

Amortization of Programs: Amortization Schedule

Consider the problem of computing an amortization schedule for a loan of size L to be paid in equal amounts over N periods with interest rate r . The schedule in general is determined as follows:

$$\left. \begin{aligned} bal[0] &= L \\ bal[i] &= bal[i-1] - pr^i[i] \\ int[i] &= bal[i-1] \cdot r \\ pr^i[i] &= pay - int[i] \end{aligned} \right\} \text{For every } i > 0$$

This yields the following recurrence equation for $bal[i]$, $i > 0$:

$$bal[i] = bal[i-1] - 1 \cdot q - pay,$$

$$\text{where } q = 1 + r.$$

It solution is given by

$$bal[i] = L \cdot q^i - pay \cdot (q^i - 1)/r$$

By setting $i = N$ and requiring that $bal[N] = 0$, we obtain

$$pay = L \cdot r \cdot q / (q^N - 1)$$

A Sequential Program for Amortization Schedule

```

Program Amortization-Schedule
always
   $q = r + 1$ 
   $pay = L \cdot r \cdot q^N / (q^N - 1)$ 
   $bal[0] = L$ 
   $\langle \square \ i : 0 < i \leq N ::$ 
     $bal[i] = bal[i - 1] - pr[i]$ 
     $\square \ int[i] = bal[i - 1] \cdot r$ 
     $\square \ pr[i] = pay - int[i]$ 
   $\rangle$ 
end Amortization-Schedule

```

Since the set of equations is proper (dependency graph is acyclic), they admit a sequential implementation.

A Parallel Version

The above equations cannot be directly parallelized. However, since we already derived an explicit formula for $bal[i]$ we can simplify it as follows:

$$bal[i] = L \cdot q^i - pay \cdot (q^i - 1)/r = (L - pay/r) \cdot q^i + pay/r$$

Now we can organize these equations in a form which leads to synchronous parallelization as follows:

Program Parallel-Amortization-Schedule
always

$q = r + 1$ \square $pay = L \cdot r \cdot q^N / (q^N - 1)$ \square $pay/r = pay/r$

$Lpayr = L - payr$ \square $bal[0] = L$

$\langle \parallel i : 0 \leq i \leq N :: bal[i] = Lpayr \cdot q^i + payr \rangle$

$\langle \parallel i : 0 < i \leq N :: int[i] = bal[i - 1] \cdot r \rangle$

$\langle \parallel i : 0 < i \leq N :: pri[i] = pay - int[i] \rangle$

end Parallel-Amortization-Schedule

A Saddle Point of a Matrix

For an integer array $A[0..N-1, 0..N-1]$, element $A[u, v]$ is called a **saddle point** if

$$is_sp(u, v) : A[u, v] = \langle \min i : 0 \leq i < N :: A[i, v] \rangle = \langle \max j : 0 \leq j < N :: A[u, j] \rangle$$

That is, $A[u, v]$ is the minimal in its column and the maximal in its row.

We are required to compute a boolean variable sp whose value is 1 iff array A has a saddle point.

Observe that for any u, v ,

$$\langle \min i :: A[i, v] \rangle \leq A[u, v] \leq \langle \max j :: A[u, j] \rangle$$

Hence $A[u, v]$ is a saddle point iff

$$\langle \min i :: A[i, v] \rangle \geq \langle \max j :: A[u, j] \rangle$$

Therefore

$$sp = \exists u, v :: \langle \min i :: A[i, v] \rangle \geq \langle \max j :: A[u, j] \rangle$$

Let $X[v] = \langle \min i :: A[i, v] \rangle$ and $Y[u] = \langle \max j :: A[u, j] \rangle$. Then

$$sp = \langle \exists u, v :: X[v] \geq Y[u] \rangle = \langle \langle \max v :: X[v] \rangle \geq \langle \min u :: Y[u] \rangle \rangle$$

A Program Testing for the Existence of a Saddle Point

```

Program Saddle-Point
  declare  $X, Y$  : array[0..N-1] of integer
  always

```

```

  <|  $v :: X[v] = \langle \min i :: A[i, v] \rangle \parallel \langle \max j :: A[u, j] \rangle \rangle =$ 
  <|  $sp = \langle \max v :: X[v] \rangle \geq \langle \min u :: Y[u] \rangle \rangle$ 
end Saddle-Point

```

This program can be implemented on a

- Sequential architecture with complexity $O(N^2)$,
- On a synchronous parallel architecture with N processors with complexity $O(N)$,
- On a parallel architecture with N^2 processors with complexity $O(\log N)$.

A Binary Addition Circuit

Consider a circuit (program) which adds two binary numbers represented by the bit arrays $A[0..N-1]$ and $B[0..N-1]$. We will concentrate on the computation of the array of carry bits $C[0..N-1]$. The conventional (sequential) computation of carry's follows the following definition:

$$C[0] = 0 \quad \wedge \langle \forall i : 0 \leq i < N :: C[i+1] = (A[i] + B[i] + C[i] > 1) \rangle$$

The following claim implies an algorithm which enables parallel computation of the carry array in $O(\log N)$ steps, using $O(N)$ parallel processors.

Claim 5. [parallel carry computation]

$$\langle \forall i : 0 \leq i < N :: C[i+1] = A[i] \text{ if } A[i] = B[i] \sim C[i] \text{ if } A[i] \neq B[i] \rangle$$

The claim is proven by case analysis on $A[i] = B[i] = 0$, $A[i] = B[i] = 1$, and $A[i] \neq B[i]$.

A Parallelizable Algorithm

Program *Carry*

declare

d : **array**[0..N-1] **of** (0, 1, U)

t : **integer**

initially

$t = 1 \parallel d[0] = 0 \parallel$

assign

$\langle \parallel i : 0 \leq i < N :: d[i + 1] = A[i] \text{ if } A[i] = B[i] \sim U \text{ if } A[i] \neq B[i] \rangle$
 $\langle \parallel i : t \leq i \leq N :: d[i] := d[i - t] \text{ if } d[i] = U \rangle$
 $\parallel t := 2 \cdot t \text{ if } t \leq N$

end *Carry*

Correctness of this program can be proved using the following invariant:

$$t > 0 \wedge \langle \forall i : 0 \leq i < t \wedge i \leq N :: d[i] = C[i] \rangle$$

$$\wedge \langle \forall i : t \leq i \leq N :: d[i] \neq U \rightarrow d[i] = C[i] \rangle$$

$$\wedge \langle \forall j : i - t \leq j < i :: A[j] \neq B[j] \rangle$$

Using TLV to Model and Verify Unity Programs

We show how to model and verify program *Carry*. In file `carry.smv`, we place:

```

MODULE main
  DEFINE N      := 12;
          N-1 := N - 1;
          N2   := N+N;
  C[0] := 0;
  For (i=0; i < N; i=i+1) {C[i+1] := (A[i]+B[i]+C[i]>1);}
  VAR      AA : array 0..N-1 of boolean;
          BB : array 0..N-1 of boolean;
          d  : array 0..N of {0,1,UU};
          t  : 1..N2;
  For (i=0; i<N; i=i+1) {P[i] : doi(i,AA,BB,d,t,N);}
  ASSIGN
    !init(t) := 1;
    !init(d[0]) := 0;
    next(t) := case
      t <= N : t+t;
      1 : t;
    esac;
    next(d[0]) := 0;

```

```

MODULE doi(i,AA,BB,d,t,N)
  DEFINE
    i1 := i + 1;
    diff := i1 - t;
    indx := case
      diff < 0 : 0;
      diff > N : N;
      diff > 0 : 1
    esac;
    diff > 0 : 0;
    diff > N : N;
    diff > 0 : 1
  esac;
  ASSIGN
    init(d[i1]) := case
      AA[i]=BB[i] : AA[i];
      1 : UU;
    esac;
    next(d[i1]) := case
      d[i1]=UU : d[indx];
      1 : d[i1];
    esac;
    next(AA[i]) := AA[i];
    next(BB[i]) := BB[i];
  !

```

File carry.pf

```

To prepare!
Let degc := 1; -- Partial correctness assertion
For (i in 0..N)
  Let degc := degc & (d[i]=c[i]);
End -- For (i in 0..N)
-- Construct invariant
Let inva := t>0;
For (i in 0..N)
  Let i-1 := i - 1;
  Let inva := inva & (d[i] = uu) -> (d[i]=c[i]);
  Let invj := 1;
  For (j in 0..i-1)
    Let invj := invj & ((j > i - t) | (AA[j] i = BB[j]));
  End -- For (j in 0..i-1)
  Let inva := inva & (d[i]=uu) -> invj;
End -- For (i in 0..N)
End -- To prepare!

```

Rest of File carry.pf

```
prepare!  
Print "\n Model Check Partial correctness\n"!  
Let parcor := t>N -> degc!  
Call Invariance(parcor) !  
Print "\n Deductive Verification\n"!  
Call inv(parcor, inva) !
```