Register Allocation for Predicated Pipelining using Spiral Graph

Hiroya Itoga
Doctoral Program in Engineering*
University of Tsukuba
Tsukuba, Ibaraki, Japan.

Yoshiyuki Yamashita
Department of Information Science
Saga University
Saga, Japan.

Jiro Tanaka
Institute of Information Sciences and Electronics
University of Tsukuba
Tsukuba, Ibaraki, Japan.

Abstract *The framework of the Spiral Graph is proposed to allocate registers for software pipelining in register-renaming architectures. This will a result in an allocation with the least number of required registers in polynomial time on a rotating register, with the names of the registers renamed one by one and simultaneously. However, the original Spiral Graph cannot inherently manage the conditional branches by predicated execution.*

In this paper the authors propose Predicated Spiral Graph framework for architecture with a rotating register and predicated execution, such as the Intel IA-64 architecture. The proposed method introduces fictitious unit-time intervals to accomplish register allocation with the optimal number of required registers, which is equal to or one more than the maximum number of variables which live simultaneously. The proposed method yields the result in polynomial time.

Keywords: software pipelining, register allocation, register renaming, predicated execution, Spiral Graph.

1 Introduction

The efficiency of register allocation is an important issue in modern processors, where the latency gap increases between the operations and memory references. Register allocation is also more complex since some modern processors have special hardware facilities, such as register renaming[1].

Software pipelining[2] is an optimization method for loop intensive programs using Instruction Level Parallelism (ILP). It schedules the instructions in the iterations in order to overlap partially on the compilation time. Two software pipelining problems have been solved by hardware support: one was that the lifetimes of the variables span more than the *Initiation Interval* (II), which was solved by register renaming architectures. The other was that the various execution paths by the conditional branches, which was solved by predicated execution architectures. For example, the Intel IA-64 architecture has both hardware facilities to support software pipelining: Rotating register and Predication.

The framework of the Spiral Graph[3] is proposed for a rotating register architecture. The graph is an expansion of the Cyclic Interval Graph[4] so it expresses the register renaming as slanted lines on the graph. Figure 1 shows an example of the Spiral Graph. The vertical axis expresses the physical register numbers (names), and the horizontal axis expresses the instruction steps of the program.

The live range *v6* is defined on a rotating

*H. Itoga is now with Ibaraki Industrial Technology Center, Ibaraki, Japan.

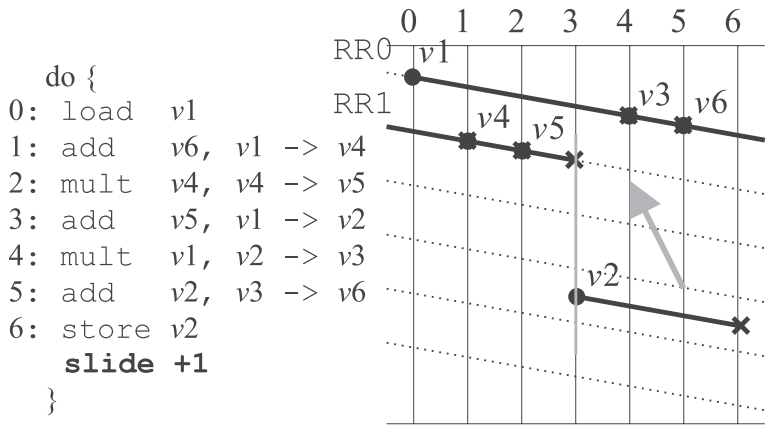

Figure 1: Spiral Graph

register 0 (RR0) and is used on the RR1 since the right and left of the graph are continuous. The Short Bridge Algorithm arranges the live ranges on the graph, narrowing the gaps between the live ranges, similar to *v2*. This framework enables us to obtain the allocation result with the least number of required registers in polynomial time.

However the Spiral Graph framework can not naturally express the predicated execution.

Predicated execution uses a hardware facility on which the instructions are executed according to the predicates. Predicates are added to each instruction by the compilers. If the predicate is set as true, and instruction is executed. If it is not, the instruction does not effect the result of the calculation. The original instruction of a conditional branch is translated into the instruction of the predicate definition: if the condition is true, the specified predicate is set as true.

The translated program for predicated execution has only a single execution path: the processor selects the instructions, which are executed in true condition or false condition of the original conditional branch. Therefore, there are variables that are defined and used only in the true condition or only in the false condition. These live ranges in the true condition and the false condition overlap in the same time. The overlapped live ranges must not be allocated in the same physical register. The live ranges in predicated execution do not practically live in the same time, since the conditions do not become true and false simultaneously. Therefore, live ranges that have different predicates may be allocated in the same physical register even if they overlap. We call this *sharing* of the physical register by the live ranges with the predicates.

The authors explain the algorithm using the sample program shown in Figure 2. Figure 3 shows the scheduled code in software pipelining, with instructions and architecture we assume are similar to the Intel IA-64 architecture. Comments such as F(5*II+0) represent the instruction that the 'F'loating point number is issued in the 5th stage of software pipelining at 0 step of the instruction steps in the iteration.

There are various methods of the ordering between instruction scheduling and register allocation on compilation. The authors assume that the instruction scheduling is already finished when allocating registers , which is referred to as pre-pass scheduling. If register allocation fails because of unsuitable instruction scheduling, the register allocator should return the information to the scheduler and the scheduler should schedule the instructions again under the information.

```
DO I=1, 1000
  IF ((( A(I) + PI )*2.0 ) > 0.0 )
    THEN
      Y(I)=(( X(I)+1.0 )**2.0 + X(I) + 1.0 ) * 3.0
    ELSE
      W(I)=U(I)**2 + 4.0
  END IF
END DO
```

Figure 2: A sample program.

```
L: (p17) ldfd f(d)=X[I+3]             : M(3*II+0)
   (p19) fma  f(f)=f(e)*f(e)+f(e) ;;  : F(5*II+0)

         ldfd f(a)=A[I]               : M(0*II+1)
         fmp  f(c)=f(b)*2          ;; : F(2*II+1)

   (p33) ldfd f(h)=U[I+3]             : M(3*II+2)
   (p18) fadd f(e)=f(d)+1          ;; : F(4*II+2)

   (p20) stfd Y[I+6]=fg               : M(6*II+3)
         fadd f(b)=f(a)+PI         ;; : F(1*II+3)

   (p35) stfd W[I+5]=fi               : M(5*II+4)
   (p34) fma  f(i)=f(h)*f(h)+4    ;;  : F(4*II+4)

   (p19) fmpy f(g)=f(f)*3          ;; : F(5*II+5)

         fcmp.gt p16,p32=f(a),0       : F(2*II+6)
         br.ctop L                 ;; : B
```

Figure 3: A result of instruction scheduling.

2 Predicated Spiral Graph

The authors proposed the framework of the Predicated Spiral Graph to express the predicated execution on the Spiral Graph[5]. The tracks in the original Spiral Graph, which represent the physical registers of the processor, are expanded to a set of "sub-tracks"[1].

Figure 4 shows a portion of the Predicated Spiral Graph. Physical rotating register `RR0` is considered to be a set of sub-tracks, "true" and "false." A specific sub-track represents the physical register when the predicate is set as the related value. The live ranges on the sub-tracks are considered in some ways to be the live range on the main-track. For example, s and t create the live range from `RR1` to `RR2`. We refer to that as the live range s and t *sharing* the physical register.

[1]The concept of sub-tracks on the Predicated Spiral Graph is different from the "sub-tracks" on the original Spiral Graph that express the execution paths in the Enhanced Modulo Scheduling[6].

3 Lower Boundary

The width of the Spiral Graph is defined as the number of variables that lives on each step of the instructions. The maximal number of widths is called $W_{\max}$. $W_{\max}$ is obviously the greatest lower boundary of the number of required registers on the graph or the program.

The authors define the greatest lower boundary on the Predicated Spiral Graph as having the same meaning as that of the original Spiral Graph. The number of variables that live in the same time differs from the greatest lower

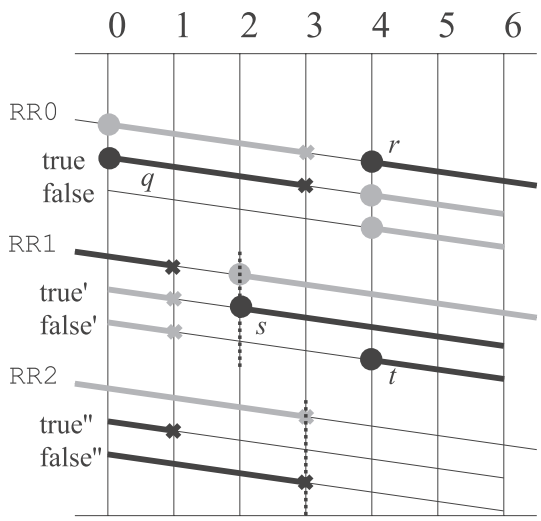

Figure 4: Predicated Spiral Graph.

boundary of the number of required registers since the variables that have different predicates share a physical register on the predicated execution. The $W_{\max}$ of the the Predicated Spiral Graph should be defined as the maximum number of widths possible where the physical registers are shared by any variables on each instruction step. Two variables that share a physical register must live on the same stage in software pipelining and have different predicates.

The width of step 0 in the sample program is 6 since $a_1, b_2, d_3, d_4, f_5, g_6, h_4$, and i_5 live on the step (digits express the number of stages) but d_4 and h_4, as well as f_5 and i_5, live in the same stage and have different predicates, therefore d_4 and, h_4, f_5 and i_5 *share* the same physical register and occupy only one register. The width of step 1 is 7 in the same manner, and the $W_{\max}$ of the graph is 7.

4 Optimal Register Allocation Algorithm

The proposed optimal register allocation algorithm yields an allocation result with the number of required registers equals to $W_{\max} + 1$ in polynomial time. We can obtain the result with $W_{\max}$ when certain conditions are satisfied.

A summary of the optimal register allocation algorithm can be expressed as:

1. introduction of *fictitious intervals*
2. *raising* the predicated live ranges to normal live ranges
3. creation of *Slide Covers* from live ranges
4. rearrangement of live ranges in Slide Covers

4.1 Fictitious Intervals

The authors first introduce two types of fictitious intervals to the Predicated Spiral Graph in the algorithm: normal fictitious intervals and predicated fictitious intervals. Fictitious intervals are the length of one instruction step. They are dealt with as real live ranges in register allocation but are removed after register allocation. These fictitious intervals simplify the problem of connecting the live ranges, raising the predicated live ranges, and creating the Slide Covers. The graph that is introduced in the fictitious intervals is referred to as closure of the original one.

The predicated fictitious intervals are the length of one instruction step with a particular predicate. They are introduced on the sub-tracks of the graph if any predicated live ranges remain by *sharing* when the physical registers are shared as much as possible by the live ranges. Sharing indicates that the predicated live ranges that have exclusive predicates are allocated on the same physical register. Note that the predicated fictitious intervals do not increase the $W_{\max}$ of the graph.

Figure 5 contains a sample of the predicated fictitious intervals on the sample program. The graph is expanded to the number of pipelined stages for simplification. The numbers on the horizontal axis are the number of stages and the divisions are the instruction steps. In the figure, a, b, and c are defined or used on the main-track, and d, e, f and g are defined or used on the sub-tracks of the true condition.

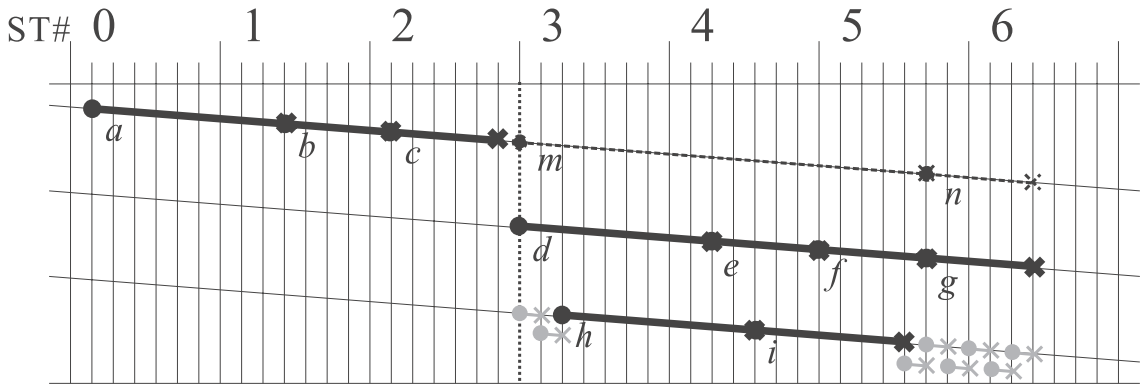

Figure 5: Expansion of Predicated Spiral Graph.

In contrast, h and i are on the sub-track of the false condition. We introduce predicated fictitious intervals (colored gray), that have a predicate of false since the live ranges d, f, and g are on the sub-tracks of the true condition.

Normal fictitious intervals are the length of one instruction step and do not have a predicate. They are the same as those called unit-time intervals[7] or fictitious intervals on the original Spiral Graph. The intervals are introduced to fix the widths of the graph in $W_{\max}$.

4.2 Raising Predicated Live Ranges

We next translate the predicated live ranges into normal live ranges by sharing the registers, while considering the predicates so that all the live ranges are dealt with as normal live ranges. The authors refer to the translation as *raising* the predicated live ranges to normal live ranges.

The predicated live ranges are combined until the end points become the same. The same end point is always found on the specific predicates, since predicated fictitious intervals have been introduced.

We find the normal live ranges m and n on Figure 5, which contain the predicated live ranges. The raised live range m contains the live ranges $(\{d, e, f\}, \{\_, \_, h, i, \_\})$ where $\_$ expresses the fictitious interval. The raised live range n also contains $(\{g\}, \{\_, \_, \_, \_, \_\})$.

4.3 Slide Covers

We can then apply the optimal register allocation algorithm of the original Spiral Graph since no live ranges on the graph have a predicate after the raising translation. The live ranges on the closure, the graph that introduced the fictitious intervals, can only be connected and translated onto some *Slide Covers*.

The connection of live ranges indicates that two live ranges are arranged on the graph without gaps between them: the end step of one and the start step of the other are the same. Note that the start or end step is the modulo II of the start or end point of the connection, and therefore, live ranges a and c have the same start step in the sample program, as in Figure 5.

The Slide Cover is the live range that contains connected live ranges with no gaps and that have the same start step and end step on the graph. The Slide Cover is thus the length of only a multiple of II.

A Spiral Graph that contains only Slide Covers is $W_{\max}+1$-allocatable since we can arrange the Slide Covers on the graph on the order of a monotone increasing as in Figure 6. Eight registers are required, which is equal to $W_{\max}+1$.

4.4 $W_{\max}$-Allocation

Finally, we verify whether the number of the required registers becomes equal to $W_{\max}$,

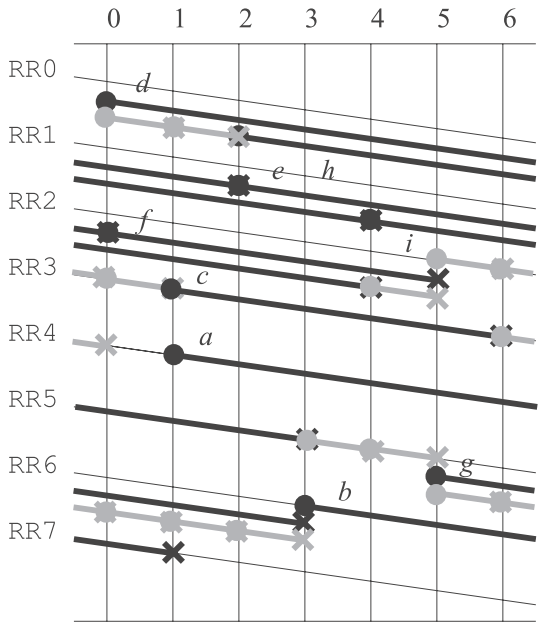

Figure 6: Result of Allocation with $W_{\max} + 1$.

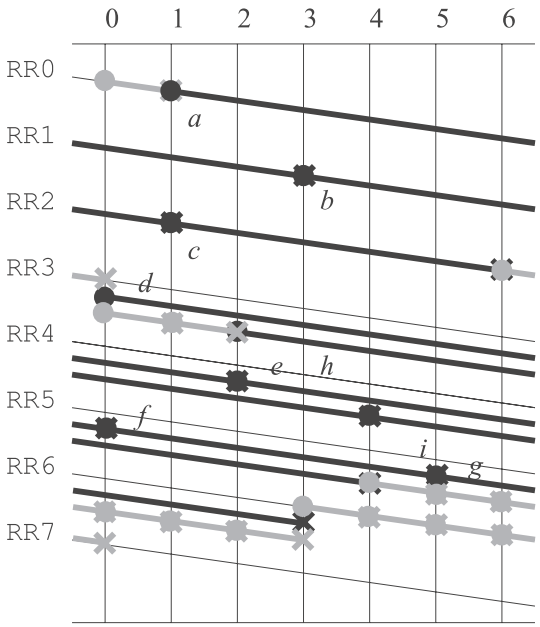

Figure 7: Result of Allocation with $W_{\max}$.

since the result is obviously optimal when the required registers equals to $W_{\max}$. The condition is that all live ranges are connected into only one Slide Cover, and the live ranges at the tail of the Slide Cover before register rotation are fictitious ones.

Rearrangement is effected when any Slide Cover is translated into the Slide Cover that started in the same instruction step with any live ranges the Slide Cover contained. The definition of closure guarantees that there must be live ranges that have the same start step as any live ranges on the graph.

Insertion is effected when a Slide Cover is inserted to another Slide Cover if they have live ranges with the same start step. The Slide Cover is rearranged into a Slide Cover with a particular start step, and it can be inserted into another because the start step and end step of the Slide Cover are the same.

Rearrangement and insertion of the Slide Covers can verify the $W_{\max}$-allocatable condition. In the end, if the condition is satisfied, we can obtain a result with the number of $W_{\max}$, if it is not satisfied, the result yields the number of $W_{\max} + 1$.

Figure 7 illustrates the result with $W_{\max}$ after rearrangement and insertion of the graph of the sample program.

All of these operations on register allocation end in polynomial time when the program has a single conditional branch. The computational complexity of the raising operation increases the power of two of the number of conditional branches. However, the number of conditional branches are usually a one finger digit, which strongly suggests that the register allocation algorithm results with the least number of the required registers in polynomial time.

5 Related Work

A register allocation method using an interference graph for predicated execution has been proposed[8]. In this method, the nodes of the graphs that which have different predicates are bundled into one node. It is difficult to handle software pipelining or register renaming architecture with this method. Furthermore, the

method can not guarantee the optimal number of required registers ion the result since some heuristics yield bundles.

The Meeting Graph[7] has been proposed to allocate registers with the near optimal number of required registers in polynomial time: this method yields a the result below $W_{\max} + 1$. The Meeting Graph considers the continuity of live ranges. The proposed method in this paper deals with predicated execution, which makes the register allocation problem more difficult. Moreover the proposed method describes the conditions necessary to achieve $W_{\max}$-allocation.

6 Conclusion

In this paper, the authors propose a register allocation algorithm for architectures that have the rotating register and predicated execution. The authors explain the Predicated Spiral Graph to represent register sharing by variables with different predicates. The proposed algorithm of the Predicated Spiral Graph provides a result with the optimal number of required registers in polynomial time, if the $W_{\max}$-allocatable condition is satisfied. If it is not satisfied, the algorithm gives a result of $W_{\max} + 1$. The algorithm contributes to the optimizing compilers of architectures such as IA-64, and it provides the benefit of rotating register architecture.

The authors will implement the algorithm on the compiler in future work to measure the performance with the standard benchmarks.

References

[1] Gary R. Beck, David W. L. Yen, and Thomas L. Anderson. The cydra 5 minisupercomputer: Architecture and implementation. In *The Journal of Supercomputing*, volume 7 (1–2), pages 143–180, May 1993.

[2] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, Sep 1995.

[3] Tomohiro Haraikawa, Motohide Soeno, Yoshiyuki Yamashita, and Ikuo Nakata. Register allocation frameworks for slide-window architecture. *Transactions of Information Processing Society of Japan*, 39(9):2684 – 2694, 1998. (in Japanese).

[4] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. In *Lecture Notes in Computer Science (LNCS)*, volume 641, pages 176–191. Springer-Verlag, 1992.

[5] Hiroya Itoga, Tomohiro Haraikawa, Yoshi yuki Yamashita, and Jiro Tanaka. Register allocation for software pipelining with predication using spiral graph. In *Proceedings of the International Symposium on Future Software Technology (ISFST2001)*, pages 58–65, 2001.

[6] Tomohiro Haraikawa. *Register Allocation for Slide-Window Architecture*. Doctoral thesis, doctoral program in engineering, University of Tsukuba, 2000.

[7] Christine Eisenbeis, Sylvain Lelait, and Bruno Marmol. The meeting graph : a new model for loop cyclic register allocation. In *Proceedings of the 5th Workshop on Compilers for Parallel Computers (CPC95)*, pages 503–516, Jun 1995.

[8] Alexandre E. Eichenberger and Edward S. Davidson. Register allocation for predicated code. In *Proceedings of the 28th Annual International Symposium on Microarchitecture (MICRO-28)*, pages 180–191, 1995.