Initial Results on the Performance and Cost of Vector Microprocessors

Corinna G. Lee and Derek J. DeVries
University of Toronto

What to Do With 1G Transistors? (Sept '97 Issue of IEEE Computer)

- evolutionary to revolutionary proposals
  - very wide superscalar, superspeculative, trace processors, vector IRAM, chip multiprocessors
- most approaches attempt to exploit parallelism to improve performance
- we're investigating the viability of single-chip vector microprocessors
  - achieve excellent performance on FP programs
  - but have reputation for being costly to implement
- talk presents initial results of investigation

Overview of Presentation

Vector & Superscalar Processor Configurations

Performance of a Highly Parallel Integer Loop

Implementation Cost of Processor Configurations

Summary & Conclusions

Vector Microprocessors

- based on the Torrent-0 (T0) developed at Berkeley:
  - vector extension of MIPS ISA
  - implementation is a highly parallel vector microprocessor
  - support for integer and fixed-point arithmetic only
  - fully functional chip in April 1995
- T0-specific features:
  - T0 vector instruction executes up to 32 operations
  - 2 vector units with 8 integer units each
  - 128-bit data bus capable of 4 32-bit accesses/cycle
- with appropriate mix of instructions, T0 can execute up to 21 operations per cycle

General Vector Features

- simple instruction fetch, issue, and decode
- large capacity, high bandwidth register file
  - important feature when executing many operations in parallel
- flexible chaining
- scalar CPU to execute scalar instructions

Torrent-0 Vector Register File

- large capacity:
  - 16 32-element 32-bit vector registers
  - ⇒ 512 registers or 2KB capacity
- high bandwidth:
  - each vector register has 5 read ports & 3 write ports
  - entire vector register file is able to do 40 read and 24 writes in one clock cycle

Vector Processor Configurations

processors vary in number of functional units:

- 2 vector units with N integer units each
- up to N memory accesses/cycle

Vector Processor	N	Max Operations Executed Per Cycle (3N+1)
Vsmall	2	7
Vmed	4	13
Vbig	8	25

Processor Configurations

Component	SSsmall	SSbig	Vsmall	Vmed	Vbig
Fetch width	16	16	1		
Issue width	4	6	1		
ROB size	48	128	---		
# physical registers	64 int 64 FP (128 tot)	160 int 160 FP (320 tot)	32 scalar 16 32-element vector (544 total)		
# int units	4	16	4	8	16
# L/S units	2	8	2	4	8

superscalar processors: based on MIPS R10000

vector processors: extended MIPS ISA, N=#L/S units

Memory Configuration

- for this study, ideal memory
- more recent experiments simulate realistic memory hierarchy based on Alpha 21264
- performance results do not change significantly
- conclusions drawn from this study still hold

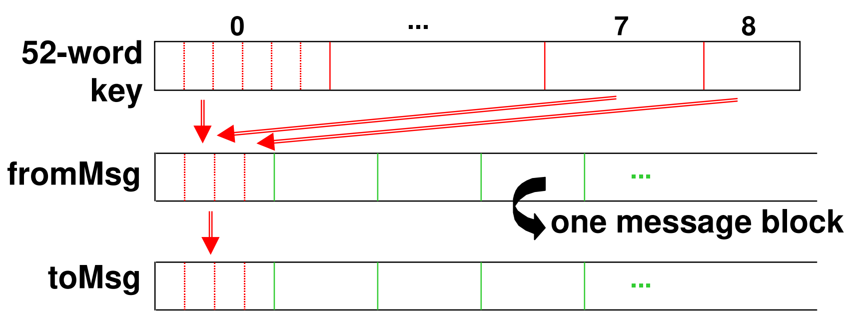
PGP Encryption/Decryption

- algorithm is a double-nested loop
- inner loop:
  - executed a fixed number of times (8)
  - contains a dependence cycle

  ⇒ don't expect to find much parallelism in inner loop
- outer loop:
  - number of iterations depends on length of message
  - no dependence cycles

  ⇒ iterations can be executed in parallel
- same algorithm for encrypting/decrypting but use different keys
  - use "decryption" for remainder of talk

PGP Encryption/Decryption



same algorithm for encrypting/decrypting but use different keys

⇒ use "decryption" for remainder of talk

Expected Parallelism in PGP Decryption

- applying i^{th} portion of key to a message block is data dependent on applying previous portions of key to that message block

  ⇒ don't expect to find much parallelism in code that decrypts a single message block
- encrypting a message block is independent of encrypting any other message block

  ⇒ expect to find lots of parallelism in code that decrypts N message blocks, especially for large N

Schematic Code for PGP Algorithm

```
cipher (fromMsg, toMsg, key) {
  for (cnt=0; cnt<MsgLength; cnt += 4) {
    // Encrypt message in 4-word blocks.
    x0 = fromMsg[cnt]
          :
    for (r=0; r<8; r++) {
      // Apply 6 words of key to 4 words of message
      x0 = f(x0,x1,x2,x3,key[r])
          :
    }
    toMsg[cnt] = g(x0,key[8])
          :
  }
}
```

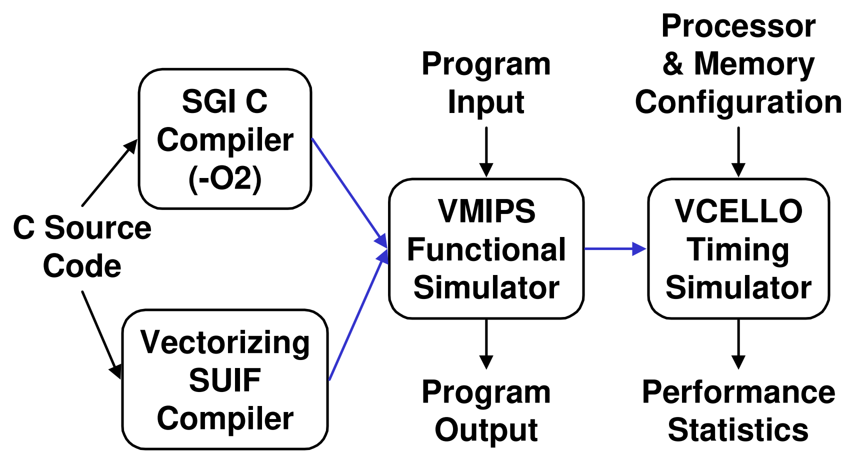
Different Code Versions of Loop

- ✗ as is with double-nested loop
- ✓ unroll inner loop completely
- ✓ apply loop transformations to interchange loop
  - ¤ privatize scalar variables x0, x1, x2, x3 and expand into temporary arrays x0[MsgLength/4], ….
  - ¤ distribute (or split) contents of outer loop into three separate loops
  - ¤ interchange perfectly nested double loop
- ✓ handcoded vector version of interchanged loop
  - ¤ temporary arrays are allocated to vector registers rather than memory

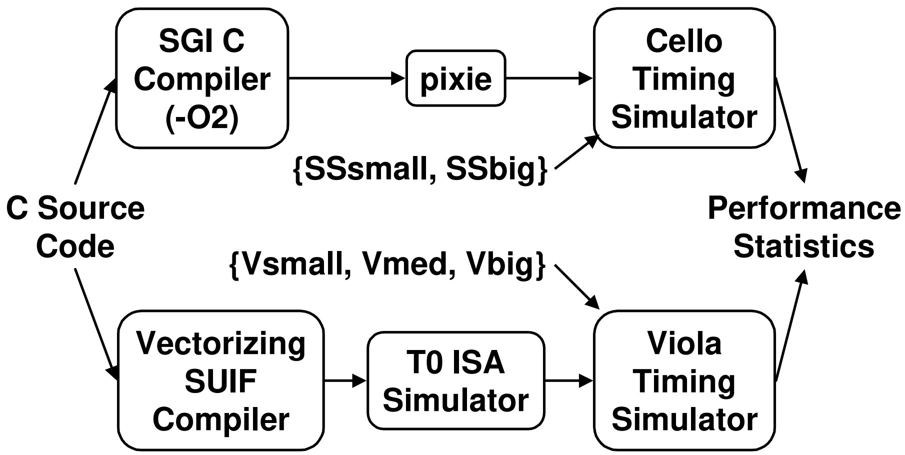
Schematic Code for Interchanged Version

```
cipher (fromMsg, toMsg, key)
{
  for (cnt) { x0[cnt] = fromMsg[cnt]; … }

  for (r=0; r<8; r++) {
    for (cnt) {
      x0[cnt] = f(x0[cnt],x1[cnt],…,key[r])
          :
    }
  }

  for (cnt) { toMsg[cnt] = g(x0[cnt],key[8]); … }
}
```

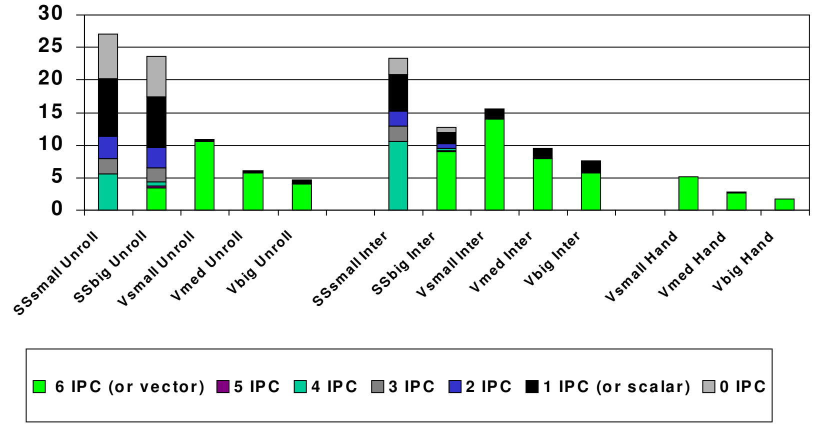
Simulation Framework for Evaluating Code Performance



Experimental Framework



PGP Performance (Cycles per Bit)



Analysis of PGP Performance

- vectors exploit parallelism more effectively
  - vector processor executes highly parallel loop 1.5 to 7.3 times faster than comparably configured superscalar processor
- superscalars are affected by how loop is coded
  - performance impact is less for “small” superscalars
  - greater with wider, more parallel superscalar processors
- more loop-level parallelism than ILP
- greater performance possible from vector processor with more sophisticated compiler
  - handwritten version is 2-3 times faster than compiled versions

Performance Impact of Vector Instructions

- vector instructions
  - exploit parallelism across loop iterations
  - implicitly re-schedule operations and temporally localize parallelism
  - reduce instruction bandwidth by more than a factor of 10
- results of other performance experiments in paper
  - demonstrate performance impact of vector instructions

PGP Performance on Unrealistic Superscalar Configurations for Interchanged Version

Lower Bounds on Superscalar Decryption Time

Effects of Software Pipelining

compiler options:

-O3 -SWP:=ON -SWP:body_ins_count=0 -OPT:alias=restrict
-mips4 -n32

Code Version	SWP Result (from asm comments)
Original	Not pipelined due to recurrence with 96 instructions 69 cycles per key iteration
Unrolled	Not pipelined because exceeded spill backtracking limit (7 hrs to compile) 555 cycles per msg iteration 69 cycles per key iteration
Interchanged	49 cycles per key iteration 44 min cycles for resources

Overview of Presentation

- Vector & Superscalar Processor Configurations
- Performance of a Highly Parallel Integer Loop
- ☞ Implementation Cost of Processor Configurations
  - ¤ use die area as measure of cost
- Summary & Conclusions

Expensive Vector Implementations?

highly parallel features:

- large number of integer and L/S units
- large capacity register file with tremendous bandwidth
  - 512 registers in vector register file
  - register file supports up to 40 reads and 24 writes in one clock period

⇒expect die area to be large!

but

- instruction issue and decode are simple
- organization of vector register file is area-efficient

Die Area Comparison

- compare die areas of actual implementations
  - consider processor portion only
  - exclude cache & TLB structures, external interface logic, pad ring

Processor Die Areas Scaled to 0.25μm

Micro-processor	Issue Mech	ROB size	Die Area (mm^2)
Torrent-0	1-way vector	----	14.73
Alpha 21164	4-way	----	21.86
Alpha 21164B	4-way	----	27.61
UltraSPARC II	4-way	----	37.77
MIPS R10000	4-way OOO	48-entry	66.92
HP PA-8000	4-way OOO	56-entry	67.77
Alpha 21264	6-way OOO	35-entry	69.81
SSbig	*6-way OOO*	*128-entry*	*250.00*

Comparison of Die Areas

- die areas of the two classes of superscalars are clearly different: in-order is about 2-3 times smaller than OOO
- vector processor is 1.5-2.5 times smaller than any of the supercalar processors despite supporting more parallelism
- SSbig is 3.5 times larger than other OOO: mainly due to its much larger ROB

Breakdown of Die Areas

Percentage Breakdown of Die Areas

Die Area Comparison

- compare die areas of actual implementations
  - consider processor portion only
  - exclude cache & TLB structures, external interface logic, pad ring
- adjust areas to account for features that are orthogonal to how a processor exploits parallelism
  - fabrication technology, clock frequency, datapath width, circuit and layout design
- remaining differences in area due to parallel-specific features
  - datapath configuration, register organization, instruction-issue mechanism

Estimated Die Areas for Vsmall, Vmed, Vbig

- figure 6b

Summary: Highly Parallel Vector Processor

- greater performance on highly parallel code because vector instructions
  - ¤ implicitly re-schedule operations and temporally localize parallelism
  - ⇒ exploit parallelism across loop iterations
  - ¤ reduce instruction bandwidth by more than a factor of 10
- die area requirements are comparable to future wide, highly parallel superscalar processor because
  - ¤ organization of vector register file provides tremendous bandwidth without incurring a large area penalty
  - ¤ control logic for issuing vector instructions is simple

Conclusions

amount of loop-level parallelism is greater than amount of instruction-level parallelism

performance gains and cost savings due to

- ¤ use of vector instructions
- ¤ reliance on vectorizing compiler, rather than hardware, to detect parallelism

⇒ tremendous cost/performance benefit

Future Studies

- demonstrate that results are not a one-loop phenomenon
- conduct performance analysis with realistic memory system
- investigate cost/performance tradeoff of combining highly parallel vector processor with moderately wide superscalar
- explore architectural techniques to increase "vectorizable" applications