Reuse-Aware Modulo Scheduling for Stream Processors

Li Wang
National Laboratory for
Parallel and Distributed Processing,
School of Computer, NUDT, China

Jingling Xue
Programming Languages & Compilers Group,
School of Computer Science and
Engineering, UNSW, Australia

Xuejun Yang
National Laboratory for
Parallel and Distributed Processing,
School of Computer, NUDT, China

Abstract—**This paper presents reuse-aware modulo scheduling to maximizing stream reuse and improving concurrency for stream-level loops running on stream processors. The novelty lies in the development of a new representation for an unrolled and software-pipelined stream-level loop using a set of *reuse equations*, resulting in simultaneous optimization of two performance objectives for the loop, reuse and concurrency, in a unified framework. We have implemented this work in the compiler developed for our 64-bit FT64 stream processor. Our experimental results obtained on FT64 and by simulation using nine representative stream applications demonstrate the effectiveness of the proposed approach.**

I. Introduction

The stream processors, such as Imagine [6], Cell [9], AMD FireStream and Merrimac [2], represent a promising alternative in achieving high performance in media applications [6] and some scientific applications [2], [9]. In [12], we introduced the design and fabrication of FT64, the first 64-bit stream processor for scientific computing. Like Imagine [6], Cell [9] and Merrimac [2], FT64 is well mapped to the SVM (stream virtual machine) architecture [4].

Under the stream programming model, such as Brook [1], StreamC/KernelC [3], an application is divided into a *stream-level program* running on the host and a *kernel-level program* running on the stream processor. The former specifies the flow of streams between kernels and initiates the execution of kernels. The latter executes these kernels compiled to VLIW microcode on clusters of ALUs, one at a time.

Stream processors incorporate a software-managed on-chip memory, called the SRF (Stream Register File), as the nexus of the ALUs and off-chip memory to store streams. It is performance-critical to fully utilize SRF to capture the abundant stream reuse in stream applications so as to minimize the expensive off-chip memory traffic. In addition, a stream processor typically supports a few concurrent stream I/O operations that can be overlapped with kernel execution. This enables the SRF to hide memory latency if prefetching loads and delayed stores are scheduled judiciously.

This work is concerned with the development of compiler techniques for optimizing stream-level programs (or loops) by maximizing the reuse between streams and improving the concurrency through overlapping kernel execution and stream I/O on SVM-like stream processors (which is well mapped to the stream virtual machine architecture [4]). Its novelty lies in formulating the underlying optimization problem via a set of *reuse equations* and performing both unrolling and software pipelining in a unified framework.

Loop unrolling and software pipelining [7] can expose stream reuse and concurrency, respectively, but they aim to achieve different goals. When applied to stream-level loops separately, either reuse or concurrency or both may be inadequately exploited (as discussed in Section II).

This paper makes the following contributions:

- We characterize an unrolled and software-pipelined stream-level loop using a set of reuse equations, by which both reuse and concurrency are made explicit (Section III).
- We propose a reuse-aware modulo scheduling technique to simultaneously maximize stream reuse and improve concurrency for a stream-level loop based on the notion of reuse equations (Section IV).
- We demonstrate the effectiveness of our compiler technique by running nine representative benchmarks on FT64 and by simulation (Section V).

II. A Motivating Example

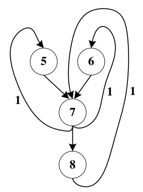
We use an example to motivate the development of Reuse-Aware Modulo Scheduling (RAMS). In FT64, there are three resources, two memory units, $\mathtt{Mem}_1$ and $\mathtt{Mem}_2$, for performing stream transfers between off-chip memory and the SRF and one stream processor, `Exe`, for executing kernels (one at a time). To focus on the basic idea, we assume that all loads, stores and kernels are each executed in one cycle.

Modulo scheduling [7], the most popular technique for software pipelining, aims to minimize the II (the Initiation Interval between successive iterations) of a loop. The algorithm starts with the Minimum II, i.e., $\text{MII} = \max(\text{ResII}, \text{RecII})$. ResII is the largest N/R, where N is the number of units of a particular resource required by the loop and R is the number of such units available (per cycle). RecII is the largest $\text{delay}(c)/\text{dist}(c)$ among all recurrence cycles c in the DDG (Data Dependence Graph) of the loop, where $\text{delay}(c)$ is the sum of the latencies of all operations in c and $\text{dist}(c)$ is the sum of the dependence distances of all edges in c.

Consider a stream-level loop given in Figure 1. In line 1, two arrays are declared. In line 2, three streams of size `N` are declared. In line 3, `dataInit` is called to initialize `A`

```
1   float A[(M+2)*N], O[M*N];
2   stream<float> a(N), b(N), o(N);
3   dataInit('amatrix.dat', A);
4   for (k = 0; k < M; k++) {
5     Load(A[k*N, (k+1)*N-1], a);
6     Load(A[(k+2)*N, (k+3)*N-1], b);
7     Kernel('foo', a, b, o);
8     Store(o, O[k*N, (k+1)*N-1]);
9   }
10  dataSave('omatrix.dat', O);
```

(a) Code



(b) DDG

Fig. 1: An example loop (with only the non-zero dependence distances for inter-iteration dependences shown).

residing in off-chip memory. In lines 5 and 6, the two data sections in `A` are gathered into streams `a` and `b`, resulting in the two data sections being loaded from off-chip memory into the SRF. In line 7, the kernel named `foo` is called to perform its computation on the stream processor. As shown, `a` and `b` are input streams and `o` is an output stream. In line 8, `o` is stored from the SRF back into `O` in off-chip memory. In line 10, `dataSave` is called to save the data of `O` into a file. There is a reuse from the second load in line 6 to the first load in line 5. Thus, the first load at iteration k accesses, i.e., reuses the same stream as the second load at iteration $k-2$.

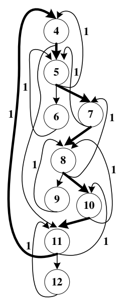
If we apply modulo scheduling to this loop without unrolling it first, a redundant load, i.e., operation 5 is executed every iteration since the reuse is not exploited. If we perform loop unrolling first, we must know what its best unroll factor is. Figure 2(a) shows the unrolled code with an unroll factor $\mathcal{F}=3$. The reuse could now be exploited. Applying modulo scheduling to the DDG in Figure 2(b) yields the schedule in Figure 3. In this case, $\mathrm{II} = \mathrm{MII} = 6$, meaning that each iteration in the original loop takes $6/3 = 2$ cycles to complete. Although the reuse is fully exploited, concurrency is poor — kernel execution and memory transfers are not overlapped.

```
    ...
1   Load(A[0*N, 1*N-1], a_1);
2   Load(A[1*N, 2*N-1], c_1);
3   for (k = 0; k < M; k+=3) {
4     Load(A[(k+2)*N, (k+3)*N-1], b);
5     Kernel('foo', a_1, b, o_1);
6     Store(o_1, O[k*N, (k+1)*N-1]);

7     Load(A[(k+3)*N, (k+4)*N-1], a_1);
8     Kernel('foo', c_1, a_1, o_2);
9     Store(o_2, O[(k+1)*N, (k+2)*N-1]);

10    Load(A[(k+4)*N, (k+5)*N-1], c_1);
11    Kernel('foo', b, c_1, o_3);
12    Store(o_3, O[(k+2)*N, (k+3)*N-1]);
13  }
    ...
```

(a) Code



(b) DDG

Fig. 2: Unrolled loop of Figure 1(a) with $\mathcal{F}=3$.

Figure 4 shows the optimal schedule found by RAMS for this loop. As shown in Figures 4(a) and (b), the reuse is fully exploited. In addition, the highest throughput is attained with one iteration executed per cycle. So concurrency is maximized, too. Both performance objectives are optimized in the final loop in Figure 4(c) obtained when $\mathcal{F}=4$ (not 3).

This example demonstrates that the best unroll factor, and consequently, the optimal performance of a stream-level loop is determined by not only the amount of reuse but also the

	Exe	Mem$_1$	Mem$_2$
0		4	
1	5		
2		7	6
3	8		
4		10	9
5	11		
6			12
...	...	...	...
11			

(a) Scheduling

	Exe	Mem$_1$	Mem$_2$
0		4[k]	12[k-1]
1	5[k]		
2		7[k]	6[k]
3	8[k]		
4		10[k]	9[k]
5	11[k]		

(b) Kernel

```
    ...
    for (k = 0; k < M-3; k+=3) {
4     Load(A[(k+5)*N, (k+6)*N-1], b);
12    Store(o_3, O[(k+2)*N, (k+3)*N-1]);
5     Kernel('foo', a_1, b, o_1);

7     Load(A[(k+6)*N, (k+7)*N-1], a_1);
6     Store(o_1, O[(k+3)*N, (k+4)*N-1]);
8     Kernel('foo', c_1, a_1, o_2);

10    Load(A[(k+7)*N, (k+8)*N-1], c_1);
9     Store(o_2, O[(k+4)*N, (k+5)*N-1]);
11    Kernel('foo', b, c_1, o_3);
    }
    ...
```

(c) Kernel loop

Fig. 3: Modulo scheduling of unrolled loop in Figure 2 (a), where the numbers on the left side of the charts are cycles, the ones inside the charts are the operations (identified by their line numbers) in the loop in Figure 2 (a) and $I[K]$ stands for the instance of operation I at iteration K of the loop.

	Exe	Mem$_1$	Mem$_2$
0		6	
1	7		
2			8

(a) Scheduling

	Exe	Mem$_1$	Mem$_2$
0	7[i-1]	6[i]	8[i-2]

(b) Kernel

```
   ...
1  for (k = 0; k < M-4; k+=4) {
2    Load(A[(k+4)*N, (k+5)*N-1], a);  Kernel('foo', c, d, o_2);  Store(o_1, O[k*N, (k+1)*N-1]);
3    Load(A[(k+5)*N, (k+6)*N-1], c);  Kernel('foo', b, a, o_1);  Store(o_2, O[(k+1)*N, (k+2)*N-1]);
4    Load(A[(k+6)*N, (k+7)*N-1], b);  Kernel('foo', d, c, o_2);  Store(o_1, O[(k+2)*N, (k+3)*N-1]);
5    Load(A[(k+7)*N, (k+8)*N-1], d);  Kernel('foo', a, b, o_1);  Store(o_2, O[(k+3)*N, (k+4)*N-1]);
6  }
   ...
```

(c) Kernel loop

Fig. 4: Optimal schedule with an unroll factor $\mathcal{F}=4$.

amount of concurrency exploited. One ad hoc approach is to try a limited number of unroll factors and settle with the best one [5]. To avoid this, RAMS generates an unrolled- and modulo-scheduled loop from a stream-level loop systematically by simultaneously maximizing stream reuse and improving concurrency, as shown in Figure 4, based on a new concept of reuse equations introduced below.

III. Reuse Equations

The key insight behind RAMS is the realization that every unrolled and software-pipelined loop of a stream-level loop $\mathcal{L}$ can be characterized by a set of *reuse equations* for its steady state. Then the problem of finding a desirable schedule for $\mathcal{L}$ is simply reduced to one of building from $\mathcal{L}$ a set of reuse equations, from which the schedule is derived.

There are two types of equations: DR (Data Reuse) and SR (Space Reuse) equations. Every reuse equation specifies a reuse between two streams at a fixed reuse distance. For a DR equation, there is a reuse of both the data and space/name between the two streams. For an SR equation, only the space/name is reused. The set of DR equations dictates the amount of reuse to be realized while the set of SR equations constrains the amount of concurrency in a schedule of $\mathcal{L}$. Both determine the minimal unroll factor used to maximize reuse and concurrency in a schedule of $\mathcal{L}$.

A. Establishing Reuse Equations

Let $s(i,j)$ be the i-th stream parameter of the j-th kernel in $\mathcal{L}$. Every reuse equation (or relation) has the form:

$$e(i,j,k) \quad = \quad e(i',j',k-d), \quad k \geqslant d \tag{1}$$

which indicates the existence of a reuse flowing from $s(i', j')$ at iteration $k-d$ to $s(i,j)$ at iteration k with a reuse distance of d iterations. For the three indices in $e(i,j,k)$, i and j start from 1 and k from 0. Note that $e(i,j,k)$ enjoys no reuse when $k < d$. How this equation is initialized at its boundary is done in the prologue of a schedule generated by RAMS.

Let all stream parameters in $\mathcal{L}$ be divided into m *reuse groups*, $\mathcal{R}_1, \ldots, \mathcal{R}_m$, where $\mathcal{R}_r = \{s(i_1^r, j_1^r), \ldots, s(i_n^r, j_n^r)\}$ such that there is a reuse from $s(i_{\ell-1}^r, j_{\ell-1}^r)$ to $s(i_\ell^r, j_\ell^r)$ with a distance of d_ℓ^r. Thus, every $\mathcal{R}_r$ induces the DR equations:

$$\begin{array}{rcll} e(i_2^r, j_2^r, k) & = & e(i_1^r, j_1^r, k - d_2^r), & k \geqslant d_2^r \\ e(i_3^r, j_3^r, k) & = & e(i_2^r, j_2^r, k - d_3^r), & k \geqslant d_3^r \\ \ldots & & & \\ e(i_n^r j_n^r, k) & = & e(i_{n-1}^r, j_{n-1}^r, k - d_n^r), & k \geqslant d_n^r \end{array} \tag{2}$$

which are obtained by performing a stream reuse analysis in $\mathcal{L}$ [13]. This sequence of reuse relations forms a path, called a ***DR path***, of length $D_r = \sum_{\ell=2}^n d_\ell^r$. The first parameter $s(i_1^r, j_1^r)$ is the *producer stream*, which is either an input stream to be loaded by a load instruction in $\mathcal{L}$ or an output stream stored by a store instruction in $\mathcal{L}$. The remaining $n-1$ parameters are *consumer* streams, which may be initialized by different loads in $\mathcal{L}$. To complete this path into a reuse cycle of length $\sum_{\ell=1}^n d_\ell^r$, we add the following SR equation:

$$e(i_1^r, j_1^r, k) \;=\; e(i_n^r, j_n^r, k - d_1^r), \quad k \geqslant d_1^r \tag{3}$$

where the reuse distance d_1^r, found by RAMS, affects the amount of renaming introduced in a schedule of $\mathcal{L}$, and consequently, determines the amount of concurrency exploitable and SRF pressure introduced in the schedule. The reuse distance d_1^r is related to a false dependence in $\mathcal{L}$. A detailed discussion on all these is given in Section III-B below.

For the loop in Figure 1, there are two reuse groups: $\mathcal{R}_1 = \{s(1,1), s(2,1)\}$ and $\mathcal{R}_2 = \{s(3,1)\}$. We obtain the first two equations for $\mathcal{R}_1$ and the last one for $\mathcal{R}_2$ below:

$$\begin{array}{lrcl} \mathsf{DR}: & e(1,1,k) & = & e(2,1,k-2), k \geqslant 2 \\ \mathsf{SR}: & e(2,1,k) & = & e(1,1,k-d_1), k \geqslant d_1 \\ \mathsf{SR}: & e(3,1,k) & = & e(3,1,k-d_2), k \geqslant d_2 \end{array} \tag{4}$$

which can be visualized in Figure 5. Since it is unnecessary to consider transitive reuse edges, the reuse inherent in a reuse group always manifests itself as a reuse cycle graphically.

From now on, the edges on all DR paths are called ***DR edges*** and the edges corresponding to all SR equations ***SR edges***. Every reuse cycle has only one SR edge.

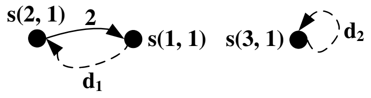


Fig. 5: Visualizing the reuse equations of (4) (with the DR edges in solid lines and SR edges in dashed lines).

B. Understanding Reuse Equations

The set of DR equations is determined by a stream reuse analysis in $\mathcal{L}$ [13]. Let us consider the DR path P_r of length $D_r = \sum_{\ell=2}^n d_\ell^r$ associated with $\mathcal{R}_r$ whose DR equations are given in (2) and SR equation in (3). Consider the iteration k of $\mathcal{L}$ when the producer stream $S_r = s(i_1^r, j_1^r)$ is generated (loaded or stored). This stream will be reused or consumed along P_r, first by $s(i_2^r, j_2^r)$ at iteration $k + d_2^r$, then by $s(i_3^r, j_3^r)$ at $k + d_2^r + d_3^r$, and eventually by $s(i_n^r, j_n^r)$ at $k + \sum_{\ell=2}^n d_\ell^r$.

To exploit all reuse on P_r, we must unroll $\mathcal{L}$ by at least $D_r + d_{1,\min}^r$ times, where $d_{1,\min}^r$ is the minimal reuse distance being substituted for d_1^r in the SR equation (3):

$$d_{1,\min}^r \;=\; \begin{cases} 0 & \text{if } j_1^r > j_n^r \\ 1 & \text{otherwise} \end{cases} \tag{5}$$

Here, $j_1^r > j_n^r$ implies the j_1^r-th kernel, where the producer S_r is generated, appears lexically after the j_n^r-th kernel in $\mathcal{L}$. This way, we can eliminate the loads in $\mathcal{L}$ for initializing redundantly the last $n-1$ stream parameters on P_r by replacing them with the producer S_r. As a result, the live range of S_r has been extended effectively from one single iteration in $\mathcal{L}$ to $D_r + d_{1,\min}^r$ iterations in the unrolled loop.

To expose simultaneously all reuse inherent in all m reuse groups, $\mathcal{R}_1, \ldots, \mathcal{R}_m$, in the unrolled and modulo-pipelined loop of $\mathcal{L}$, the unroll factor is chosen to be:

$$\mathcal{F} = \mathrm{LCM}(L_1, L_2, ..., L_m), \;\; L_r = \sum_{\ell=1}^n d_\ell^r \tag{6}$$

where L_r is the length of the reuse cycle for $\mathcal{R}_r$ ($1 \leqslant r \leqslant m$).

For our example discussed earlier with respect to (4), $L_1 = 2 + d_1$ and $L_2 = d_2$, where d_1 and d_2 are the reuse distances in the two SR equations that are to be determined by RAMS. $\mathcal{F} = \mathrm{LCM}(L_1, L_2) = \mathrm{LCM}(2 + d_1, d_2)$.

The reuse distance d_1^r in the SR equation (3), i.e., the weight of its SR edge, which is found by RAMS (Section IV), specifies the amount of renaming allowed for the stream S_r represented by the first node on the path P_r. By renaming S_r for $d_1^r - d_{1,\min}^r$ times in d_1^r consecutive iterations, we increase concurrency by overlapping prefetching loads, delayed stores and kernel execution at some slight increase in SRF pressure.

IV. Reuse-Aware Modulo Scheduling

RAMS takes as input a stream-level loop $\mathcal{L}$ characterized by a set of reuse equations and produces as output an unrolled and modulo-scheduled loop for stream processors. RAMS is reuse-aware since it fully realizes the reuse specified by the reuse equations when looking for a schedule to improve concurrency in $\mathcal{L}$. In addition, an unroll factor is automatically derived for realizing the reuse and concurrency exposed.

A. Modulo Scheduling for Concurrency

The input to ConMax in Algorithm 1 is a loop $\mathcal{L}$ together with a set of reuse equations for the loop, where the weights of DR edges in (2) are known and the weights of SR edges in (3) are not. However, the weight of every SR edge is initialized by (5) as its minimum value. The output is a modulo schedule for $\mathcal{L}$ with the weights of all SR edges being finalized.

The execution time of a load or store for a stream of size s is estimated linearly as $a_{\text{startup}} + b_{\text{byte\_cost}} \times s$, where s is the

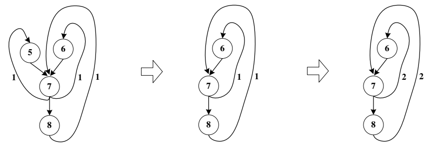
Algorithm 1 Modulo scheduling for concurrency

```
1:  Procedure Increase_SR-Rep_Edge_Weight()
2:  for every unvisited SR-rep edge $e$ (satisfying $e$.visited = false) in an
      SR-rep cycle $c$ such that $\mathrm{delay}(c)/\mathrm{dist}(c) > \mathrm{II}$ do
3:      Increase the weight of $e$ by 1 and set $e$.visited = true
4:  end for
5:  End Procedure
6:  Procedure ConMax()
7:  Input: Loop $\mathcal{L}$ and its set of reuse equations (2) and (3)
8:  Output: A schedule $\alpha$
9:  Let $\mathcal{R}_1, \ldots, \mathcal{R}_m$ be all reuse groups given in (2) and (3)
10: Remove from $\mathcal{L}$ the loads corresponding to all the stream parameters
      except the producer in each reuse group $\mathcal{R}_r$, where $1 \leqslant r \leqslant m$
11: Build the DDG for the modified $\mathcal{L}$
12: For the SR edge $d_1^r$ in every reuse group $\mathcal{R}_r$, let $\mathrm{rep}(d_1^r) =_{\mathrm{def}} e$ be the
      unique false dependence edge $e$, called the *SR-rep edge*, connected to the
      node that defines stream $S_r$ in the DDG, and set $e$.visited = false
13: Compute ResII for the stream processor
14: Compute $\mathrm{RecII} = \max_c(\mathrm{delay}(c)/\mathrm{dist}'(c))$ for all recurrence cycles $c$
      in the DDG, where $\mathrm{dist}'(c) = \mathrm{dist}(c) + 1$ if $c$ is an SR-rep cycle (i.e.,
      a cycle containing an SR-rep edge) and $\mathrm{dist}'(c) = \mathrm{dist}(c)$ otherwise.
15: Initialize $\mathrm{II} = \mathrm{MII} = \max(\mathrm{ResII}, \mathrm{RecII})$
16: Increase_SR-Rep_Edge_Weight()
17: while iterative modulo scheduling with II fails do
18:     Let $op_1, \ldots, op_{n-1}$ be all the successfully scheduled operations and
          $op_n$ be the current operation that cannot be scheduled with II
19:     if there is an unvisited SR-rep edge $e$ (satisfying $e$.visited = false)
          such that one of its two nodes is some $op_i$ for $1 \leqslant i \leqslant n$ then
20:         Increase the weight of $e$ by 1 and set $e$.visited = true
21:     else
22:         $\mathrm{II} = \mathrm{II} + 1$
23:         Increase_SR-Rep_Edge_Weight()
24:     end if
25: end while
26: for every SR-rep edge $e$ in the DDG such that $e$.visited = true do
27:     Increase the weight of the corresponding SR edge $\mathrm{rep}^{-1}(e)$ by 1.
28: end for
29: return The schedule $\alpha$ found
30: End Procedure
```

size of the stream and a_{startup} and $b_{\mathrm{byte\_cost}}$ are architecture-dependent. The execution time of a kernel call can then be estimated from its compiled VLIW code in the usual manner.

In ConMax, we apply iterative modulo scheduling to find a schedule iteratively for $\mathcal{L}$. However, the presence of SR edges leads to a fundamental difference. In ConMax, we judiciously increase the weight of a false dependence in the DDG that is uniquely related to the SR edge for a reuse group $\mathcal{R}_r$ to reduce $\mathrm{delay}(c)/\mathrm{dist}(c)$ for the corresponding recurrence cycle c that contains the false dependence. By increasing the weight of the SR edge (indirectly this way), the producer stream parameter S_r in $\mathcal{R}_r$ is renamed so that the corresponding load (store) for S_r is turned into a prefetching load (delayed store) in the generated schedule, resulting in improved concurrency.

Starting with all m reuse groups in line 9, we modify $\mathcal{L}$ in line 10 by deleting the loads corresponding to all the consumer stream parameters in every reuse group $\mathcal{R}_r$ and then obtain in line 11 a simplified DDG for the new $\mathcal{L}$. The SR edge d_1^r for $\mathcal{R}_r$ given in (3) is defined in the DDG of the original loop $\mathcal{L}$. In line 12, we identify its unique representative edge e, denoted $\mathrm{rep}(d_1^r) =_{\mathrm{def}} e$, in the simplified DDG. After all redundant loads in $\mathcal{R}_r$ are removed (line 10), the producer stream S_r is involved in a recurrence cycle c with two nodes: a load/store *op* for S_r and a kernel call with S_r as a parameter. If S_r is an input stream, then *op* is a load that fetches S_r from off-chip memory into the SRF to be used in the kernel. If S_R is an output stream, then *op* is a store that writes back S_r produced by the kernel from the SRF into off-chip memory.



(a) Original (b) Simplified (c) Final

Fig. 6: The DDGs of the loop in Figure 1.

In either case, one edge in the dependence cycle c represents a true dependence and the other a false dependence. In addition, one of the two edges has the weight 1 and the other 0 so that $\mathrm{dist}(c) = 1$. As a result, d_1^r in the DDG of the original $\mathcal{L}$ is identified with the false dependence e, known as its *SR-rep edge*, in the simplified DDG: $\mathrm{rep}(d_1^r) =_{\mathrm{def}} e$. Such a cycle is called a *SR-rep* cycle. During scheduling, increasing the weight of e amounts to increasing d_1^r, causing S_r to be renamed as motivated earlier in Section III-B.

Given a recurrence cycle, the classic scheduling constraint $\mathrm{delay}(c) \leqslant \mathrm{dist}(c) \times \mathrm{II}$ must be satisfied. This implies that a dependence path cannot span more than $\mathrm{dist}(c) + 1$ stages. When there are m resources available, it is generally sufficient to have m concurrent stages being executed in parallel. In FT64, $m = 3$. So a maximum of three stages is sufficient to maximize concurrency based on our experience. Since $\mathrm{dist}(c) = 1$ for an SR-rep cycle c in the DDG initially, it suffices to increase the weight of the SR-rep edge in the cycle by 1 to create concurrency. As renaming may increase SRF pressure, ConMax introduces it only wherever necessary. In line 13, we calculate ResII as usual. In line 14, we calculate RecII also as usual except that $\mathrm{delay}(c)/(\mathrm{dist}(c) + 1)$ rather than $\mathrm{delay}(c)/\mathrm{dist}(c)$ is used for an SR-rep cycle since its $\mathrm{dist}(c)$ can be increased by 1 to improve concurrency.

Let us examine the rest of ConMax. Each SR-rep edge e is initially marked as unvisited in line 12 to prevent its weight from being increased twice. In line 16, we call Increase_SR-Rep_Edge_Weight to introduce renaming into every SR-rep cycle that prevents a feasible schedule to be found with the current II. In lines 17 – 25, we apply iterative modulo scheduling by starting with II initialized in line 15. Whenever a schedule cannot be found for the current II, we try to reduce $\mathrm{delay}(c)/\mathrm{dist}(c)$ in lines 18 – 20 by increasing the weight of an unvisited SR-rep edge e in an SR-rep cycle c by 1. Note that a schedule with the current II may be found next due to a relaxation of dependences in line 20 (even though $\mathrm{delay}(c)/\mathrm{dist}(c) \leq \mathrm{II}$ for the corresponding SR-rep cycle just before line 20). Otherwise, we search for a feasible schedule at the next larger II (line 22) with the minimal amount of renaming introduced in line 23. Finally, in lines 26 – 28, any weight increase in an SR-rep edge is reflected in the corresponding SR edge. In line 29, the schedule α is found.

Let us apply ConMax to our example loop. We reproduce in Figure 6(a) its DDG given earlier in Figure 1. Recall that this example has two reuse groups $\mathcal{R}_1 = \{s(1,1), s(2,1)\}$ and $\mathcal{R}_2 = \{s(3,1)\}$. Its reuse equations are given in (4). After the load corresponding to $s(1,1)$ is deleted in line 10, the simplified DDG found in line 11 is given in Figure 6(b). As illustrated in Figure 5, there are two SR edges d_1 and d_2 associated with $\mathcal{R}_1$ and $\mathcal{R}_2$, respectively. They are initialized to be $d_1 = d_2 = 1$ by (5) in line 7. We know that $s(2,1)$ denotes `b` and $s(3,1)$ denotes `c`. In line 12, the algorithm finds that $\text{rep}(d_1) = (7,6)$ and $\text{rep}(d_2) = (8,7)$ with both being initialized as $(7,6).\text{visited} = (8,7).\text{visited} = \mathsf{false}$.

Given the simplified DDG in Figure 6(b), we obtain $\text{ResII} = 1$ in line 13 and $\text{RecII} = 1$ in line 14 (in contrast to $\text{RecII} = 2$ obtained traditionally). Thus, ConMax starts to schedule the loop with $\text{II} = \text{MII} = 1$ initialized in line 15. In line 16, the weights of the two SR-rep edges, $(7,6)$ and $(8,7)$, are increased from 1 to 2. Given the modified DDG in Figure 6(c), ConMax has found an optimal schedule with $\text{II} = 1$ in Figure 4. Since $(7,6).\text{visited} = (8,7).\text{visited} = \mathsf{true}$, the weights of d_1 and d_2 are increased by 1 so that $d_1 = 2$ and $d_2 = 2$ (lines 26 – 28). The final DDG is shown in Figure 6(c). Correspondingly, the reuse equations in (4) are:

$$\begin{array}{rrcl} \mathsf{DR}: & e(1,1,k) & = & e(2,1,k-2), k \geqslant 2 \\ \mathsf{SR}: & e(2,1,k) & = & e(1,1,k-2), k \geqslant 2 \\ \mathsf{SR}: & e(3,1,k) & = & e(3,1,k-2), k \geqslant 2 \end{array} \quad (7)$$

If renaming is not used, one will start and end with $\text{II} = \text{MII} = 2$. The schedule has poor concurrency since kernel execution cannot be overlapped with the load/store operations.

B. Code Generation

Given a modulo schedule, we apply our code generation algorithm (not shown) to turn it into an unrolled and modulo-pipelined loop. The kernel for our example, with the prologue and epilogue omitted, is given in Figure 4.

V. Experimental Results

We evaluate our work using nine representative scientific kernels listed in Table I both on our 64-bit stream processor FT64 [12] and by simulation.

Benchmark	**Size**	**Reuse**	**MD**	**Unroll Factors**		
				R	**RC**	**RCM**
Swim-calc1	512^2	C	Y	2	6	3
Swim-calc2	512^2	C	Y	2	6	3
EP	131072	N	N	1	1	1
FFT	4096	C	N	2	2	2
GEMM	512^2	I	Y	1	2	2
Jacobi	512^2	I	Y	1	2	2
Laplace	512^2	C	Y	2	6	3
NLAG-5	512^2	C	Y	3	4	4
MG	128^3	C	Y	3	4	4

TABLE I: Scientific applications.

A. Performance on FT64

In Table I, Swim-calc1 and Swim-calc2 are from SPEC 2000, EP and MG from NPB, GEMM from BLAS and Lalapce from NCSA. NLAG-5 is a nonlinear algebra solver for 2D nonlinear diffusion of hydrodynamics. These benchmarks are initially available in FORTRAN and later transformed into stream programs. Column "*Reuse*" shows the inherent reuse in these programs: *C* stands for cross-iteration reuse, *N* for non-reuse and *I* for intra-iteration reuse. For each program, three unroll factors, denoted by *R*, *RC* and *RCM*, are listed. *RC* is directly generated by RAMS. A smaller unroll factor, denoted *RCM*, may be generated by redefining (6) as $\mathcal{F} = \text{LCM}(L'_1, \ldots, L'_m)$, where L'_r is the smallest integer no smaller than L_r such that L'_r is a factor of $\max(L_1, \ldots, L_M)$. This may cause more SRF pressure due to a higher amount of renaming generated. For *RC* and *RCM*, the same amount of concurrency is achieved. The column *MD* shows if RAMS has changed any reuse distances for SR equations or not. Finally, *R* is generated if ConMax is not allowed to modify the weights of SR-rep edges. As a result, only reuse is exploited. No attempt for improving concurrency via renaming is made.

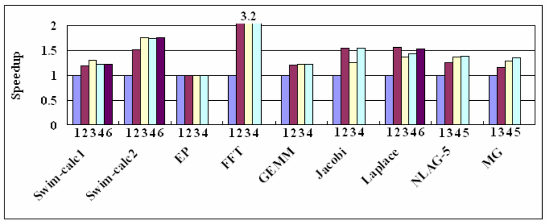


Fig. 7: Speedups of optimized over sequential loops.

Figure 7 gives the performances on FT64 under a few different unroll factors, which include those found by RAMS as a subset. By comparing the bars corresponding to the *R* unroll factors given in Table I and those with smaller unroll factors (if any), the importance of exploiting stream reuse is validated. This is particularly pronounced in `Swim-calc2`, `FFT` and `Laplace`. By comparing the bars corresponding to the *RC* and *RCM* unroll factors in Table I and those no smaller than *R*, we also observe the importance of exploiting concurrency. This is particularly significant for `Swim-calc2`, `GEMM` and `Jacobi` and noticeable for `Swim-calc1`, `NLAG-5` and `MG`. The *RC* and *RCM* performance bars also show that RAMS is successful in simultaneously exploiting reuse and concurrency. *Laplace* is the only one that does not follow the trend.

Figure 8 demonstrates the effectiveness of in reducing the number of loads and stores due to improved reuse.

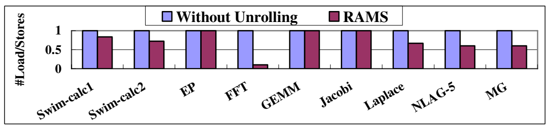


Fig. 8: Load/Store reduction achieved by RAMS.

For `FFT`, the data are kept in the SRF during computation due to reuse exploitation without incurring any off-chip memory transfers. Thus, RAMS achieves the largest speedup for `FFT`. The execution trace when $\mathcal{F} = 2$ is shown in Figure 9. The top band, "Mem Ops", represents the busy/idle status of the two memory units combined, and the bottom band, "Ker Ops", represents the busy/idle status of the stream processor. In each case, colored regions represent busy cycles. Clearly, memory transfers happen only at both ends.

`Laplace` is the only one that contradicts the performance trend expected by RAMS. This is attributed to the inaccuracy in estimating the execution times of its loads/stores.

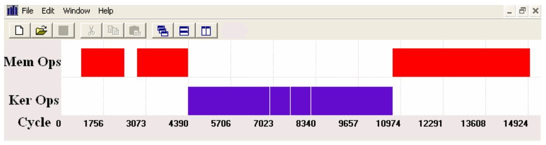

File Edit Window Help
Mem Ops
Ker Ops
Cycle 0 1756 3073 4390 5706 7023 8340 9657 10974 12291 13608 14924

Fig. 9: Execution trace of FFT ($\mathcal{F} = 2$).

B. Performance on the Imagine Stream Processor

We have also evaluated our algorithm using Imagine [6], another SVM-like stream processor. We have implemented a source-to-source translator, which takes a loop as input, applies RAMS and produces an optimized loop, which is then compiled by the Imagine compiler [3] to its binary code. We compare RAMS with their `doUnroll()` and `doSoftwarePipeline()` using their cycle-accurate simulator for Imagine. These two techniques can only be applied separately. In addition, loop unrolling is not automated and must be used with a programmer-supplied unroll factor.

Figure 10 shows the performance improvements of the nine programs achieved by RAMS over the Imagine compiler. To apply `doUnroll()`, the same unroll factor discovered by RAMS is used. As for `doSoftwarePipeline()`, no unrolling is applied. Better speedups are observed in all except `EP` and `GEMM`. For `EP`, no stream reuse exists. Due to strong loop-carried dependences, memory operations are hardly overlappable with kernel execution. Neither loop unrolling nor software pipelining is beneficial. As for `GEMM`, only intra-iteration reuse exists and it can be exploited equally by both the Imagine and our compilers.

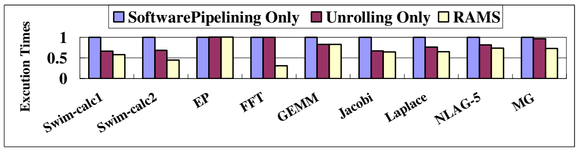


Fig. 10: Execution times of three techniques (normalized with respect to “SoftwarePipelining Only”).

VI. Related Work

The compiler developed for the Imagine stream processor [3] applies loop unrolling and software pipelining to improve the performance of stream programs. However, it does not recognize and exploit loop-carried stream reuse. Neither does it support automatic loop unrolling nor software pipelining. Furthermore, both cannot be applied together to a loop.

The work of [13] performs a data-flow analysis to recognize loop-dependent stream reuse. Our earlier work of [8] focuses on SRF allocation by using graph coloring. In particular, the work of [8] improves reuse and concurrency by requiring the programmer to experiment with different unroll factors. Our earlier work also does not address cross-iteration parallelism.

This work also differs from the prior work on improving array reuse [10], [11] for general-purpose processors in several ways. First, array reuse is exposed for a hardware-managed cache while stream reuse is for a software-managed SRF. Second, the former research examines the reuse between individual array elements while the latter considers the reuse between streams (with often thousands of elements). Third, stream copy operations are more costly than register copy operations, particularly for stream processors that do not support inter-lane communication in the SRF [13], and should thus be minimized. Finally, concurrency exploitation in stream processors requires special attention since their memory transfers and computation are decoupled.

VII. Conclusion

This paper characterizes an unrolled and software-pipelined loop of a stream-level loop using reuse equations. Based on this new notion, we have developed a new compiler approach, RAMS, that can simultaneously unroll and pipeline a stream-level loop to maximize reuse and improve concurrency in the loop. We have implemented RAMS in a stream compiler developed for our 64-bit FT64 stream processor. Validation by running nine benchmarks on FT64 as well as Imagine [6] and by simulation shows the effectiveness of the proposed approach. Our technique should be also useful to other SVM-like stream processors, such as Cell [9], Merrimac [2].

Acknowledgment

This work is supported in part by a NSF of China Grant (60621003), Two National High-Tech Research and Development Plan of China Grants (2006AA01A104, 2009AA01Z102), and a China Council Scholarship and in part by an Australian Research Council Grant (DP0881330) and a UNSW Engineering-International Research Collaboration Grant (PS16380).

References

[1] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[2] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, and J.-H. A. et al. Merrimac: Supercomputing with streams. In *SC '03*, page 35, 2003.

[3] A. Das, W. J. Dally, and P. Mattson. Compiling for stream processing. In *PACT '06*, pages 33–42, 2006.

[4] F. Labonte, P. Mattson, W. Thies, I. Buck, C. Kozyrakis, and M. Horowitz. The stream virtual machine. In *PACT '04*, pages 267–277, 2004.

[5] D. M. Lavery and W.-M. W. Hwu. Unrolling-based optimizations for modulo scheduling. In *MICRO-28*, pages 327–337, 1995.

[6] J. D. Owens, U. J. Kapasi, P. Mattson, B. Towles, B. Serebrin, S. Rixner, and W. J. Dally. Media processing applications on the imagine stream processor. In *ICCD' 02*, pages 295–302, Sept. 2002.

[7] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *MICRO-27*, pages 63–74, 1994.

[8] L. Wang, X. Yang, J. Xue, Y. Deng, X. Yan, T. Tang, and Q. H. Nguyen. Optimizing scientific application loops on stream processors. In *LCTES '08*, pages 161–170, 2008.

[9] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *CF '06*, pages 9–20, 2006.

[10] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *PLDI '91*, pages 30–44, June 1991.

[11] J. Xue and C.-H. Huang. Reuse-driven tiling for data locality. In *LCPC '97*, pages 16–33, 1997.

[12] X. Yang, X. Yan, Z. Xing, Y. Deng, J. Jiang, and Y. Zhang. A 64-bit stream processor architecture for scientific applications. In *ISCA '07*, pages 210–219, 2007.

[13] X. Yang, Y. Zhang, J. Xue, I. Rogers, G. Li, and G. Wang. Exploiting loop-dependent stream reuse for stream processors. In *PACT '08*, 2008.