, 2001.

[12] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *PLDI'92*, pages 283–299, 1992.

[13] H. Rong, A. Douillet, R. Govindarajan, and G. R. Gao. Code generation for single-dimension software pipelining of multi-dimensional loops. In *Proc. of CGO'04*, pages 175–186, 2004.

[14] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao. Single-dimension software pipelining for multi-dimensional loops. In *Proc. of CGO'04*, pages 163–174, 2004.