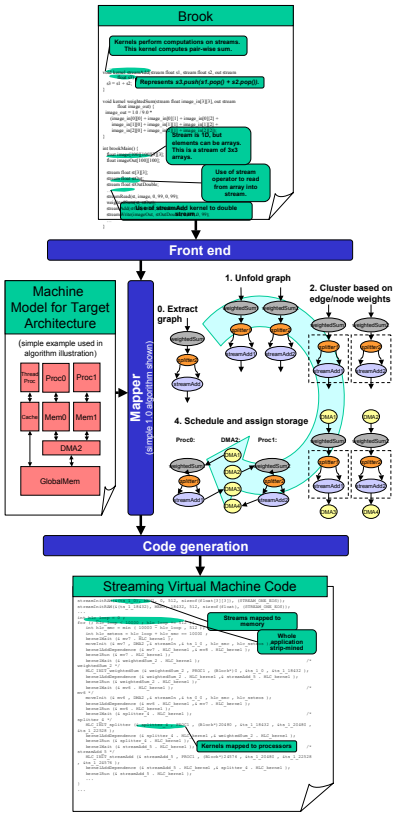
Merrimac Compilation System

Merrimac Compiler Team and Reservoir Labs Inc.

Reservoir's R-Stream Compiler

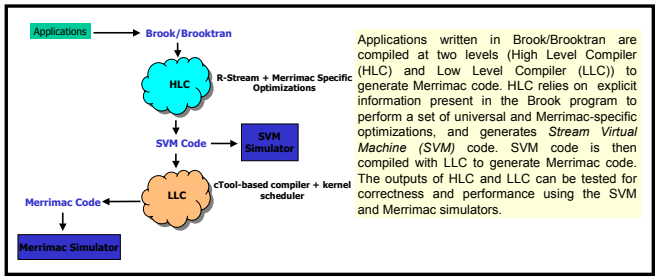


R-Stream Implementation Plan

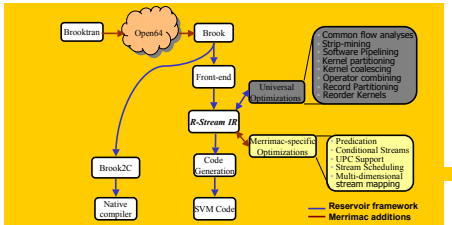
- **Three major releases:**
  - **Release 1.0: Functionality, but no performance optimization**
    - **Work with multiple low-level compilers through SVM interface**
  - **Release 2.0: Common-case, phase-ordered performance optimization**
    - **Incorporate task, data and pipeline parallelism in one framework**
  - **Release 3.0: General, unified performance optimization**
    - **Unify scheduling of computation, data and communication**

Year	03			04				05			
Quarter	2	3	4	1	2	3	4	1	2	3	4
Release				1.0			2.0			3.0	
Front-end											
Streaming IR											
Mapping											
Code generation											

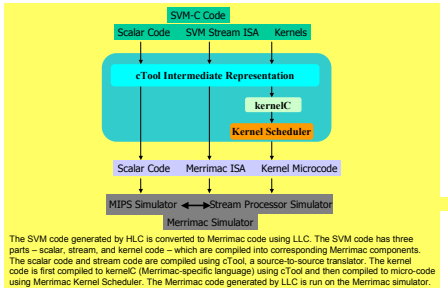
Non existent	
Discardable	
Draft	
Final	



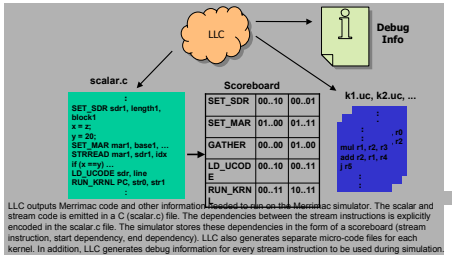
Applications written in Brook/Brooktran are compiled at two levels (High Level Compiler (HLC) and Low Level Compiler (LLC)) to generate Merrimac code. HLC relies on explicit information present in the Brook program to perform a set of universal and Merrimac-specific optimizations, and generates *Stream Virtual Machine (SVM)* code. SVM code is then compiled with LLC to generate Merrimac code. The outputs of HLC and LLC can be tested for correctness and performance using the SVM and Merrimac simulators.



HLC compiles programs written in Brook/Brooktran into SVM code. Applications written in Brooktran are translated into Brook using Open64 compiler as a source-to-source translator. Brook code is parsed and represented as R-Stream IR which is optimized using several universal and Merrimac-specific optimizations. Finally, R-Stream IR is converted into SVM code in the code generation phase. For testing the correctness of Brook code, a tool *Brook2C* can be used to compile Brook to C. The C code can then be compiled and run using a native compiler.



The SVM code generated by HLC is converted to Merrimac code using LLC. The SVM code has three parts – scalar, stream, and kernel code – which are compiled into corresponding Merrimac components. The scalar code and stream code are compiled using cTool, a source-to-source translator. The kernel code is first compiled to kernelC (Merrimac-specific language) using cTool and then compiled to micro-code using Merrimac Kernel Scheduler. The Merrimac code generated by LLC is run on the Merrimac simulator.



LLC outputs Merrimac code and other information needed to run on the Merrimac simulator. The scalar and stream code is emitted in a C (scalar.c) file. The dependencies between the stream instructions is explicitly encoded in the scalar.c file. The simulator stores these dependencies in the form of a scoreboard (stream instruction, start dependency, end dependency). LLC also generates separate micro-code files for each kernel. In addition, LLC generates debug information for every stream instruction to be used during simulation.

Current status/Future work

- High Level Compiler
  - R-Stream 1.1 compiles streamFEM, streamMD, and streamLES
  - R-Stream 2.0 under active development
  - Merrimac specific optimizations to be integrated with R-Stream 2.0
- Low-Level Compiler
  - SVM kernels compile to generate Merrimac kernel micro-code
  - SVM stream ISA compiles to Merrimac stream ISA
    - Scalar fields passed to kernels need to be handled
  - LLC generates debug information for every stream instruction dispatched to the Merrimac simulator
- HLC, LLC, and the Merrimac simulator are integrated

Illustrative example

```
int main (void) {
  int as<>; int bs<>; int cs<>;
        ...
  sum (as, bs, cs);
        ...
}
```

Brook

```
kernel void sum (int a<>, int b<>, out int c<>) {
  c = a + b;
}
```

High Level Compiler

```
streamInitRAM ( &cs_str, SRF, 65536,
  ...);
        ...
sum_INIT (&sum_1, ..., 128, as_str,
bs_str, &cs_str);
kernelRun ( (Kernel*) &sum_1);
        ...
        ...
blockInit (&(cs_blk), global_mem,
((char*)cs_addr), 128 , ...);
stridedScatterInit (
&StridedScatterDMA_1, SSS_DMA ,
cs_str, &cs_blk, 128, ...);
kernelAddDependence
(&StridedScatterDMA_1, &sum_1 );
kernelRun (&StridedScatterDMA_1 );
```

SVM-C

```
loopCounter =  ( 0 );
  while (1) {
    hlc_tmp_170 =( loopCounter )<( sum->nelem
);
    if ( hlc_tmp_170 ) {
      streamPop ( sum->i0_ , &(hlc_tmp_204 ) );
      streamPop ( sum->i1_ , &(hlc_tmp_207 ) );
      hlc_tmp_202 =( hlc_tmp_204 )+(
hlc_tmp_207 );
      streamPush ( sum->o0_ , &(hlc_tmp_202)
);
      loopCounter =( loopCounter )+( 1 );
    }
    else { break; }
  }
```

Low Level Compiler

```
ADD_DEP_S(8, 0UL, 24);
SETSDR(5, 128, 1024, 25);
...
//ASM ADD_DEP_C 9 0 484 //2 5 6 7 8
ADD_DEP_C(9, 484, 26);
RUN_KRNL(0, 0, 0, 3, 4, 5, 0, 0, 0, 0, 0, 0,
0, 27);
...
ADD_DEP_S(10, 0UL, 28);
SETSDR(6, 128, 1024, 29);
block_addr = ((int )((char *)(cs_addr)) + 0);
SETMAR(2, block_addr, 1, 4, 0, 31);
//ASM ADD_DEP_C 12 0 3616 //5 9 10 11
ADD_DEP_C(12, 3616, 32);
STRWRITE(6, 2, 33);
```

Merrimac ISA

```
kernel sum(istream<int> i0_, istream<int> i1_,
ostream<int> o0_, uc<int>& uc_nelem)
...
loopCounter = 0 + cid();
hlc_tmp_170 = loopCounter <
commclperm(ucid(),0,uc_nelem);
cc cc_63_3_hlc_tmp_170 = itocc(hlc_tmp_170);
loop_while_all (cc_63_3_hlc_tmp_170) {
  i0_ >> hlc_tmp_204;
  i1_ >> hlc_tmp_207;
  hlc_tmp_202 = hlc_tmp_204 + hlc_tmp_207;
  o0_ << hlc_tmp_202;
  loopCounter = loopCounter + ucid();
hlc_tmp_170 = loopCounter <
commclperm(ucid(),0,uc_nelem);
...
```