Software Bubbles:
Using Predication to Compensate for Aliasing in Software Pipelines*

Benjamin Goldberg and Emily Crutcher
Department of Computer Science
New York University
goldberg@cs.nyu.edu, emily@cs.nyu.edu

Chad Huneycutt and Krishna Palem
College of Computing and Department of Electrical and Computer Engineering
Georgia Institute of Technology
chadh@cc.gatech.edu, palem@ece.gatech.edu

Abstract

This paper describes a technique for utilizing predication to support software pipelining on EPIC architectures in the presence of dynamic memory aliasing. The essential idea is that the compiler generates an optimistic software-pipelined schedule that assumes there is no memory aliasing. The operations in the pipeline kernel are predicated, however, so that if memory aliasing is detected by a run-time check, the predicate registers are set to disable the iterations that are so tightly overlapped as to violate the memory dependences. We refer to these disabled kernel operations as software bubbles.

1 Introduction

Software pipelining and other methods for parallelizing loops rely on the compiler's ability to find loop-carried dependences. Often the presence or absence of these dependences cannot be determined statically. Consider the fragment in figure 1(a), where the value of `k` cannot be determined at compile time. In such situations, the compiler cannot rule out the possibility that `k`=1, which would create a loop carried dependence with a dependence distance of 1. That is, the value written to `a[i]` in one iteration would be read as `a[i-k]` in the next iteration. Pipelining this loop would provide little benefit, since the store of `a[i]` in one iteration must complete before the load of `a[i-k]` in the next iteration.

For the purposes of this paper, we will refer to dependences that cannot be analyzed at compile time as *dynamic dependences* and those that can be analyzed statically as *static dependences*.

The program fragment in figure 1(b) has the same effect in limiting the compiler's ability to perform software pipelining as the one in figure 1(a). In this example, the arrays `a` and `b` may overlap and the distance (in words) between `a[0]` and `b[0]` corresponds to the value `k` in figure 1(a).

One solution is to generate several different versions of a loop, with different degrees of software pipelining. During execution, a test is made to determine the dependence distance (e.g. `k` in figure 1(a)) and a branch to the appropriately pipelined loop is performed. The drawbacks of this approach include possible code explosion due to the multiple versions of each loop as well as the cost of the branch itself.

This paper presents a technique, called *software bubbling*, for supporting software pipelining in the presence of dynamic dependences without generating multiple versions of the loop. This approach is aimed at EPIC architectures, such as Intel's IA-64 and HP Laboratories' HPL-PD, that support both instruction-level parallelism and predication.

The notation we'll use in this paper for predication, i.e. the ability to disable the execution of an operation based on a one-bit predicate register `p`, is

`p:` *operation*

where *operation* is performed only if the value of the predicate register `p` is 1. In the HPL-PD, thirty-two predicate registers can collectively be read or modified as a single 32-bit register, and in the IA-64 the same is true with 64 predicate registers. We assume a similar capability here, and refer to the aggregate register as `PR`. We'll use the syntax `p[i]` to refer to the ith rotating predicate register (if rotating predicate registers are available) and `pi` to refer to the

*This work has been supported by the Hewlett-Packard Corporation and the DARPA Data Intensive Systems program.

```
for(i=k;i<n;i++)
  a[i] = a[i-k];
```

(a)

```
void copy(int a[],int b[])
{ for(int i=0,i<n;i++)
    a[i] = b[i];
}
```

(b)

Figure 1: Program fragments exhibiting dynamic dependences

ith non-rotating predicate register.

The idea behind software bubbling is that the compiler, when encountering a loop that has dynamic dependences but can otherwise be pipelined, generates a single version of the pipeline that is constrained only by the static dependences and resource constraints known to the compiler. The operations within the pipeline kernel, however, are predicated in such a way that if dynamic dependences arise, the predicate registers are set in a pattern that causes some of the kernel operations to be disabled each time the kernel is executed. With this disabling of operations, the effective overlap of the iterations in the pipeline is reduced sufficiently to satisfy the dynamic dependences. We refer to the disabled operations as *software bubbles*, drawing an analogy with bubbles in processor pipelines. In figure 2, we see an example of a software pipeline with and without bubbling. The crossed out boxes in figure 2(b) correspond to iterations that have been disabled using predication.

2 Predication in Software Pipelining

For clarity of the presentation, we will make the following simplifying assumptions (except where stated otherwise):

- In our examples, all operations have a latency of 1 (i.e. the result is available in the next cycle). Although this assumption is unrealistic, it has no impact on the technical results of this paper. It simply makes our examples more compact. This assumption was *not* made in our experiments.
- Rotating registers, including rotating predicate registers, are available. As with normal software pipeling, rotating registers are a convenience that reduce code size but do not fundamentally change the pipelining or bubbling methods.
- The 32-bit aggregate predication register, PR, is used only to set the predication pattern for bubbling. The predicate registers used for other purposes (conditionals, etc.) are assumed not to be part of PR.
- There is only one dynamic dependence in a loop. This assumption is relaxed in section 10.

We will also use a simple instruction set rather than the HPL-PD or IA-64 ISA. Operations appearing on the same line are assumed to be within the same VLIW instruction.

We continue to use the code fragment in figure 1(a) as the driving example in this paper. We'll assume that the assembly code generated for the body of the loop is

```
                    ; q points to a[i-k]
L1:                 ; s points to a[i]
    r = load q      ; load a[i-k] into r
    store s,r       ; store r into a[i]
    q = add q,4     ; increment q
    s = add s,4     ; increment s
    brd L1
```

where the branch operation, brd, decrements and tests a dedicated loop counter register, LC. If rotating registers are supported, the brd operation also decrements the rotating register base.

A possible pipelining of this loop is illustrated in figure 2(a), where add_1 and add_2 refer to the first and second add operations in the body of the loop, respectively.

In this pipelining, the kernel of the pipeline consists of a single VLIW instruction containing the operations in each horizontal gray box of figure 2(a), namely[1].

```
add₂      add₁      store      load
```

The overlaying of iterations in figure 2(a) assumes that the memory location written by the store operation in one iteration is not read by the load operation in the next iteration. If there were such a dependence between the store and the load, then the two operations could not occur in the same VLIW instruction and the pipelining in figure 2(a) would be incorrect[2].

In our driving example, figure 1(a), the value of the variable k directly determines the dependence distance of the loop-carried dependence. Rather than assume the worst, i.e. that k will be 1, a compiler performing software bubbling generates the most optimistic pipeline, but one in which each kernel operation is predicated. Figure 2(b) illustrates the execution of software-bubbled code in the case where k

[1] We have intentionally oversimplified this discussion by leaving out the operands of the operations in the kernel. We address the issues of operands in subsequent sections.

[2] On the IA-64, unlike HPL-PD, stores and loads to the same address can occur in the same cycle. This increases the possible overlapping of a pipeline by one cycle, but doesn't change the reasoning in this paper

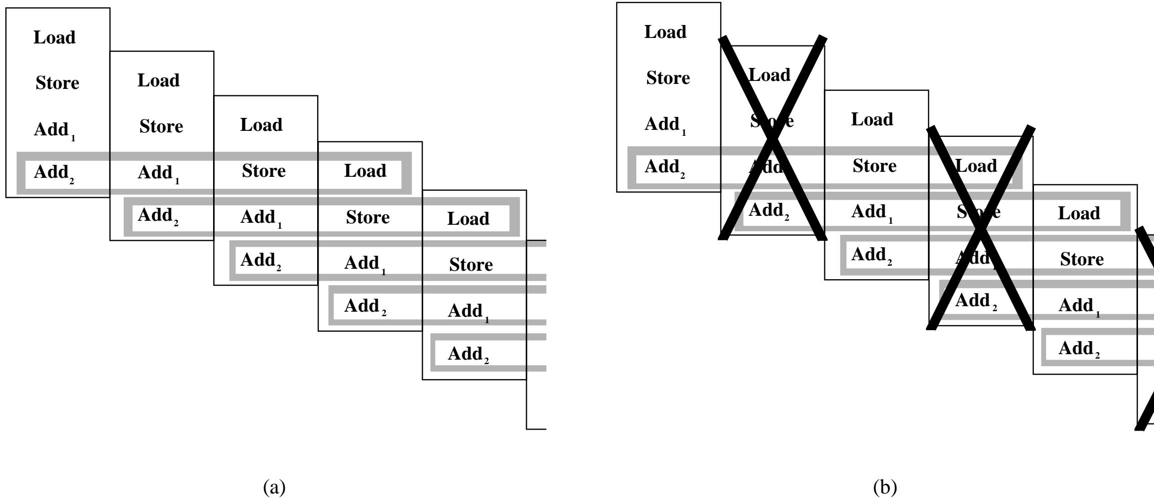


Figure 2: Software pipeline (a) without bubbling and (b) with bubbling

has indeed been determined at run time to be 1. The operations in the crossed-out boxes are disabled by clearing their predicate registers, thus each `load` operation doesn't occur until the cycle after the previous `store`, as desired. If the `store` operation had a latency greater than one, additional iterations would have had to be disabled.

It is helpful to view the crossing out of the boxes in figure 2(b) as simply pushing iterations down and to the right in the pipeline. It is not that the second iteration is not being executed, but rather is being executed in the slot where the third iteration was initially scheduled. Where confusion might arise, we'll use the term *iteration slot* to refer to an iteration in the original schedule (i.e. each box in figure 2(a)) and the term *enabled iteration* to refer to an iteration in an iteration slot that is actually executed. In figure 2(b), every other iteration slot is disabled, so the second enabled iteration is executed in the third iteration slot, the third enabled iteration is executed in the fifth iteration slot, and so on.

3 Simple Bubbling vs. Generalized Bubbling

There are actually two bubbling techniques described in this paper:

1. **Simple Bubbling**, used when the dependence distance, k, is constant over every iteration of the loop (but whose value is unknown at compile time), and

2. **Generalized Bubbling**, used when the dependence distance k changes within the loop.

We describe simple bubbling over the next several sections. Section 8 commences the description of generalized bubbling. Much of what is described for simple bubbling also applies to generalized bubbling.

4 The Kernel in Simple Bubbling

Upon examination of figure 2(b), where every other iteration slot has been disabled, we see that only two operations in the kernel are being executed each time. These two operations are

$\texttt{add}_2$ $\quad$ `store`

followed by

$\texttt{add}_1$ $\quad$ `load`

and repeating in this manner. This pattern can be accomplished by predicating the kernel as follows:

$\texttt{p1:add}_2$ $\quad$ $\texttt{p2:add}_1$ $\quad$ `p1:store` $\quad$ `p2:load`

The first time the kernel is executed, we set `p1` = 1 and `p2` = 0. The second time, we set `p1` = 0 and `p2` = 1. This pattern is repeated over and over. A formal description of how the predication pattern is created is given in subsequent sections.

5 The Predication Pattern in Simple Bubbling

As is common, we refer to the number of cycles from the start of one iteration slot to the start of the next as the *iteration interval*, *II*. We'll use the term *dynamic interval*, *DI*, to refer to the number of cycles between the initiation of successive enabled iterations in the bubbled code. For example, in figure 2(b), *II*=1 and *DI*=2.

For the moment, we only consider bubbling loops that have a single dynamic dependence. Given a dynamic dependence from operation o_1 to operation o_2 with a dependence distance of k, the relationship between *DI* and *II* can be expressed by

$$DI = (L/k) \times II \quad (1)$$

where

$$L = \lceil (latency(o_1) - offset(o_1, o_2))/II \rceil \quad (2)$$

and $offset(o_1, o_2)$ is the number of cycles from the initiation of o_1 to the initiation of o_2 within the same iteration slot.

To understand the formulation of L, notice that the quantity $latency(o_1) - offset(o_1, o_2)$ expresses the number of cycles that must elapse between the start of the iteration containing o_1 and the start of the iteration containing the operation o_2. Since consecutive iteration slots are offset by II, dividing the above quantity by II and taking the ceiling of the result gives you the whole number of iterations slots that must elapse between the iterations containing o_1 and o_2. That is, the operations o_1 and o_2 must be executed L iterations slots apart. Notice that the computation of L does not depend on any run-time quantity, thus is computed by the compiler.

Given a dependence distance of k, we enable k out of every L iteration slots. This way, we are sure that the operations o_1 and o_2, executed k enabled iterations apart, are separated by L iteration slots, satisfying the dynamic dependence.

Consider again our driving example in figure 1(a). In this example, the dependence distance k is exactly the value of the variable `k` in the program. Suppose (for this example) the latency of a `store` is 2. Given the pipeline illustrated in figure 2(a), the value of L is given by

$$\begin{aligned} L &= \lceil (latency(\texttt{store}) - offset(\texttt{store}, \texttt{load}))/II \rceil \\ &= 3 \end{aligned}$$

since $II = 1$, $latency(\texttt{store}) = 2$ and $offset(\texttt{store}, \texttt{load}) = -1$.

Suppose that during execution, upon entry to the loop the value of the variable `k` is 2. Since $L = 3$ and $k = 2$, for every three iteration slots in the original pipeline, only two slots should be enabled. This leads to the bubbling situation illustrated in figure 3(a). The desired predication pattern is achieved by using L predication registers, say `p[1]` through `p[L]`, where the kernel operations from the ith iteration slot are predicated on $p[((i-1) \; mod \; L) + 1]$. Initially, the first k predicate registers are set to 1 and the remaining predicate registers to 0. Upon each execution of the pipeline kernel, the predication pattern rotates, as we saw in figure 2(b).

The predication pattern in simple bubbling is inexpensive to compute at run time. A pattern of k consecutive ones is given by the integer $2^k - 1$, constructed by:

```
PR = shl 1,r_k
PR = sub PR,1
```

where $\texttt{r}_k$ contains k and `PR` is the aggregate predicate register[3]

5.1 The predication rotation

How the rotation of the L-bit pattern in the predicate registers is accomplished depends on whether the machine supports rotating predicate registers or not. If so, then the only extra operation that must be inserted into the kernel is

```
p[0] = move p[L]
```

The next time the kernel is executed (i.e. after the rotating register base is decremented), `p[1]` will contain the value of `p[L]` from the previous kernel, which gives the desired rotating behavior.

If the machine does not support rotating predicate registers, an explicit shift of the aggregate predicate register is necessary.

```
p0 = move pL
PR = shl PR,1
```

The shift operation should be performed in the last cycle of the kernel so that it doesn't affect the other predicated operations in the kernel.

5.2 Determining k

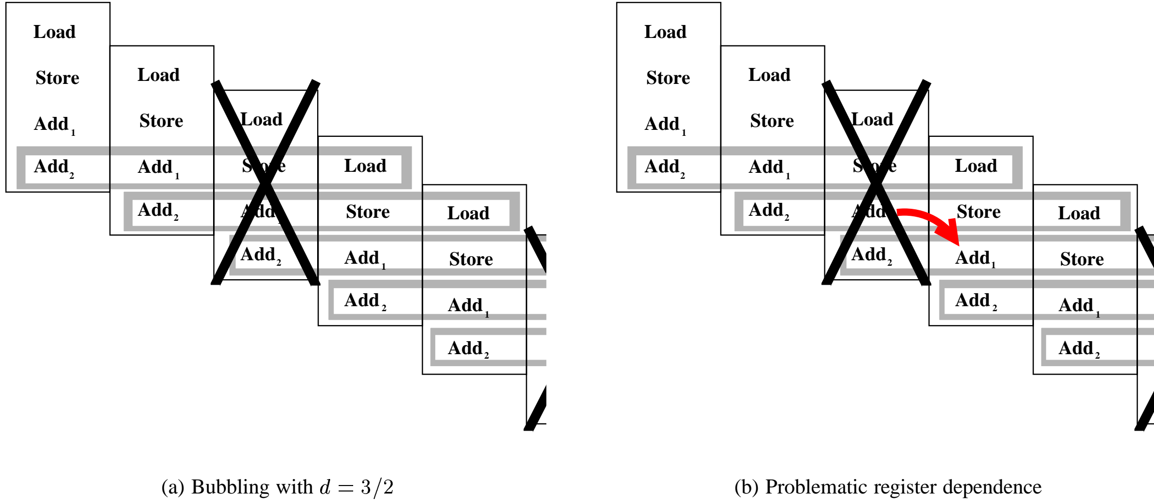
It remains, at run time, to compute the dependence distance, k. The computation of k is generally straightforward, since it essentially arises due to simple array references, as in the driving example, or from the difference between two pointer values, as we saw in figure 1(b). Existing dependence analysis techniques (see Wolfe [11] or Muchnick [5] for a survey of this field) are generally sufficient to determine how k is to be computed. For example, if the compiler computes a dependence distance vector of the form $< 0, 0, ..., 0, \texttt{j} >$, where `j` is a program variable whose value cannot be determined at compile time, then the dependence distance k used for bubbling is the value of the variable `j`.

6 The Loop Counter

If a dedicated loop counter register is used, as is often the case, this loop counter is generally decremented after each execution of the pipeline kernel, as in

```
      LC = ...
L1:
      ...kernel...
      brd L1
```

[3]If the machine doesn't support these operations on `PR`, a general purpose register will have to be used and then copied to `PR`.



(a) Bubbling with $d = 3/2$

(b) Problematic register dependence

Figure 3: Bubbling examples

In the presence of software bubbles, a new iteration isn't necessarily initiated every time through the loop, thus the number of times the loop is executed must be increased.

A brute force way to accomplish this is to observe that a new iteration slot is enabled only when `p[1]` is 1. Thus, the `LC` should only be decremented `when p[1]` is 1, which can be implemented by managing `LC` explicitly and using a branch operation that doesn't decrement `LC`:

```
L1:
   ...<kernel> ...
p[1]: LC = sub LC,1
        p = cmpp> LC,0
p: br L1
```

Another possibility is to insert an operation to increment `LC` whenever `p[1]` is 0

```
p_n[1]: add LC,1
```

where $\mathtt{p}_n$`[1]` is the complement of `p[1]`. If the architecture doesn't support predication upon the complement of a predicate register, then the compiler will have to maintain a set of predicate registers $\mathtt{p}_n$`[0]` through $\mathtt{p}_n[L]$ that contain the complements of `p[0]` through $\mathtt{p}[L]$. Using a two-output `cmpp` operation to assign to `p[0]` and $\mathtt{p}_n$`[0]` simultaneously accomplishes this.

A more attractive method, which can be used only for simple bubbling, is to adjust the value of `LC` before entering the loop. Given a predication pattern of length L of which the first k bits are 1 (i.e. only k iterations are being executed every L iteration slots) the loop counter must be adjusted according to the formula:

$$LC' = (\lfloor (LC/k) \rfloor * L) + (LC \bmod k) \qquad (3)$$

This choice would be worthwhile only if the loop executed a sufficient number of times to amortize the cost of the division and multiplication.

7 Register-Carried Variables

In a software pipeline, there are often register-resident variables that are propagated across iterations in the pipeline (i.e. where one iteration writes to a register and another iteration reads from the same register). This poses a problem for bubbling. Suppose, for example, that an enabled iteration reads a register in order to get the value computed by the previous iteration slot. If the previous iteration slot was disabled, the value found in the register would not be correct. This situation is illustrated in figure 3(b). These register-carried variables can be divided into two categories: induction variables and non-induction variables.

7.1 Induction Variables

A common situation, which we've seen in every example above, is when a register contains an induction variable (a variable whose value changes by some constant amount in each iteration). Figure 4 shows the pipeline kernel, with operands this time, generated from the code in figure 1(a). The induction variables are those containing the addresses of `a[i]` and `a[i-k]`, namely `s` and `q`, respectively. Notice that the operation `s[0]=add s[1],4` uses the value of `s[1]` which was assigned as `s[0]` in the previous cycle by the previous iteration slot. Suppose, however, that every other iteration slot is disabled, as in figure 2(b). In this case, the `s[1]` used by an enabled iteration slot would refer to the `s[0]` assigned to in a disabled iteration slot. Since, due to bubbling, this assignment didn't occur, the use of `s[1]` is incorrect.

When performing simple bubbling, the solution for induction variables, such as `q` and `s` above, arises from noting

```
s[0]=add s[1],4    q[0]=add q[1],4    store s[2],r[1]    r[0]=load q[2]
```

Figure 4: Pipeline Kernel wth Operands

that if an iteration slot is enabled, then the iteration slot L iteration slots before must also be enabled (since the first k out of every L iteration slots are always enabled). Instead of computing the new value of an induction variable based on its value in the previous iteration slot, the value of an induction variable is computed based on its value L iterations slots before [4]. Thus, an assignment to an induction variable of the form

```
r[i] = add r[i+1],d
```

is replaced by

$\texttt{r[i] = add r[i+}L\texttt{], r}_{kd}$

where i+L is a compile-time constant and, before entering the loop, register $\texttt{r}_{kd}$ is assigned the value of $k\times$d.

If the compiler had already performed induction variable expansion, so that the assignment is:

```
r[i] = add r[i+c],e
```

then this operation would be replaced by

$\texttt{r[i] = add r[i+}L\texttt{], r}_{kc}$

where register $\texttt{r}_{kc}$ is assigned the value of $k\times$(e/c), where (e/c) is a compile-time constant.

7.1.1 Other register-carried variables

Not all register-carried values correspond to induction variables, of course. Consider the loop

```
for(i=k;i<n;i++) {
  a[i] = a[i-k] + b
  b = b + i;
}
```

b is not an induction variable and thus its value must be explicitly propagated across disabled iterations. Suppose that the operation for computing b in the pipelined loop is

$\texttt{r[3] = add r[4], r}_i$

where the new value of b is stored in r[3] and computed by adding the value of b from the previous iteration (previously in r[3], now in r[4]) to the register containing i. In the bubbled version, the above operation would be replaced by the two operations

p: $\texttt{r[3] = add r[4],r}_i$ p_n: r[3] = move r[4]

where p would be the predicate register used for bubbling (as usual) and p_n would be the complement of p. This simply ensures that r[3] always contains the current value of b, whether or not that iteration slot is enabled.

[4] This is similar to induction variable expansion used in modulo scheduling.

8 The Generalized the Bubbling Technique

So far, we have assumed that the dependence distance, k, is unknown statically but is constant throughout the loop. We relax that constraint in this section. For illustrative purposes, we will continue to assume that there is a single dynamic dependence, but this is by no means necessary.

As with simple bubbling, the predication pattern is of length L (defined by equation 2 earlier). An iteration with dependence distance k can be enabled only if the kth previous iteration began at least L iteration slots before. That is, a new iteration slot can be enabled only if it would be no more than the kth enabled iteration within the most recent L iteration slots.

Perhaps the easiest way to ensure this condition is satisfied is to use a register r_c to store the number of 1-bits (i.e. enabled iterations) in the predication pattern. As the predication pattern changes due to changing k, the value of r_c is modified. At the beginning of each iteration slot, r_c is compared against r_k, the register containing k, to see if the iteration slot should enabled. The extra code is:

```
p[L+1]: rc  = sub rc,1
        p[1] = cmpp.< rc, rk
p[1]:   rc =  add rc,1
```

The first operation indicates that if a 1 has been shifted out of the first L predicate bits, then r_c should be decremented. The next two operations say that if there are now less than k 1-bits in the predication pattern then p[1] is set, r_c is incremented, and the rest of the operations in the iteration slot (predicated on p[1]) are enabled. Notice that the first operation can be executed concurrently with the computation of r_k and the third operation can be executed concurrently with other operations in the iteration slot. The value of r_c after the third operation is needed to determine if the next iteration slot will be enabled. Thus, unless r_c for an iteration slot can be precomputed several cycles ahead (which in many cases should be possible), iteration slots must be scheduled at least three cycles apart. Thus, a bubbled kernel can be no fewer than three cycles in the generalized case.

Here is a very simple example (a variant of which a number of benchmarks we examined contained):

```
for(i=0;i<n;i++) {
  a[i] += ... a[i] ... a[j] ...;
}
```

where we assume the RHS of the assignment is somewhat expensive to compute. Notice that the dependence distance starts at j-i and decreases in each iteration. Assuming the sequential code for the loop body is:

```
r = load t
q = load s
...
...
...
store r,s
add s,4
```

a safe pipelining of the loop is shown in figure 5(a), providing very little ILP. A bubbled version of the loop code would look like

```
rk = sub rk,1               p[L]: rc = sub rc,1
p[1] = cmpp.< rc, rk
p[1]: rc =  rc,1            p[1]: r = load t
p[1]: q = load s
...
...
...
p[1]: store r,s
p[1]: add s,4
```

The pipelined execution of this bubbled loop is shown in figure 5(b), where b1...b4 refer to the four additional operations required for bubbling. A new iteration is able to start every three cycles (given a sufficient number of functional units), regardless of the size of the body of the loop.

The three cycle lower limit for the kernel holds only when there is no early computation of r_c and r_k for an iteration slot. In fact, the simple bubbling mechanism (described in previous sections) for an invariant k can be looked at as a precomputation of r_c and r_k for all iteration slots, outside the loop.

An additional cost incurred by generalized bubbling is the cost of passing register-carried values across disabled iteration slots, as described in section 7. Taking advantage of induction variables (as described in section 7.1) is not generally possible, since there is no guarantee that if an iteration slot is enabled, then the iteration slot L iteration slots before was also enabled (due to the changing predication pattern). Therefore, the technique described in section 7.1.1 of inserting operations to propagate register values across disabled iteration slots must be used. This adds one `move` operation in the bubbled kernel for each such variable.

9 Bubbling vs. Using Disambiguation Hardware

Several EPIC architectures, including HPL-PD and the IA-64, contain operations that support run-time memory disambiguation. In HPL-PD, for example, the speculative load operation,

`lds` *loc*

performs a load from the location *loc*. The load-verify operation,

`ldv` *loc*

checks to see if a store to *loc* has occurred since the last `lds` from *loc*. If so, a new load is issued and the processor stalls. Otherwise, execution proceeds without an additional load. In the IA-64, the ALAT facility provides the same functionality.

A question that naturally arises is: Why is software bubbling better than using `lds` and `ldv` for pipelining in the presence of aliasing? Consider again the code in figure 1(a), along with the pipelined code for it in figure 2(a). If it is possible for the store operation in one iteration to write to the same location as that loaded in the next iteration (i.e. where `k`=1), then, using `lds/ldv`, we still have to be sure that the store operation in one iteration occurs before the `ldv` operation of the next iteration. At best, the code would have to be as shown in figure 6(a). Notice that the iteration interval is 2, compared to 1 in the bubbled case. This is because the `ldv` in each iteration must occur after the store in the previous iteration if aliasing is possible.

It may be that a store appears much further down the body of the loop than the potentially conflicting load, as seen in figure 6(b). In such a case, pipelining using `lds/ldv` provides very little instruction level parallelism.

10 Handling Multiple Dynamic Dependences

If there are several dynamic dependences in the loop, generalized bubbling can still be performed. What is required is that L is computed statically for each dependence and a separate r_k and r_c is maintained for each such dependence at run time. The r_c for a given dependence counts how many 1-bits in the predication pattern there are in the first L bits, for that particular dependence's L. If the r_c is less than the r_k for each dependence, the iteration can be enabled.

For simple bubbling, a fixed predication pattern must be created upon entry to the loop such that for each dependence i and each k_i and L_i for that dependence, there are no more than k_i 1-bits out of L_i bits of the predication pattern. Although this is not cheap to compute, it occurs only when the loop is entered. Essentially, it involves finding at compile time L_m, the largest L value, and finding at run-time the smallest ratio of k_i to L_i over all the dependences. Then, upon entry to the loop a predication pattern of length L_m consisting of a repeated pattern of k_i 1-bits followed by $L_i - k_i$ 0-bits is created. We have found that, for the benchmarks we considered, this pattern can be created in 20 to 30 cycles before entering the loop.

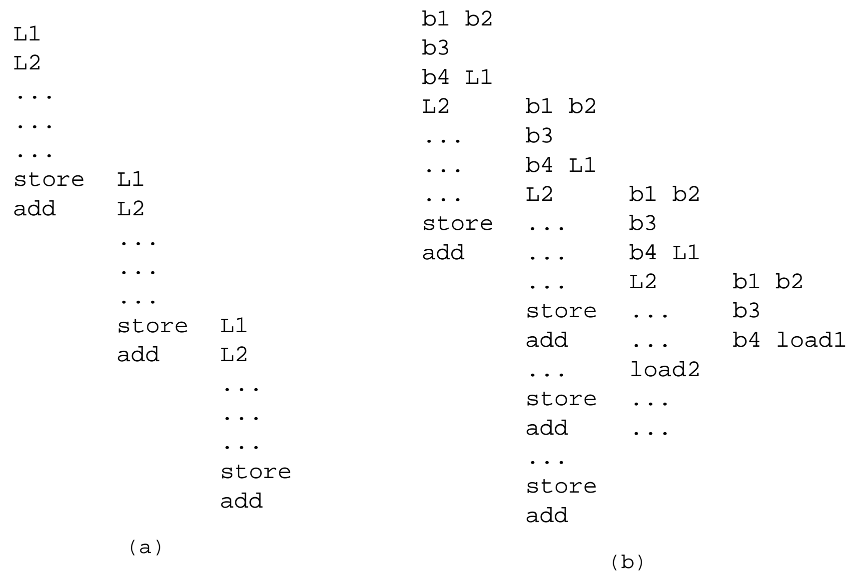


Figure 5: Software pipeline (a) without bubbling and (b) with generalized bubbling

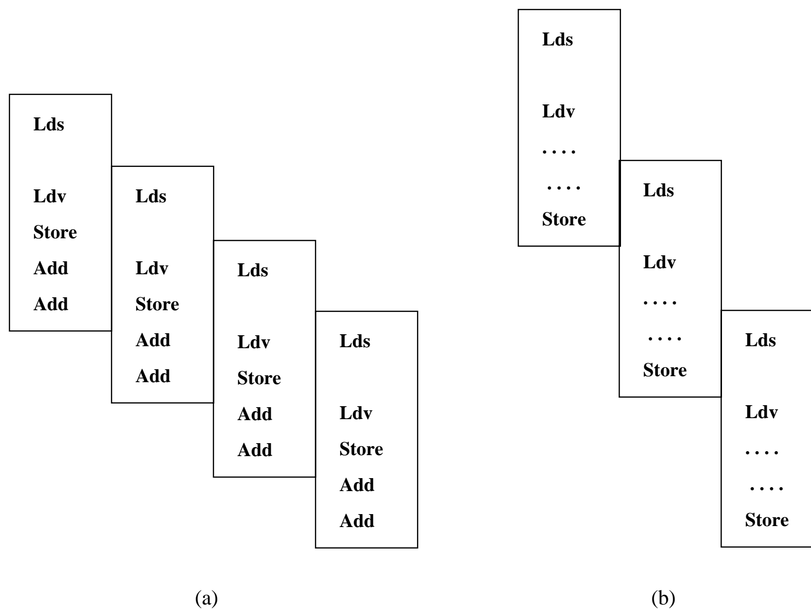


Figure 6: Software pipeline using `lds`/`ldv`

11 Experimental Results

A number of experiments were performed using the Trimaran Compiler Research Infrastructure [9], whose compiler performs modulo scheduling. We first examined the literature on dynamic disambiguation, particularly [4], [8], and [2]. These papers identified some benchmarks where dynamic disambiguation improved performance. We chose several of these benchmarks to determine the effectiveness of software bubbling on a wide range of machine configurations whose number and types of functional units varied.

11.1 Simple Bubbling

For simple bubbling, where the dependence is constant within a loop, there were only a few examples in common benchmarks. These cases arise, for example, in loops for performing operations among arrays (such as copying, addition, etc.) where the source and target arrays are not actually aliased but the C compiler cannot determine that for certain. We ran three such codes, a matrix copy, matrix addition, and "Loop s152" taken from the Callahan-Dongara-Levine benchmark suite [3] for vectorizing compilers. This last piece of code was identified in [4] as benefiting from dynamic memory disambiguation for C compilers. The results of the benchmarks for five different machine configurations are presented in table 1. Each machine configuration in the leftmost column is given as a triple identifying the number of integer, floating point, and memory functional units, respectively. The total cycles are given for the benchmarks when safe pipelining and when software bubbling were used. The speedup factor (safe_cycles divided by bubbled_cycles) is given as well. `mcopy` is an interesting case because, since the loop is very short, the little bit of bubbling overhead incurred actually degraded performance on the smallest machine.

The speedups due to bubbling, over the safe pipelining performed using modulo scheduling, are significant. However, it is not clear that such examples are sufficiently common to provide substantial benefit on a range of computing problems. Furthermore, it seems apparent that since the arrays are rarely aliased, a test for aliasing followed by a conditional jump to a pessimistically pipelined loop is probably a better choice.

11.2 Generalized Bubbling

Generalized bubbling, where k varies within the loop, shows possibly greater potential than simple bubbling for improving performance on a range of programs. Such loops are quite common, including those with array indirection, such as

```
   a[b[i]] = ...a[i] ...
```

and nested loops, such as

```
for(i=0;i<n;i++)
 for(j=0;j<m;j++)
    a[i] += ... a[j] ...
```

as well as in a variety of other situations.

For experimentation, we chose two benchmarks that contained loops with varying k, namely the SPEC Alvinn benchmark and the Livermore Loops Kernel2 code. The loop of interest in the Alvinn Benchmark is

```
for (; sender <= end_sender; )
  *receiver += (*sender++) * (*weight++);
```

which is manipulating floating point data and appears in the `input_hidden()` procedure. This benchmark, and the loop in particular, was identified in [2] as being amenable to dynamic disambiguation. Notice the varying dependence distance between *receiver and *sender and between *receiver and *weight. This loop is actually more complex than it appears, since many machines require a conversion of the loaded values to double precision for the multiplication and then a conversion back to single precision after the multiplication.

The loop of interest in the Livermore Loops Kernel2 code is

```
for ( j=ipnt+1 ; j<ipntp ; j=j+2 ) {
  i++;
  x[i] = x[j] - v[j]*x[j-1] - v[j+1]*x[j+1];
}
```

Notice that the dependence distance varies because `i` is increasing by one and `j` by two.

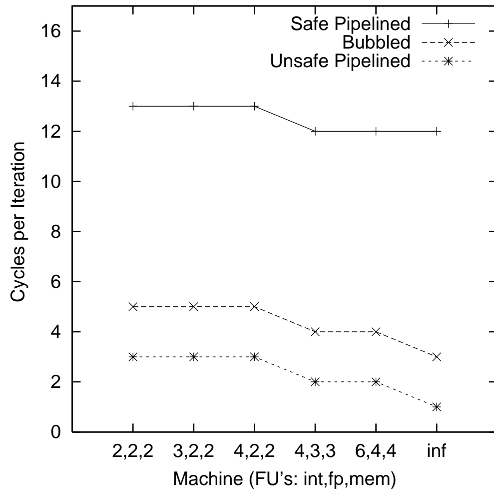
Figure 7shows the number of cycles per iteration of the above loops in the two benchmarks when performing safe pipelining, generalized software bubbling, and unsafe pipelining (which possibly violates dependences), for the same variety of machine configurations as before. In addition, the machine configuration `inf` corresponds to a machine with a unbounded number of functional units.

Although bubbling was more expensive than unsafe pipelining, it is a clear improvement over the safe pipelining generated by Trimaran's modulo scheduler. Because generalized bubbling adds a number of integer operations, both for modifying of the predication pattern and for propagating register-carried values across disabled iteration slots, machines with few integer functional units will be less likely to exploit bubbling usefully. Figure 8 compares the total execution time for the two benchmarks using safe pipelining and bubbling, for the various machine configurations. The overall improvement is substantial, mainly for machines with a sufficient number of integer functional units.

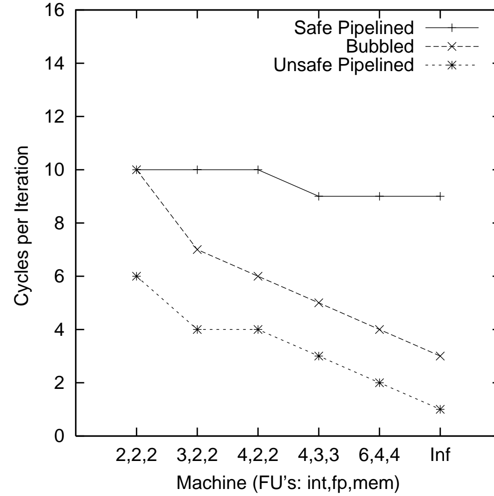
Admittedly, the experimental work presented here does not necessarily provide a compelling reason for adopting software bubbling. In particular, it remains to be seen if the technique has truly wide application. A source of difficulty in finding standard benchmarks that exhibit dynamic aliasing is the fact that the benchmarks were written in order to test the compilers ability to find static aliasing. Thus, few

	S152			madd			mcopy		
machine	safe	bubbling	speedup	safe	bubbling	speedup	safe	bubbling	speedup
2,2,2	711802	263102	2.71	87699	63539	1.38	55398	57158	0.97
3,2,2	609602	160902	3.79	81219	57059	1.42	55298	50578	1.09
4,2,2	609498	160798	3.79	74497	50337	1.48	55057	50337	1.09
4,3,3	609498	160798	3.79	74497	50337	1.48	55057	50337	1.09
6,4,4	608389	109589	5.55	74257	43617	1.70	48497	37297	1.30

Table 1: Speedups due to Simple Bubbling

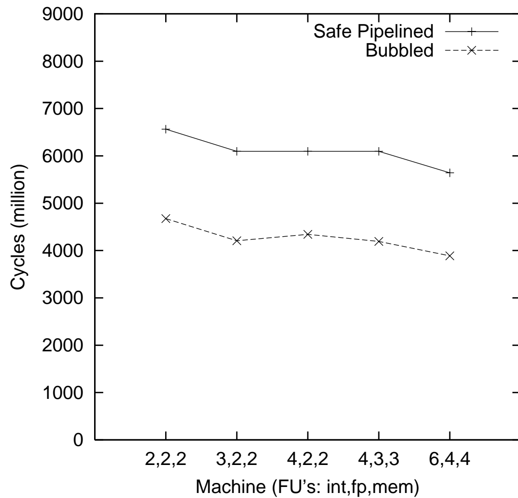


(a) Alvinn

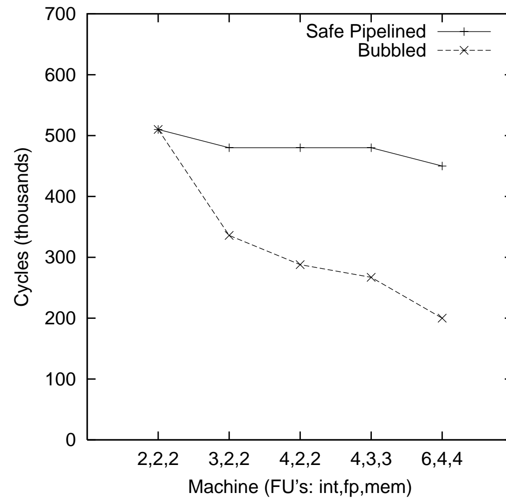


(b) Livermore Loops Kernel 2

Figure 7: Number of Cycles per Iteration



(a) Alvinn



(b) Livermore Loops Kernel 2

Figure 8: Total Execution Time

such benchmarks include true dynamic aliasing. Clearly, further experimental work is needed.

12 Related Work

This is the first technique that we know of that uses predication to compensate for memory aliasing in software pipelines. There is, of course, a large body of work on the area of software pipelining, see [1] for an extensive survey of the field. A small portion this work is concerned with software pipelining in the presence of memory aliasing. Davidson et. al [4] describe a method for using dynamic memory disambiguation and loop unrolling to improve software pipelining performance. The result of the disambiguation test is a possible jump to sequential code. Similar work is described by Bernstein et al. [2], where a run-time test and a jump to a less aggressively parallelized version of the code is performed. Su et al. [8] performed an empirical study of memory aliasing in software pipelined code, finding that it occurs very rarely, even in those cases where the compiler determines that aliasing is possible. In the same paper, Su et al. describe a scheme where run-time checks with jumps to compensation code are inserted within the pipelined code. An early use of run-time disambiguation for VLIW machines was described by Nicolau [6].

There is also work on using predication in software pipelining, although not to handle memory aliasing. Warter et. al. [10] showed that performing if-conversion on conditional statements in loop bodies greatly facilitates pipelining. Predication within software pipelined code, quite similar in flavor to the work we present here but for a different purpose, is described by Rau et al. in [7] and used to obviate the need for separate pipeline prologue and epilogue code. In this *kernel-only* scheme, predicated kernel operations are gradually enabled during the prologue and then gradually disabled during the epilogue.

References

[1] V. Allan, R. Jones, R. Lee, and S. Allan. Software pipelining. *ACM Computing Surveys*, 27(3), 1995.

[2] D. Bernstein, D. Cohen, and D. Maydan. Dynamic memory disambiguation for array references. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 105–112., San Jose, CA, November 1994.

[3] J. Dongarra D. Callahan and D. Levine. Vectorizing compilers: a test suite and results. In *Supercomputing '88*, pages 98–105, 1988.

[4] J. Davidson and S. Jinturkar. Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation. In *Proceedings of the 28th annual international symposium on Microarchitecture*, pages 125–132, 1995.

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

[6] A. Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, 38(5):663–678, 1989.

[7] B. R. Rau, M. Schlansker, and P. Tirumalai. Code generation schema for modulo scheduled loops. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 158–169, Portland, OR, December 1992.

[8] B. Su, S. Habib, W. Zhao, J. Wang, and Y. Wu. A study of pointer aliasing for software pipelining using run-time disambiguation. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 112–117, November 1994.

[9] The Trimaran Compiler Research Infrastrcture, http://www.trimaran.org.

[10] N. Warter, D. Lavery, , and W-M. Hwu. The benefit of predicated execution for software pipelining. In *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, pages 497–506, Wailea, Hawaii, January 1993.

[11] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1995.