How Low Can You Go?

Hank Dietz

LCPC, 11:30AM Oct. 11, 2017

University of Kentucky
Electrical & Computer Engineering

University of Kentucky

Abstract

It could be said that much of the evolution of computers has been the quest to make use of the exponentially-growing amount of on-chip circuitry that Moore predicted in 1965 – a trend that many now claim is coming to an end. Whether that rate slows or not, it is no longer the driver; there is already more circuitry than can be continuously powered. The immediate future of parallel language and compiler technology should be less about finding and using parallelism and more about maximizing the return on investment of power.

Programming language constructs generally operate on data words, and so does most compiler analysis and transformation. However, individual word-level operations often harbor pointless, yet power hungry, lower-level operations. This paper suggests that parallel compilers should not only be identifying and manipulating massive parallelism, but that the analysis and transformations should go all the way down to the bit or gate level with the goal of maximizing parallel throughput per unit of power consumed. Several different ways in which compiler analysis can go lower than word-level are discussed.

What's The Problem?

- No parallel programs!
  - Compiler finds stuff to execute in parallel
  - Parallel languages & libraries & tools

What's The Problem?

- No parallel programs!
  - Compiler finds stuff to execute in parallel
  - Parallel languages & libraries & tools
- How should it be done in parallel?
  - "All the wires, all the time"
  - Pipeline, SIMD, MIMD, VLIW, ...

What's The Problem?

- No parallel programs!
  - Compiler finds stuff to execute in parallel
  - Parallel languages & libraries & tools
- How should it be done in parallel?
  - "All the wires, all the time"
  - Pipeline, SIMD, MIMD, VLIW, ...
- Not enough power!
  - Scheduling to manage power use
  - Eliminate stuff I don't need to do

What Don't We Need To Do?

- Algorithms with too high O() complexity
- Common subexpressions; recomputation
- Excessive data motion

  ...

These things happen all the way down to the **bit level**, so why don't compilers look that low?

A Word About Words

- Most programming languages treat data objects as ***indivisible, atomic, entities***
- The programmer specifies type and size

  Fortran: `REAL*8 A`

  C: `int i; long long j;`
- Compiler anaylsis should ***look inside***
  - Eliminate processing meaningless bits by using smaller words or packed fields
  - Optimize algorithms at the bit level

Not All The Bits, Not All The Time

- Integer range analysis
- Floating point accuracy, not precision
- Packing of smaller data

Integer Range Analysis

- Programmers are **lazy** (paranoid?)
  - Language issues: how big is an `int`?
  - Overly generous specifications... e.g.:
    - PCC: 2,882 `int`, 174 `unsigned`, only 44 specifying 8, 16, 32, or 64 bits
    - LLVM: 3,135 `int`, 242 `unsigned`
- Why not allow syntax like C's `int:10`?
- Compiler range analysis set types in **1965**!

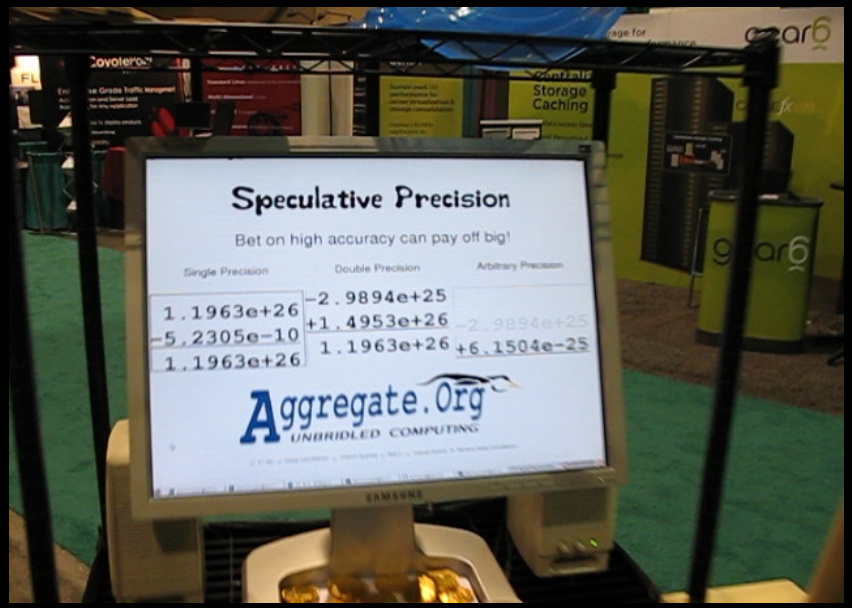
Benefits Of Integer Ranging

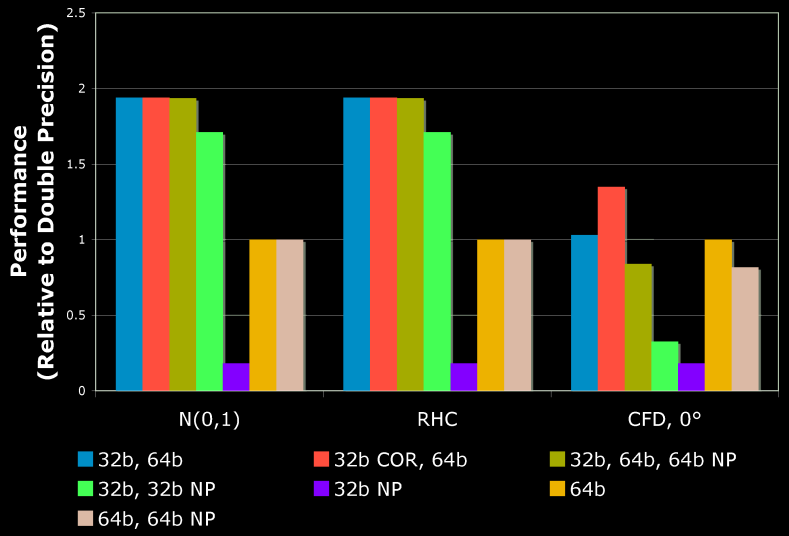
- Can ignore the bits that aren't active, e.g., only access low 16-bits of an `int` in [0..999]
  - Disable some wires and circuitry
  - Scatter/gather values (e.g., RISC-V AVS)
- Can use smaller storage space, thus reducing power use by:
  - Keeping more objects in registers/cache
  - Moving fewer bits/object

FP Accuracy, Not Precision

- Normally specify precision of floating-point
- Accuracy analysis is very difficult
- Accuracy analysis is very conservative; analysis often finds ***no*** significant digits, while computations typically have plenty
- Language constructs can help...

The Loosest Slots In Reno





- The first 32 terms of the Taylor series for $e^{-2\pi}$
- *Heavy Cancellation* sums $1 + ... + 1 + 1 \times 10^{-18} + 1 \times 10^{-18} - 1 - ... - 1$
- 32 *Uniform Spacing* values between 1.0 and 2.0
- $N(0, 1)$adds Gaussian random numbers with $\mu = 0, \sigma = 1$
- *Inverse Square* is $\sum_{i=1}^{32} \frac{1}{i^2}$
- *Random Heavy Cancellation* is $\sum_{i=1}^{32} \pm 10^{x_i}$, where x_i is Gaussian with $\mu = 0, \sigma = 35$, but clipped to [-35, +35]

- 32-bit usually ok; 64-bit sometimes isn't!

University of Kentucky

Specifying FP Accuracy

```
#faildef exit();
#specdef fd(float, double)
#speculate fd
fd a=x; double b=sqrt(a);
if (!mytest(b, x)) {
#fail
} y=b;
#commit
```

University of Kentucky

Specifying FP Accuracy

```
#define faildef { exit(1); }
#define fd float
{fd a=x; double b=sqrt(a);
if (!mytest(b, x)) {goto fail0;}
y=b; } goto commit0;
#define fd double
fail0:; {fd a=x; double b=sqrt(a);
if (!mytest(b, x)) {faildef}
y=b; } commit0: ;
```

Benefits For Floating-Point

- Huge performance gains for low precision
  - AMD RADEON INSTINCT MI25 GPU:
    64-bit: 0.768 TFLOPS
    32-bit: 12.3 TFLOPS
    16-bit: 24.6 TFLOPS
  - Memory footprint & bandwidth
- Potential to use LNS or scaled integer

Packing Smaller Data

- SWAR (SIMD Within A Register)
  - Originally, to obtain vector-like parallelism
  - More efficient use of memory & datapaths
- Virtualized in RISC-V AVS
- Compiler can ***pack unstructured things***: Common Subexpression Induction (CSI)

From Bits To Words

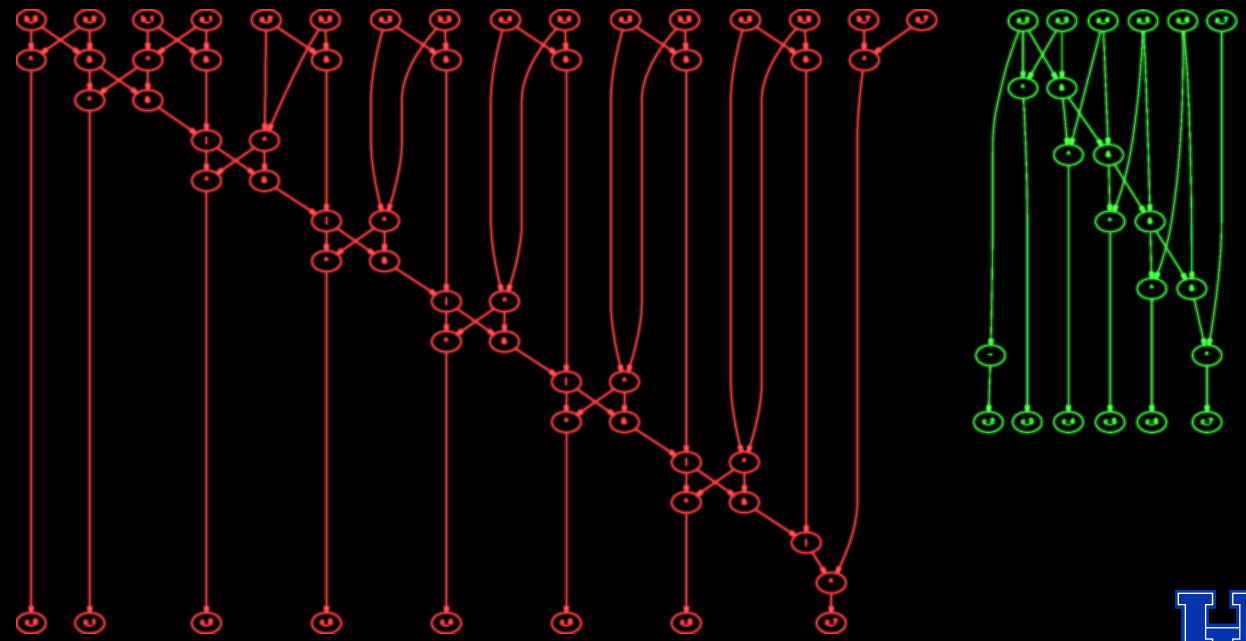
- 1958 *EDSAC 2* used microcoded bit-slicing; Various *PDP-11* were 4-bit; then 8, 16, 32, 64
- Massively-parallel microcoded bit-slicing in *DAP*, *STARAN*, *MPP*, *CM*, *CM2*, *GAPP*; *MP-1* was 4-bit; then 32 and 64
- This was done to speed sequential code... assuming not enough parallelism is available

From Bits To Words, And Back Again

- Why go back to bit-slicing?
  - Sequential code is *handled elsewhere*
  - Lots of parallelism available
- Fewer gates active per computation, e.g.:
  - 32 ripple carry 32-bit Adds in 32 clocks
  - To get one 32-bit Add in 1 clock, need *additional hardware* for carry lookahead...

True Bit-Level Optimization

- Bit-slice systems were generally microcoded to implement a simple word-level ISA
- Word-level operations can imply useless work
  - E.g., using an Add to add 4 to a register:



University of Kentucky

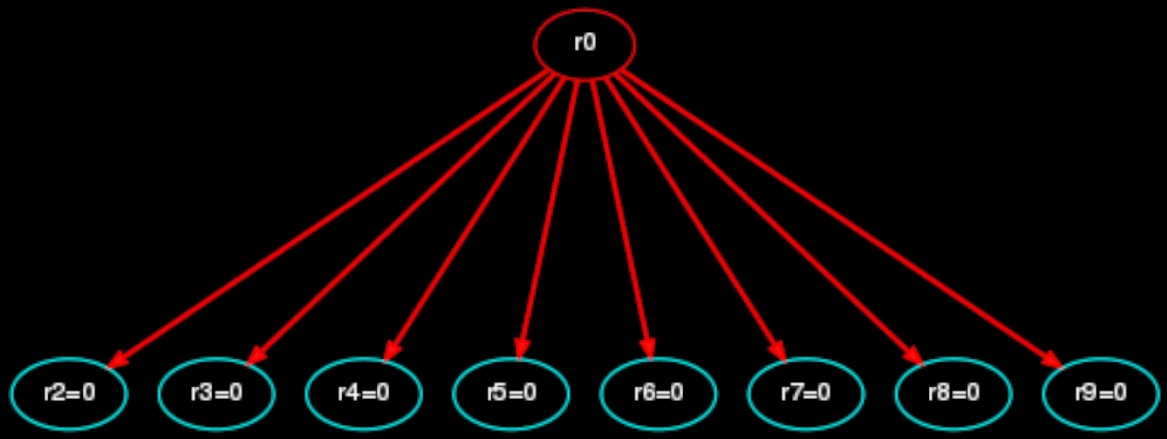
True Bit-Level Optimization

- How do we optimize gate-level designs?
  - Karnaugh maps?
  - Quine-McClusky algorithm?
  - Espresso?
  - Pattern matching with fixed modules?
- BitC language & compiler for nanocontrollers
  - Karplus algorithm for BDD normal form
  - Transformations to reduce execution cost

True Bit-Level Optimization

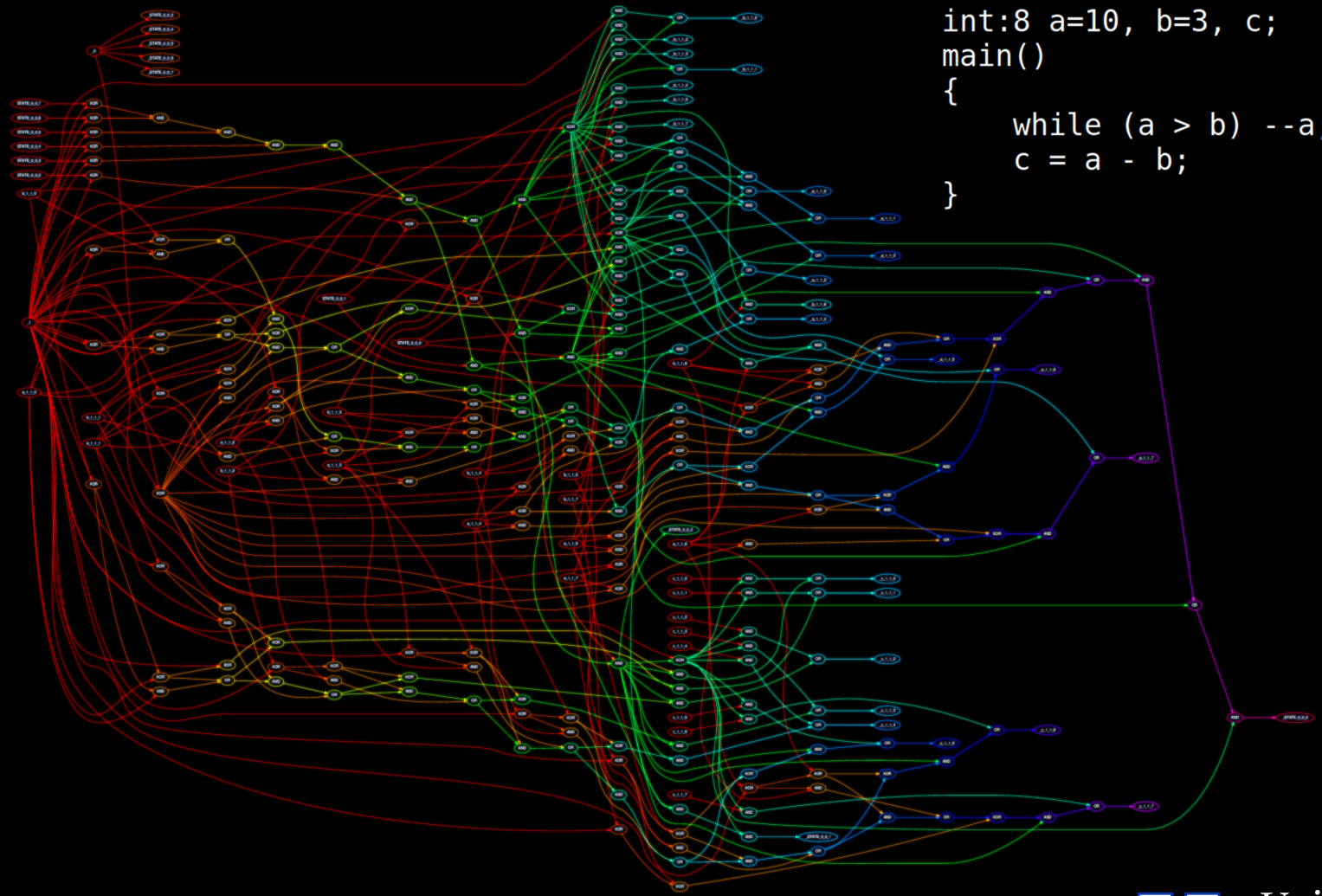
```
int:8 a, b, c;
a = (c * c) ^ 70;
a = ((a >> 1) & 1);
a = b + (c * b) + a;
a = a + ~(b * (c + 1));

Total of 206669 ITEs created, 8 kept
```



University of Kentucky

Whole Program Scale Gate Optimization

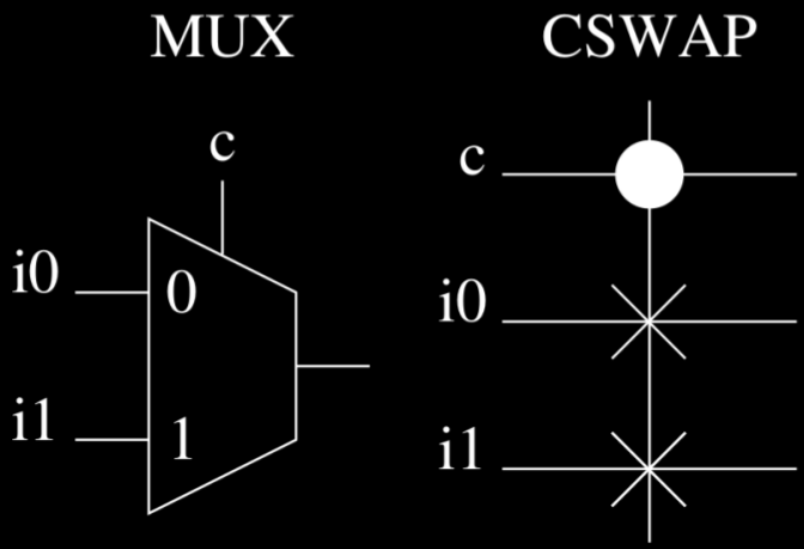


```
int:8 a=10, b=3, c;
main()
{
    while (a > b) --a;
    c = a - b;
}
```

University of Kentucky

New Targets

- Reconfigurable logic, GPUs, TrueNorth
- Adiabatic is thermodynamically reversible
- Quantum is adiabatic using entangled Qbits



c	i0	i1	MUX	CSWAP
0	0	0	0	0 0 0
0	0	1	0	0 0 1
0	1	0	1	0 1 0
0	1	1	1	0 1 1
1	0	0	0	1 0 0
1	0	1	1	1 1 0
1	1	0	0	1 0 1
1	1	1	1	1 1 1

University of Kentucky

Conclusions

- Reduce power by doing less work
  - Limit precision to what's needed
  - Remove extra bit-level operations from word-level constructs
- Reduce power by using new architectures
  - Massively-parallel bit-slice systems
  - Reversible computing, etc.

University of Kentucky