A Vectorizing SUIF Compiler: Implementation and Performance

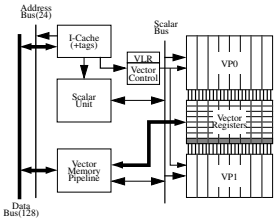
Corinna G. Lee
Department of Electrical & Computer Engineering
University of Toronto
http://www.eecg.toronto.edu/~corinna

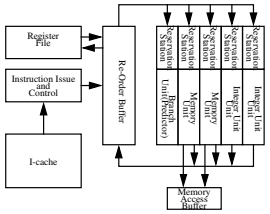
Overview

The Compiler
- Overview
- Pre-Vectorization
- Vectorization
- Post-Vectorization
- Code Generation

Evaluation and Performance
- ANL Benchmarks
- Performance Study
  - Overview
  - Data

Vector and Superscalar Architectures





T0/SPERT-II:
- 45(40)MHz
- 4GB(8MB) SRAM
- 1 16b,Fix-point mult.
- var. latency memory

Execution Styles

```
for(i=0;i<96;i++)
  a[i]=b[i]+c[i];
```

superscalar

- 96 iterations
- 768 instructions
- 3/1 cycles/iter
- 288/96 cycles

vector

- 3 iterations
- 24 instructions
- ~0.4 cycles/iter (8 pipes)
- 39 cycles

superscalar

```
r1 = a
r2 = b
r3 = c
r7 = a+96

TOP:
r4 = *r1
r5 = *r2
r6 = r4 + r5
*r3 = r6
r1 = r1+1
r2 = r2+1
r3 = r3+1
bra TOP r1 != r7
```

vector

```
r1 = a
r2 = b
r3 = c
r7 = a+96
VLR = 32
TOP:
v1 = *r1,1
v2 = *r2,1
v3 = v1 + v2
*r3 = v3,1
r1 = r1 + 32
r2 = r2 + 32
r3 = r3 + 32
bra TOP r1 != r7
```

Vector Compilers

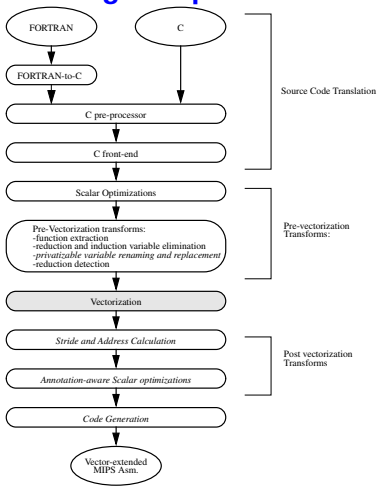
A vectorizing Compiler must:

- locate vectorizable constructs
- transform code to fit on available hardware
  - stripmining
- transform code to preserve dependencies
  - statement reordering

A vector-language compiler must:

- transform code to fit on available hardware
  - each statement stripmined separately
- transform code to reduce scalar overhead
  - loop alignment
  - loop fusion

The SUIF Vectorizing Compiler: Overview



Pre-Vectorization

Goals:

- to reduce the number of scalar expressions in the inner loop
- perform bookkeeping to expand the kinds of program constructs to which vectorization may be applied.
- transforms to expose data parallelism

Methods:

- scalar reduction elimination
- function extraction
- induction variable elimination
- array-reduction detection
- scalar privatization and renaming

Scalar Privatization

A scalar is privatizable if:

- first defined within the loop
- all uses are within the loop
- live range does not cross loop iterations
  - “carry around” variable not included

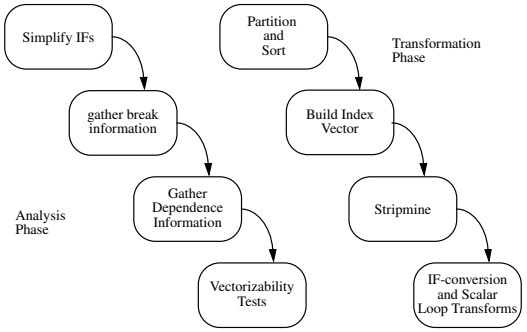
(Privatized) Scalar renaming

- each live range of a variable gets a unique name
- removes false anti-dependencies

```
for (i=0;i<N;i++){
  s=a[i]+b[i]+i;
  if (s<1)
    s=1;
  top[i]= bottom[i-1]+s;
  bottom[i]=top[i]
  }
```

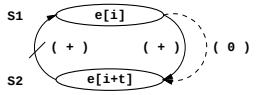
```
for (i=0;i<N;i++){
  private_s[i]=a[i]+b[i]+i;
  if (private_s[i]<1)
    private_s[i]=1;
  top[i]=bottom[i1]+private_s[i];
  bottom[i]=top[i]
  }
```

Vectorization

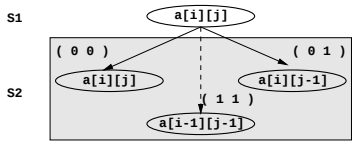


Dependence Graphs

```
for(i=0;i<100;i++)
    {
S1:  e[i]=f[i]+10;
S2:  g[i]=e[i+t]+h[i];
    }
```




```
for(i=0;i<100;i++)
   for(j=0;j<100;j++)
      {
S1:  a[i][j]=b[i][j]+c[i][j];
S2:  d[i][j]=a[i][j]+
                a[i-1][j-1] +
                a[i][j-1];
     }
```



Conditions for Vectorizability

A statement is vectorizable if:

- all references are to:
  - loop invariant scalars
  - the loop index variable
  - other variables which access different locations in each iteration of the loop
- it does not participate in a dependence cycle
  - the statement does not produce a result used by itself or by another statement on which it depends (data and control)
  - Use Tarjan's algorithm
    - recursive descent of dependence graph at statement level
    - reports cycle members and vectorizable statements
    - no node-splitting

Transformations

Partitioning:

- assume 3 sections in loop
  - pre-scalar, scalar, post-scalar

Statement reordering:

- both vector clauses must be sorted separately
- live range reduction

Stripmining

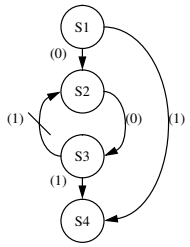
- any loop may be stripmined without altering execution order

Index Vector

- addition of index vector initialization loop if required

Vectorization: example

```
   for(i=0;i<100;i++)
      {
S1:   a[i]=b[i]+c[i];
S2:   d[i]=a[i]/e[i];
S3:   e[i+1]=(d[i]+d[i+1])/2;
S4:   f[i]=e[i]+a[i];
      }
```




```
      iterations3 = 100-0;
      VL2=MIN(iterations,32);
      set_vl_reg(VL2);
      for(Vindx1=0;Vindx1<100;Vindx1+=32){

S1:     a[Vindx1] = b[Vindx1] + c[Vindx1];
                    ANNOTE:vectorize

        for(i=Vindx1; i< Vindx1+VL2; i++) {
S2:       d[i] = a[i] / e[i];
S3:       e[i + 1] = (d[i] + d[i + 1]) / 2;
          }

S4:     f[Vindx1] = e[Vindx1] + a[Vindx1];
                    ANNOTE:vectorize

        iterations3 = iterations3 - 32;
        VL2=MIN(iterations,32);
        set_vl_reg(VL2);
}
```


Structured Control Flow

Results in different execution path on each iteration

- V-procs perform VL$_{MAX}$ iterations “simultaneously”

Solutions:

- don’t vectorize control flow
  - disqualifies many loops
- perform IF-conversion and use a predicated execution model
  - use of predicate architecture dependent but creation is not
  - applicable to IF and SWITCH control
  - one predicate/clause->expensive for switch
  - current implementation only works with IF-based control flow


```
   for (i=0;i<100;i++) {
S1:if(a[i]==0){
S2:  if(b[i]==1)
S3:     c[i]=6;
     else
S4:     c[i]=7;
     }
   else
S5:  c[i]=8;
   }
```

```
      iterations10 = 100;
      VL9 = 32;
      set_vl_reg(VL9);
      for(Vindx8=0;Vindx8<100;Vindx8+=32){
S1:     cv_then12[Vindx8]=a[Vindx8] == 0;
                    ANNOTE:vectorize
S1a:    cv_else13[Vindx8]=1&~cv_then12[Vindx8];
                    ANNOTE:vectorize
S2:     cv_then14[Vindx8]=b[Vindx8]==
                    1&cv_then12[Vindx8];
                    ANNOTE:vectorize
S2a:    cv_else15[Vindx8]=1&~cv_then14[Vindx8]&
                    cv_then12[Vindx8];
                    ANNOTE:vectorize
S3:     c[Vindx8] = 6;
                    ANNOTE:vectorize
                    ANNOTE:cv_then14
S4:     c[Vindx8] = 7;
                    ANNOTE:vectorize
                    ANNOTE:cv_else15
S5:     c[Vindx8] = 8;
                    ANNOTE:vectorize
                    ANNOTE:cv_else13
        iterations10 = iterations10 - 32;
        if (iterations10 < 32){
          VL9 = iterations10;
          set_vl_reg(VL9);
          }
    }
```


Functions and BREAKs

Functions:

- may change variables containing the loop state
  - assume functions do not change loop state, but only move data into or out of the loop
  - run all function calls in the scalar body and ensure all data dependencies are satisfied

BREAKs:

- cause premature exit from the loop
  - execute all statements before the scalar body with the full VL,
    - buffer results!
  - determine break iteration in scalar body
  - commit speculative results/execute post-scalar code

Problems: expensive to buffer speculated results

Post-Vectorization

- High level constructs useful for analysis, but we need to map to a real machine
- Array instructions (which abstract out all index calculations) must be converted into address calculations
  - loss of information
- Must determine stride of memory accesses at this time while high level info is still available
  - stride value calculation is exposed but its uses are hidden (limitation of conventional SUIF)
  - changes to dead code eliminator and propagation passes needed to preserve correctness and efficiency

```
   for(i=0;i<100;i++)
     {
S1:  a[i]=a[2*i+5]*b[s*i];
     }
```

```
   VL2=100;
   set_vl_reg(VL2);
   for(Vindx1=0;Vindx1<100;Vindx1+=32){
S1:  a[Vindx1]=a[2*Vindx1+5] *
                b[s*Vindx1];
                  ANNOTE:vectorize
     iterations3 = iterations3 - 32;
     if(iterations3 < 32){
       VL2 = iterations3;
       set_vl_reg(VL2);
       }
   }
```

```
   set_vl_reg(VL2);
   ivar1 = a;
   ivar1step6 = 4;
   ivar2 = a + 20;
   ivar2step7 = 8;
   ivar3 = b;
   ivar3step8 = s * 4;
   for(Vindx1=0; Vindx1<100; Vindx1+=32){
S1:  *ivar1 = *ivar2 * *ivar3;
          ANNOTE:vectorize
          ANNOTE:ivar1 → ivar1step6
          ANNOTE:ivar2 → ivar1step7
          ANNOTE:ivar3 → ivar1step8
     iterations3 = iterations3 - 32;
     if(iterations3 < 32) {
       VL2 = iterations3;
       set_vl_reg(VL2);
       }
     ivar1 = (int *)(ivar1 + 128);
     ivar2 = (int *)(ivar2 + 256);
     ivar3 = (int *)(ivar3 + s * 128);
   }
```

Code Generation

Each vectorizable expression tree maps to a set of instructions

- scalar instructions prepare any scalar operands required (e.g.: base addresses, strides, bit masks)
- vector instructions perform the statements work for strip
- 1:1 map for RISC machines

Code Generator handles:

- emulation of non-supported instruction types
  - 32b integer multiply, complement...
- reductions
  - recursive doubling algorithm
- use of condition vector
- simple register allocation

Code Generation

Condition Vectors

- T0 supports conditional vector register moves
- vector3 = vector1 + vector2 → 4 memory ops
  - load vector1 and vector2, then add them to vtemp1
  - load vector3 to vtemp2
  - conditionally merge vtemp1 and vtemp2
  - store merged result to vector3
- can be costly

Register Allocation

- “hints” in code indicate live ranges of arrays
- allocate on definition, spill if required
- re-allocate on use (if loaded from memory)
- privatized and condition vectors only
- removes all memory ops from conditionals in some cases

ANL Benchmarks

- Published by Argonne National Laboratory in '88, '92
- Taken from linear algebra kernels and supplied by H/W vendors
- Test for existence of many transforms with a compiler

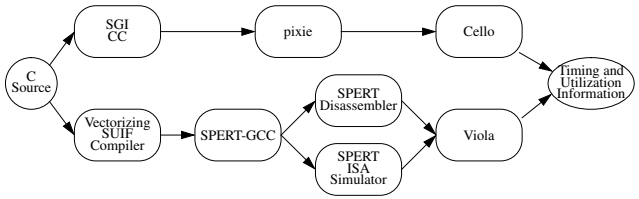
Category	Number of Loops	3090-600J		C210		CRAY-2		SUIF	
		full	partial	full	partial	full	partial	full	partial
Dependence Analysis	29	25	0	19	5	20	0	9(12)	0
Vectorization	41	31	5	30	6	14	2	14(15)	4
Idiom Recognition	24	14	0	16	1	17	0	8	1

	loop category	loops	3090-600J		C210		CRAY-2		SUIF	
			full	partial	full	partial	full	partial	full	partial
Dependence Analysis	linear dependence testing	8	8	0	8	0	5	0	3 (5)	0
	induction variable recognition	8	6	0	5	3	5	0	3	0
	global data flow analysis	2	2	0	2	0	2	0	1(2)	0
	nonlinear dependence testing	1	0	0	0	0	0	0	0	0
	interprocedural data flow analysis	2	2	0	1	0	1	0	0	0
	control flow	2(1)	1	0	0	2	1	0	0	0
	symbolics	6	6	0	3	0	6	0	2	0
	Section Totals	29	25	0	19	5	20	0	9 (12)	0
Vectorization	statement reordering	2	2	0	2	0	0	0	2	0
	loop distribution	2	0	2	0	2	0	0	0	2
	loop interchange	5(4)	3	1	3	0	0	1	0	0
	node splitting	4	3	1	3	1	0	0	0	0
	scalar and array expansion	8	6	1	6	1	3	0	2	1
	scalar renaming	1	1	0	1	0	0	0	1	0
	control flow	12 (9)	11	0	11	1	10	0	8	0
	crossing thresholds	1	1	0	0	1	0	0	0	1
	loop peeling	3	3	0	2	0	0	0	0	0
	diagonals	2	1	0	2	0	1	1	1(2)	0
	wavefronts	1	0	0	0	0	0	0	0	0
	Section Totals	41	31	5	30	6	14	2	14 (15)	4
Idiom	reductions	13	8	0	9	0	9	0	5	1
	recurrences	3	0	0	0	1	0	0	0	0
	search loops	2(1)	1	0	1	0	2	0	0	0
	packing	3	2	0	3	0	3	0	0	0
	loop rerolling	3	3	0	3	0	3	0	3	0
	Section Totals	24	14	0	16	1	17	0	8	1
Language Completeness	loop recognition	5	4	0	0	1	3	0	0	0
	storage class equivalence	4	2	1	3	0	4	0	3	0
	parameters	2	2	0	2	0	2	0	2	0
	non-logical if's	3(0)	3	0	2	0	1	0	0	0
	intrinsic functions	3	3	0	3	0	3	0	2	1
	call statements	1	1	0	0	0	0	0	0	1
	non-local gotos	2(0)	2	0	0	0	2	0	0	0
	indirect addressing	7	7	0	7	0	7	0	7	0
	statement functions	1	1	0	1	0	1	0	1	0
	Section Totals	28	25	1	18	1	23	0	15	2
	TOTALS	122	95	6	83	13	74	2	56 (60)	7

Performance: Overview

Compare vector with superscalar execution

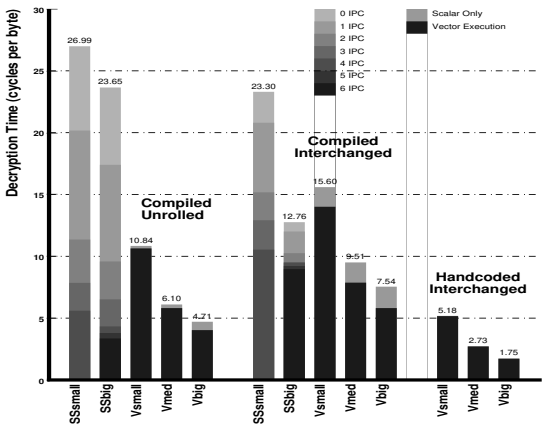
- multiple hardware configurations
- ideal memory model
- instruction and data loads do not interfere



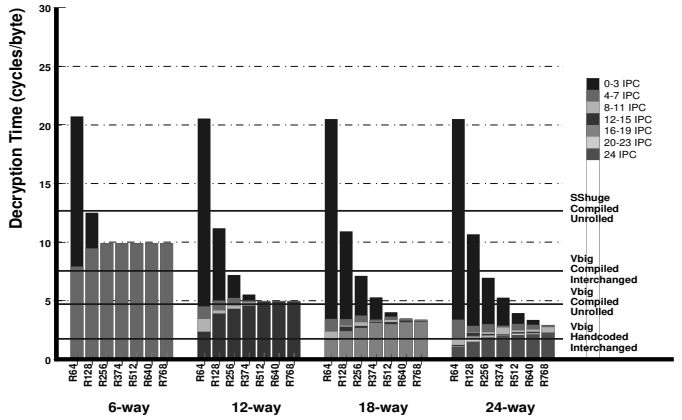
Simulation Parameters

Component	Superscalar Processors (based on MIPS R10000)			Vector Processors (based on Torrent-0)		
	SSsmall	SSbig	SShuge	Vsmall	Vmed	Vbig
ISA	MIPS			MIPS with fixed-point vector extensions		
fetch width issue width re-order buffer size	16 4 48	16 6 128	128 24 768	1 1 ——		
#physical registers	64 integer 64 FP (128 total)	160 integer 160 FP (320 total)	800 integer 800 FP (1600 total)	32 scalar 16 32-element vector (544 total)		
#integer units #load/store units	4 2	16 8	16 8	4 2	8 4	16 8
memory system	ideal			ideal		
C compiler	SGI –O2 (Version 5.3)			vector-enhanced SUIF (Version 1.1.0)		

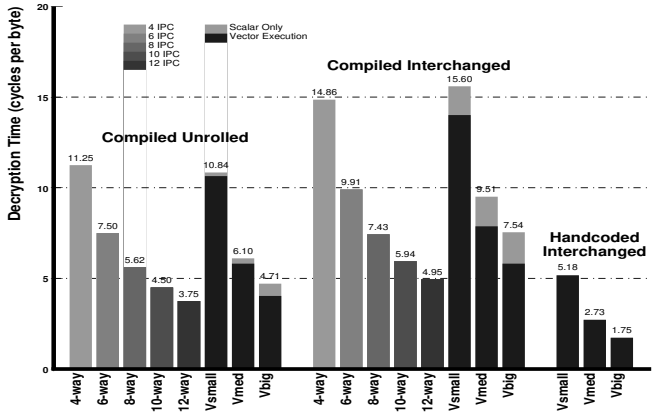
Performance of PGP Decryption



Unrealistic PGP Superscalar Performance



Lower Bounds on Superscalar Decryption Times



Conclusions and Contributions

Good starts toward a highly effective vectorizing compiler

Vector parallelism available in workstation class applications

Contributions

- high-level transformation pass for exposing vector parallelism
- T0 code-generator
- preliminary performance study

Future Work

- conditional vectorization
- additional loop transforms indicated by ANL
- machine dependent optimizations
  - scheduling and pointer update elimination
- unit-stride transforms
- other architectures
  - multimedia instruction sets
- new benchmarks