The Scc Compiler: SWARing at MMX and 3DNow!

Randall J. Fisher
Henry G. Dietz
School of Electrical and Computer Engineering
Purdue University

`rfisher@ecn.purdue.edu`

`http://min.ecn.purdue.edu/~rfisher/`

`hankd@ecn.purdue.edu`

`http://dynamo.ecn.purdue.edu/~hankd/`

Overview

- SWARC -- C-like language for portable SWAR coding
  - Vector objects and operators
  - Conditional constructs
  - Other modifications
- SCC -- module compiler for SWARC language
  - Source handling
  - The front end -- IR construction and optimizations
  - The back end -- fragmentation, tuple generation, emulation, and scheduling
- Preliminary performance measures
- Concluding remarks

The SWARC Language

Similar to C

First-class vector objects with virtualized length and element precision

- Default length of 1 and native C precision used if not specified
- Saturation operations may be specified for each vector; modular is default

New and modified operators for vector processing

- C operators defined for use with vectors
- C*-like reductions or expansions performed during casts to or from scalars
- New vector operators for min, max, average, shift, rotate, any, and all
- "typeof", "widthof", and "precisionof" keywords

The SWARC Language (cont'd)

Conditionals and loops modified to incorporate SIMD enable masking

- *if*
  Like C *if*, but only executes body if some enabled element is true
- *while*
  Like C *while*, but only executes body while some enabled element is true, disabling elements which become false
- *for* and *do*
  Related to *while* in the same way that C's *for* and *do* are related to its *while* New control constructs
- *where* and *elsewhere*
  - Similar to *if* and *else*, but bodies are always executed
  - Enable masking used to limit the effects of each body to the correct set of vector elements

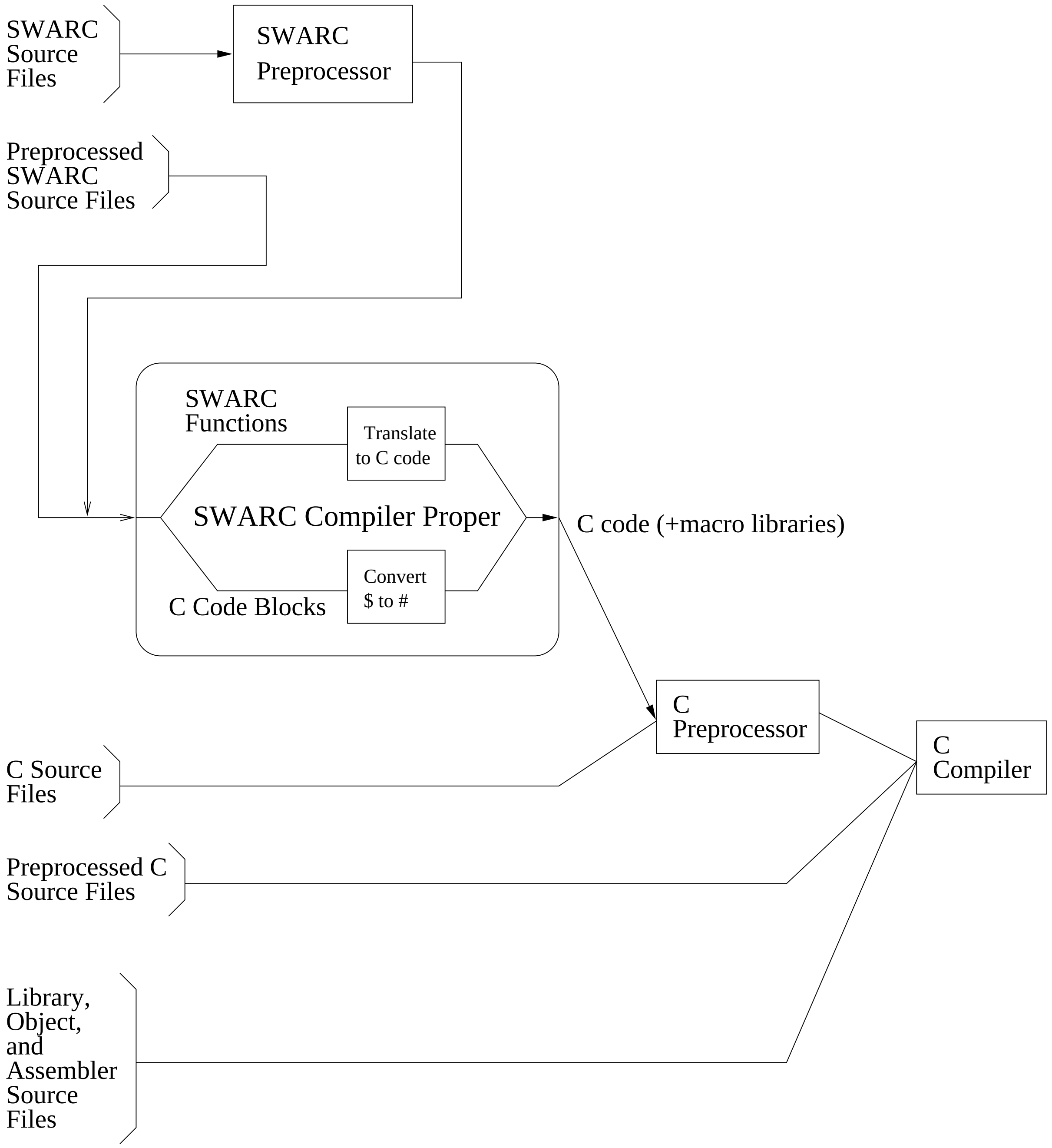
The SWARC Language (cont'd)

Other changes:

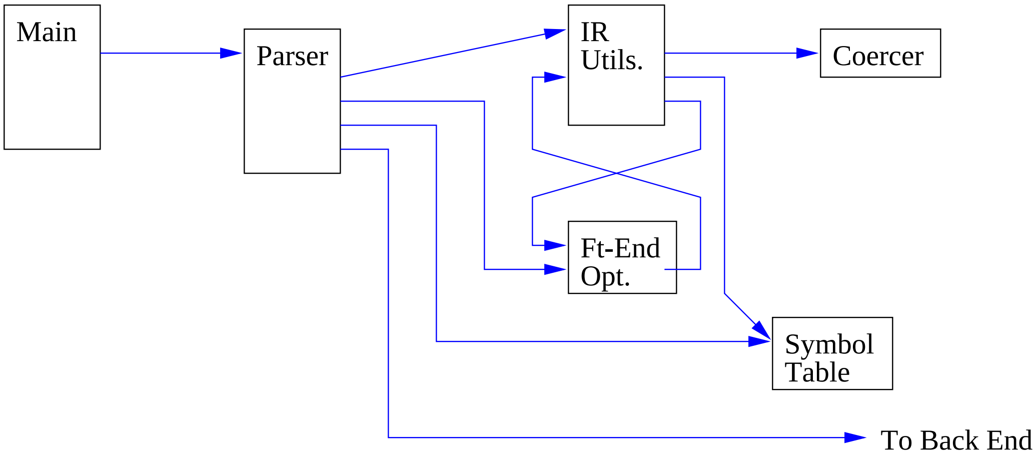
- C code contained in *${ $}* pair is emitted to C output
- *break* and *continue*
  Each takes an optional argument indicating how many levels to break from
- Function arguments passed by address to avoid excessive data copying
- All functions return void to simplify interface with C code

The Scc Experimental SWARC Compiler

- SWARC sources translated to C modules and passed to C compiler
  Modules may include macros for inlined code which accesses extended instructions
- C sources passed to C compiler
- Libraries, objects, and assembler sources passed to C compiler
- GCC compatible C compiler used to compile/link above


SWARC Source Files
SWARC Preprocessor
Preprocessed SWARC Source Files
SWARC Functions
Translate to C code
SWARC Compiler Proper
C code (+macro libraries)
Convert $ to #
C Code Blocks
C Preprocessor
C Compiler
C Source Files
Preprocessed C Source Files
Library, Object, and Assembler Source Files

Scc Front End

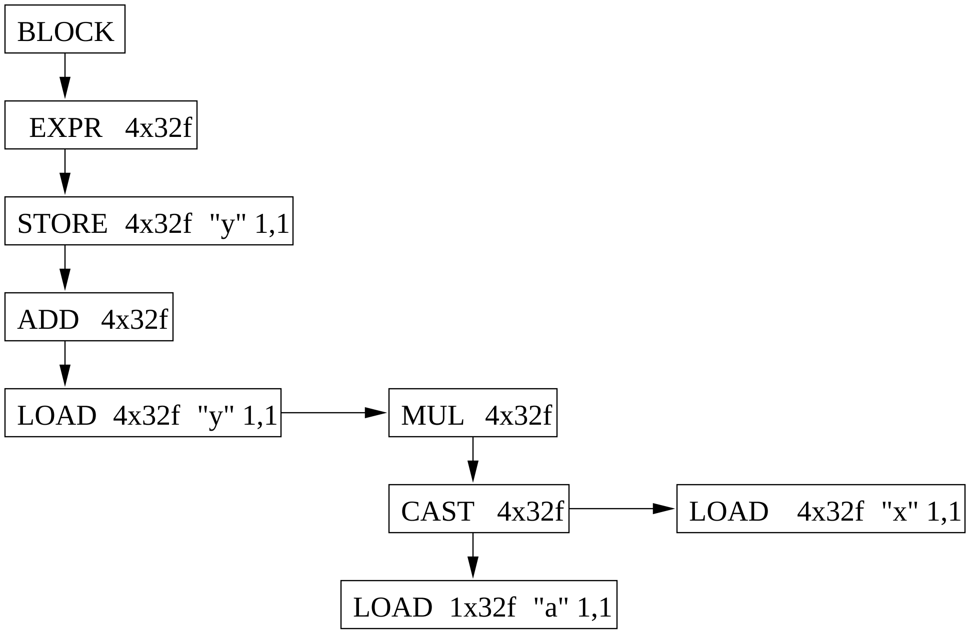


- SWARC source code converted into IR tree
- IR nodes represent scalar and vector operations, data, and control structures
- Constants stored in 128-bit wide "constant pool" for later output
- Vector-based optimizations performed on IR tree before it is passed to back end
- Control construct frameworks emitted directly to output (this will be changed soon)

Scc Front End (cont'd)

Ex: Modified SAXPY

```
void swar_saxpy(float:[4] x, float:[4] y, float a)
{
    y += a*x;
}
```



Scc Front End Optimizations

Promotion of data precision

- Used when a larger precision can be handled more efficiently with minimal loss of parallelism.
- Visible effects such as saturation to the actual precision must be maintained (Unfortunately, Scc does not currently do this)

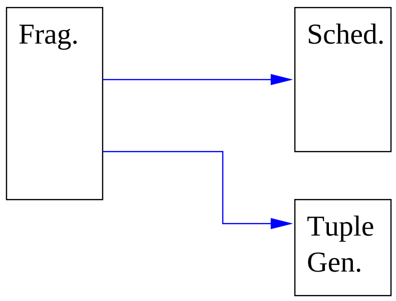
Algebraic simplification of associative operations

- Generalization of Bitwise Value Tracking as discussed last year.
- Example:

```
( ( x + C1 ) * C2 ) + C3
( ( x * C2 ) + ( C1 * C2 ) ) + C3
( x * C2 ) + ( C1 * C2 + C3 )
( x * C2 ) + Cx, where Cx = C1 * C2 + C3
```

- Particularly useful for combining masking operations

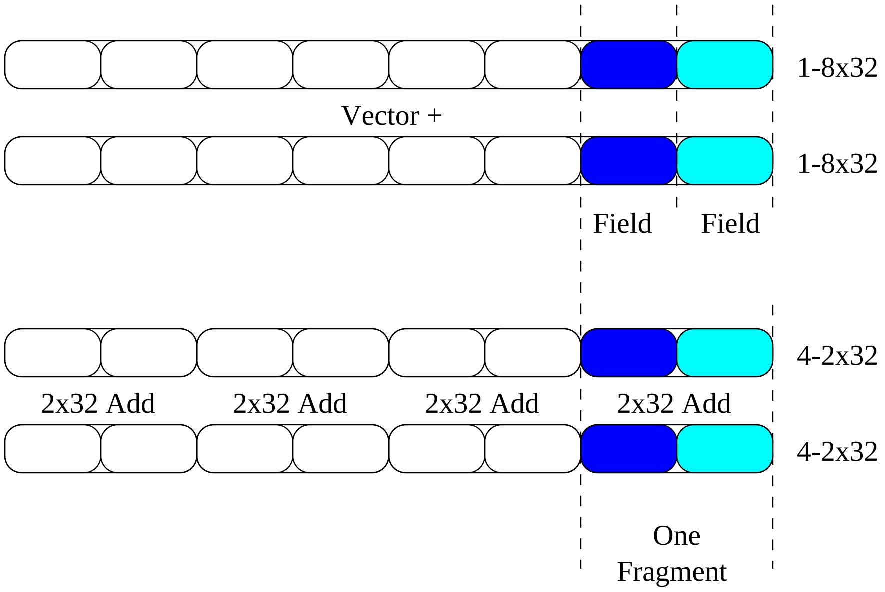
Scc Back End



- Vector operations are divided into operations on word-length sections by the *fragmenter*
- The *tuple generator* builds trees of tuples representing HW instructions for each section
- *Scheduler* schedules tuples, allocates registers, and handles spills

Fragmentation in Scc

- B-bit register can hold up to f=floor(B/b) b-bit *fields*
- Field layout hidden from programmer
- N element vector broken into ceiling(N/f) *fragments*
- Extra fields in a fragment are not treated *specially* unless the effect would be visible
- Example: Fragmentation of a Vector Addition



- Fragment operations unrolled; not strip-mined

Tuple Tree Generation

Emulation of vector operations

- Allows wide range of targets
- Must be done well to provide speedup
- Methods:
  - Direct Emulation
  - Modification of similar operations
  - Temporary promotion to a size supported by hardware instructions

Hardware SWAR instructions used to optimize emulation when available

Direct Emulation Example: 2 bit signed saturated ADD

```
/* High bit = a~cd | ~abc */
if (optcpu & CPU_MMX) {
     i = binop(ANDN, arg1, arg0, typnull);
} else {
     i = unop(NOT, arg1, typnull);
     i = binop(AND, i, arg0, typnull);
}
j = binop(SHL, arg1, immed64u(1ULL), typnull);
i = binop(AND, i, j, typnull);

if (optcpu & CPU_MMX) {
     j = binop(ANDN, arg0, arg1, typnull);
} else {
     j = unop(NOT, arg0, typnull);
     j = binop(AND, j, arg1, typnull);
}
k = binop(SHL, arg0, immed64u(1ULL), typnull);
j = binop(AND, j, k, typnull);

/* High bit */
i = binop(OR, i, j, typnull);
i = binop(AND, i, immedu(cvt1x8uto16x8u(0xaa)), typnull);

/* Low bit  = bd | ac */
j = binop(AND, arg0, arg1, typnull);
k = binop(SHR, j, immed64u(1ULL), typnull);
j = binop(OR, j, k, typnull);
j = binop(AND, j, immedu(cvt1x8uto16x8u(0x55)), typnull);

return (binop(OR, i, j, typnull));
```

Example of Modification of a Similar Operation: 4-bit signed saturated ADD

```
/* Do the signed add */
i = binop4(ADD, arg0, arg1);

/* Correct for positive saturation */
/* Clobber the calculated value with PosSat if arg1==0 */
j = binop(OR, arg0, arg1, typnull);
j = unop(NOT, j, typnull);
j = binop(AND, j, i, typnull);
j = binop(AND, j, immedu(cvt1x8uto16x8u(0x88)), typnull);
k = binop(SHR, j, immed64u((p64_t) 1ULL), typnull);
l = binop(OR, k, j, typnull);
k = binop(SHR, l, immed64u((p64_t) 2ULL), typnull);
l = binop(OR, k, l, typnull);
i = binop(OR, i, l, typnull);
j = unop(NOT, j, typnull);
i = binop(AND, i, j, typnull);

/* Correct for negative saturation */
/* Clobber the calculated value with NegSat if arg1==0 */
j = binop(AND, arg0, arg1, typnull);
k = unop(NOT, i, typnull);
j = binop(AND, j, k, typnull);
j = binop(AND, j, immedu(cvt1x8uto16x8u(0x88)), typnull);
k = binop(SHR, j, immed64u((p64_t) 1ULL), typnull);
l = binop(OR, k, j, typnull);
k = binop(SHR, l, immed64u((p64_t) 2ULL), typnull);
l = binop(OR, k, l, typnull);
l = unop(NOT, l, typnull);
i = binop(AND, i, l, typnull);
i = binop(OR, i, j, typnull);

return(i);
```

Example of Temporary Promotion to a HW Supported Precision: 4-bit unsigned modular MUL

```
/* use 8-bit multiplies */
i = binop(MUL, arg0, arg1, typ8u);
i = binop(AND, i, immedu(cvt1x8uto16x8u(0x0f)), typnull);

j = binop(SHR, arg0, immed64u((p64_t) 4ULL), typ8u);
k = binop(SHR, arg1, immed64u((p64_t) 4ULL), typ8u);
j = binop(MUL, j, k, typ8u);
j = binop(AND, j, immedu(cvt1x8uto16x8u(0x0f)), typnull);

j = binop(SHL, j, immed64u((p64_t) 4ULL), typ8u);
return(binop(OR, i, j, typnull));
```

Scheduling

Several iterations, increasing number of allowed spills until a viable schedule is found.

Each iteration...

- Attempts to generate a starting schedule built backward from live stores
- Then performs a full-width search of the possible tuple orderings to find the best based on pipeline cost estimate
- If a schedule can be found, it is known to have optimal estimated cost given the above restrictions

Note: Current pipeline model overestimates average memory access time

Why go through all this?

- Maximum parallelism is small (no greater than N/b for b-bit fields in an N-bit register)
- Easy to lose potential speedup to several pitfalls:
  - Data packing overhead
    - Many targets do not support memory access well
  - Operation emulation overhead
    - ALL targets require some emulation of virtualized vector arithmetic
  - Poor emulation methods
    - Some are too expensive to pay off
  - Control overhead for special registers or operating modes
    - example: MMX's emms
  - Emulation of missing critical SIMD constructs
    - example: many targets do not support parallel conditional tests
- Cannot afford to be sloppy about local code quality
- Scc goes to extreme lengths to minimize overhead losses and to maximize and protect speedup gains

Preliminary Performance Measures

- Students achieved greater than 5x speedup over serial C code for subpixel rendering using Scc-generated MMX code
- 80 MFLOPS for SWARC-modified Linpack on 32-bit floating-point vs 54 for serial C version
- 220 MFLOPS achieved after hand-correcting schedule for cost estimation error

Conclusions

- SWARC provides a usable HLL for SWAR programming
- Emulation of operations allows portability to many targets
- SWAR extensions can be used to optimize emulated code
- The public-domain Scc research compiler targets MMX and 3DNow!, and will soon target 32- and 64-bit standard integer registers
- More information and working compiler available at
  http://shay.ecn.purdue.edu/˜swar
- First full source code Scc release no later than SC99