UNIVERSITY OF KENTUCKY

Big Ideas
About Control Of Tiny Devices

Professor Hank Dietz

ECE Seminar, Sept. 14, 2007

University of Kentucky
Electrical & Computer Engineering

Kentucky UNBRIDLED SPIRIT

Abstract

The primary contribution of computer technology to society is not PCs, the Internet, or supercomputing for "grand challenge" problems; it is the ability to make an amazing range of ordinary devices intelligent. Programmable control is everywhere -- except where the circuit complexity of a microcontroller per device cannot be accommodated and routing signals from many devices to a centralized controller is impractical.

Over the past four years, we have been creating technologies that will allow a parallel computer containing as many as millions of independently programmable "nanocontrollers" to be embedded to control an image sensor or display, MEMS chip, or array of nanofabricated devices. Despite requiring as few as ~100 transistors per nanocontroller, a conventional C-based programming environment is supported. This talk will overview nanocontroller concepts, current status of the research, and opportunities for collaboration.

Nanocontrollers

Programmable control for **ANY** device
- Nanofabricated devices
- MEMs (e.g., DLP/DMD chips)
- Bigger, low circuit density, things… LCDs, organic semiconductor sheets, etc.

Assumes there are **LOTS** of nanocontrolled devices in a system

Currently ~100 transistors/nanocontroller

What Must A Nanocontroller Be Able To Do?

Minimal circuit complexity
Predictable real-time behavior
Localized analog & digital I/O
Coordination as a PE in a parallel computer
Each nanocontroller independently programmable (MIMD parallelism)
Reprogramability

What Architecture Can Do All That?

No previous model... SIMD is closest

MIMD-on-SIMD technology, circa 1992:

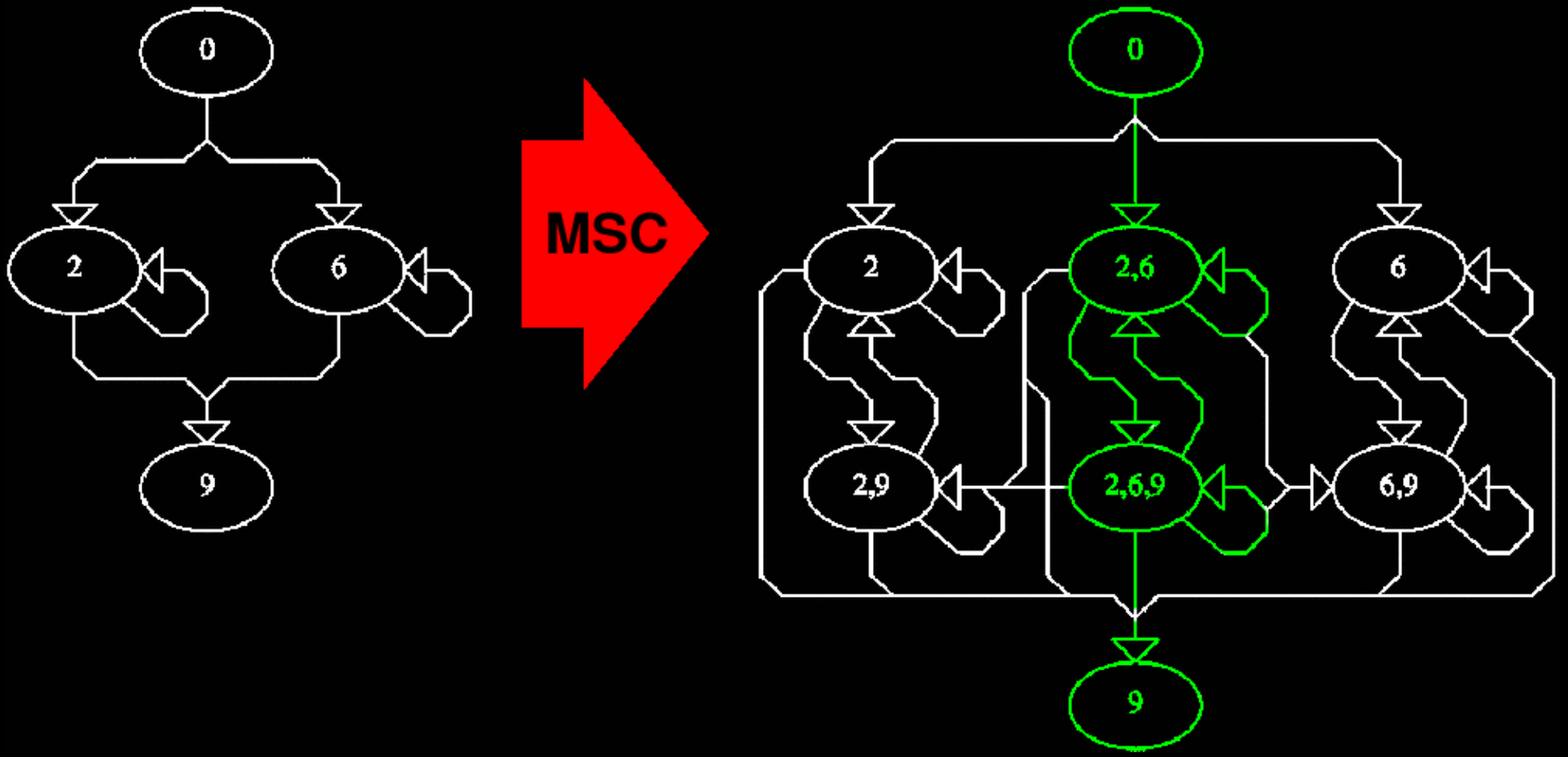
- Meta-State Conversion (MSC)
- Common Subexpression Induction (CSI)
- Worked for **MasPar MP1** supercomputer, but significance not appreciated...

Kentucky Architecture concepts:

- SIMD masking + VLIW control flow
- MSC + CSI = MIMD without code copies

Meta-State Conversion (MSC)

```
if(A){ do {B} while(C); }else{ do {D} while(E);} F
```



Common Subexpression Induction (CSI)

CSI reduces nanoprocessor time "disabled"

MSC states like **{2,6}** contain multiple code blocks – original states **2** and **6** in this case

Execute time would be **time(2) + time(6)**

CSI tries to make **2** and **6** use the same instructions so execution time approaches **max(time(2), time(6))**

Impact Of MSC + CSI

- Minimal per-nanocontroller hardware
- Real-time constraints can be maintained by timing analysis in MSC + CSI or by polling a (relatively slow) global clock signal
- Coordination as a parallel computer can use a SIMD-like inter-nanoprocessor network, without hardware routing or arbitration logic
- MIMD programs work as expected
- Reprogramming might have a long compile time, but is easily accomplished

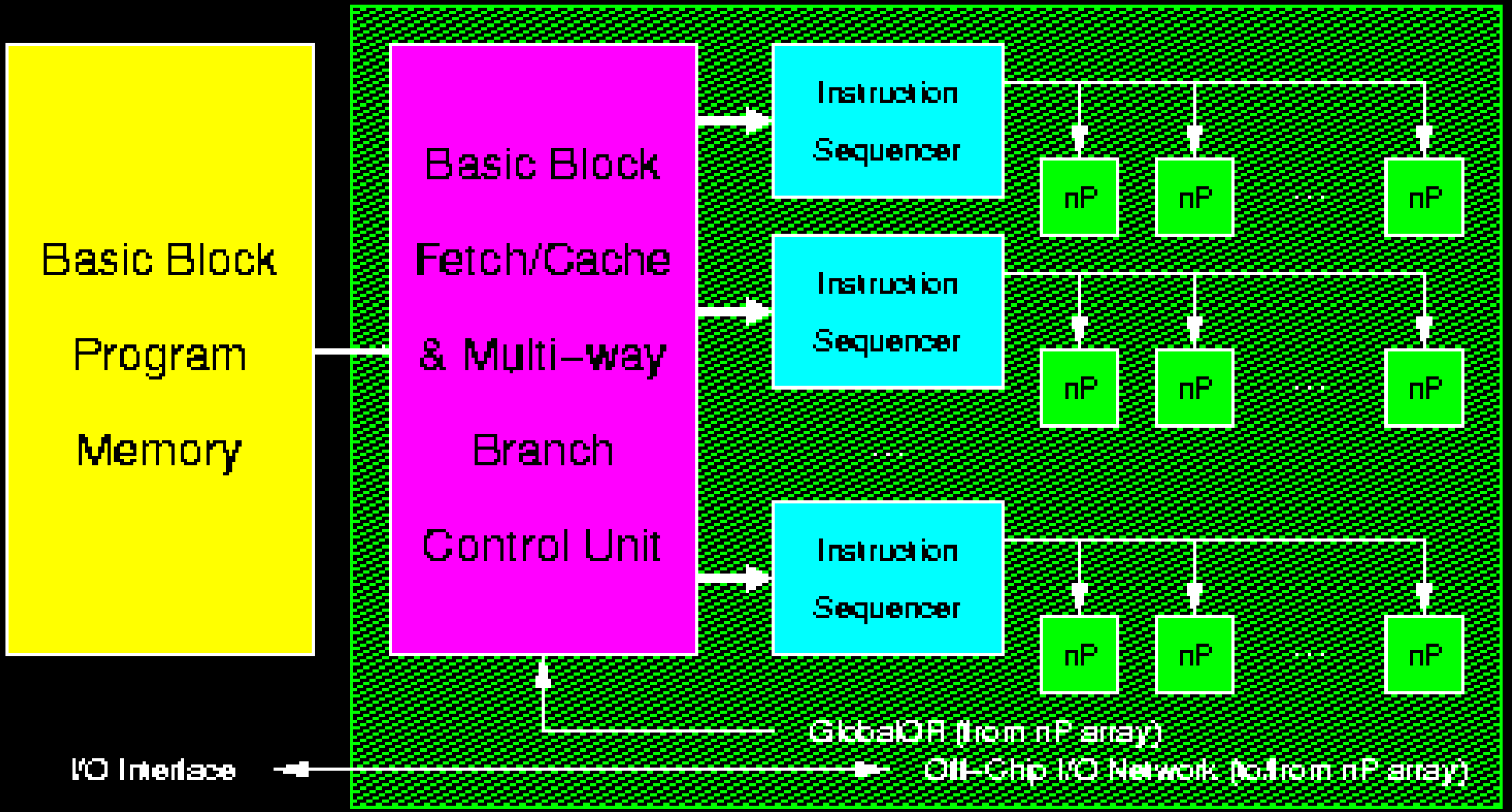
Localized Device Control Input/Output

- Digital input: decoded like reading a register
- Digital output: write a register
- Analog input: measures a time constant
  - Reset analog accumulation by write
  - Threshold crossing detected by read
  - Timing is done by a software counter
- Analog output: Pulse Width Modulation
  - PWM output is really a digital output
  - Filter in analog circuit (if needed)

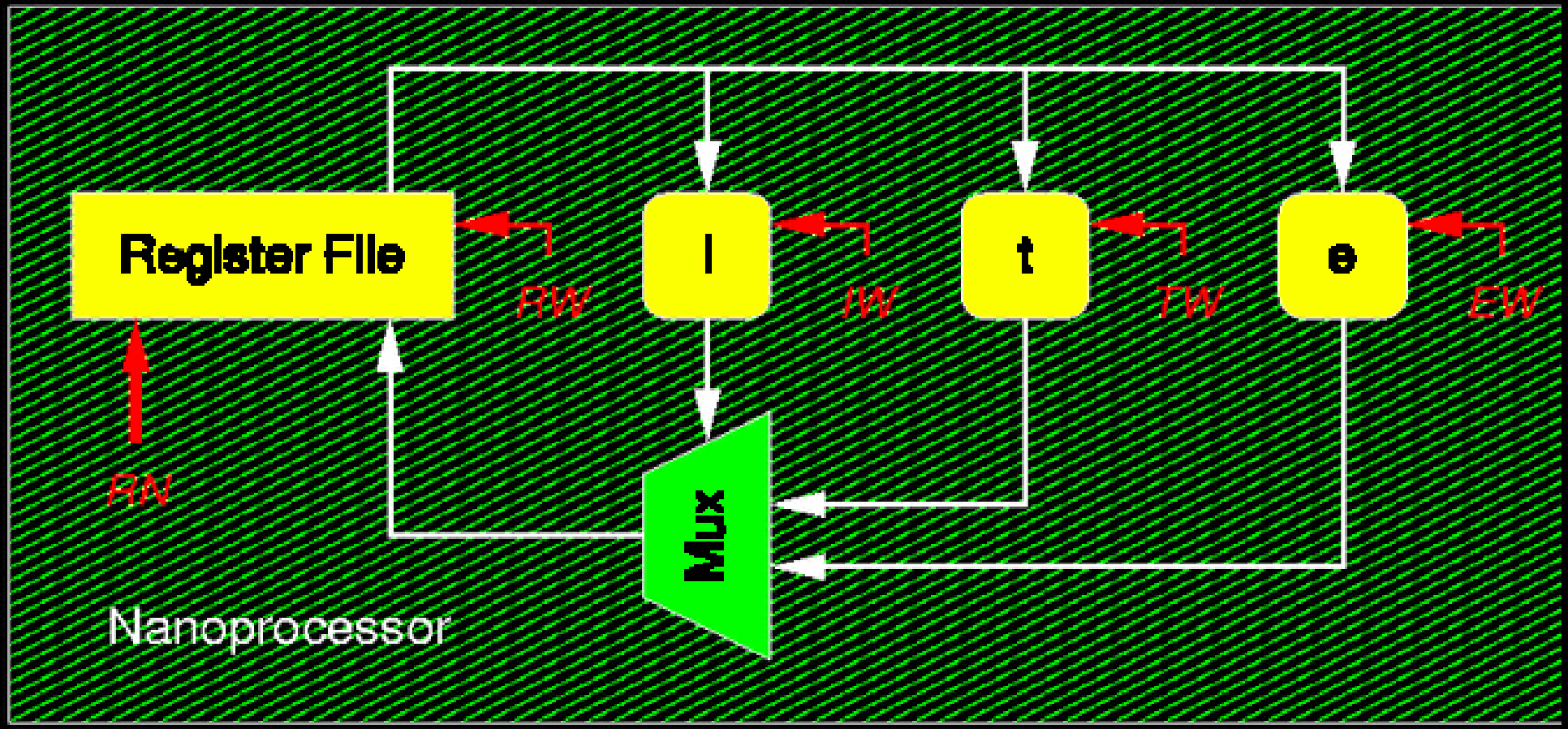
Kentucky If-Then-Else (KITE)

First “Kentucky Architecture” design
Single, off-chip, program memory
Control Unit fetches a block of instructions at a time and does VLIW-like multi-way branch
Control hierarchy allows nanocontrollers to have a fast clock, despite a slow global clock
Only instruction is Store-If-Then-Else (SITE)
ITE is a 1-of-2 multiplexor
ITE directly implements enable masking

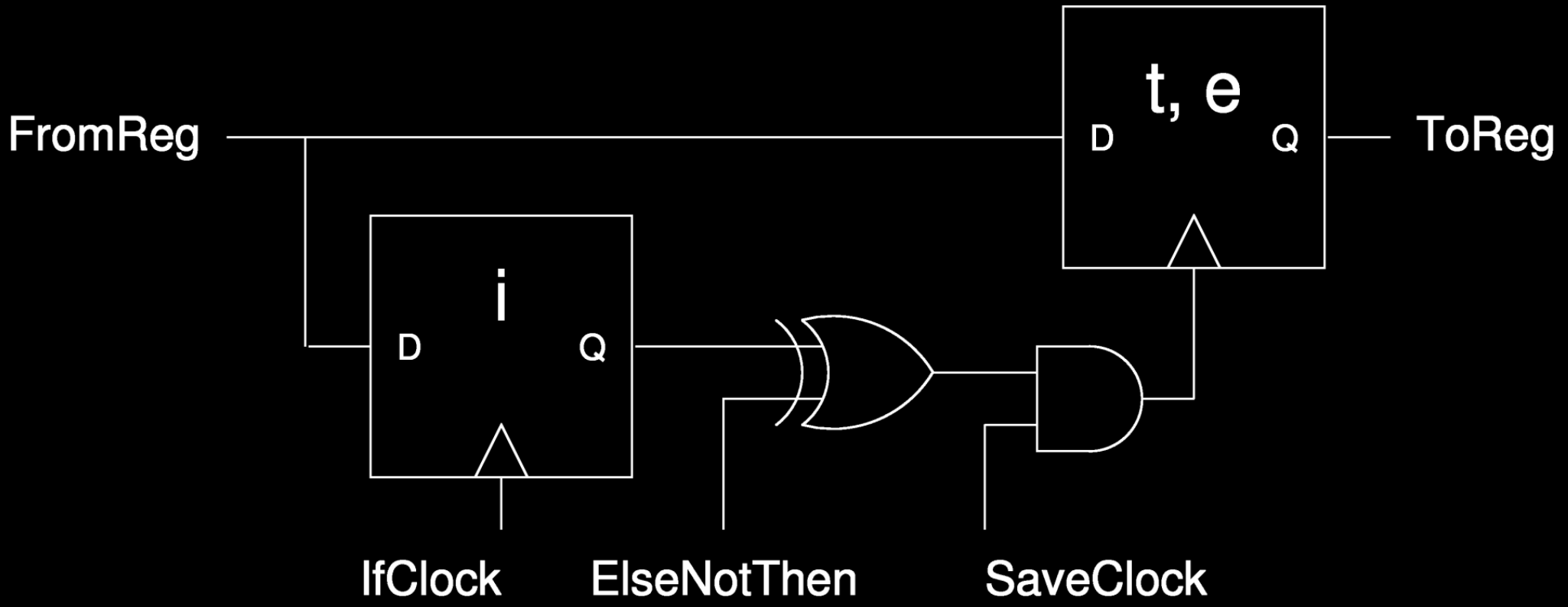
KITE Abstract Architecture



KITE Nanoprocessor Abstract Architecture



KITE Nanoprocessor Implementation Architecture



Programming Language: BitC

A very small C dialect

Minor extensions to C data types:

Explicit precision using C bitfield syntax; e.g.,

```
int:3 x;
```

I/O & network are register-mapped; e.g.,

```
int:1 adc@5;
```

All applicable C operators plus a few more: `?<` (min), `?>` (max), `$` (ones), etc.

The usual control flow constructs

From Word-Level To Bit-Level

BitC code:

```
unsigned int:2 a, b, c;
c = a + b;
```

Unlike C, full precision results are available;
adding 2-bit values produces a 3-bit result

Bitwise logic expressions:

```
c0 = (a0 XOR b0)
c1 = (a1 XOR b1) XOR (a0 AND b0)
```

ITE Equivalents For Familiar Logic Operations

Like NAND, ITEs are complete
XORs are not efficient using ITEs

Logic Operation	Equivalent ITE Structure
(x AND y)	(x ? y : 0)
(x OR y)	(x ? 1 : y)
(NOT x)	(x ? 0 : 1)
(x XOR y)	(x ? (y ? 0 : 1) : y)
((NOT x) ? y : z)	(x ? z : y)

Transformation Into ITEs

Bitwise logic expressions:

```
c0 = (a0 XOR b0)
c1 = (a1 XOR b1) XOR (a0 AND b0)
```

ITE equivalents:

```
c0 = (a0 ? (b0 ? 0 : 1) : b0)
c1 = ((a1 ? (b1 ? 0 : 1)) : b1) ?
       (((a0 ? b0 : 0) ? 0 : 1) ?
        (a0 ? b0 : 1))
```

Enable Masking Using ITEs

Consider:

```
if (a) { b=c; if (d) e=f; else g=h; i=j; } k=l;
```

By simple if-conversion, we get:

```
b = (a ? c : b);
e = ((a ? d : 0) ? f : e);
g = ((a ? (d ? 0 : 1) : 0) ? h : g);
i = (a ? j : i);
k = l;
```

Our Transformation Into ITEs

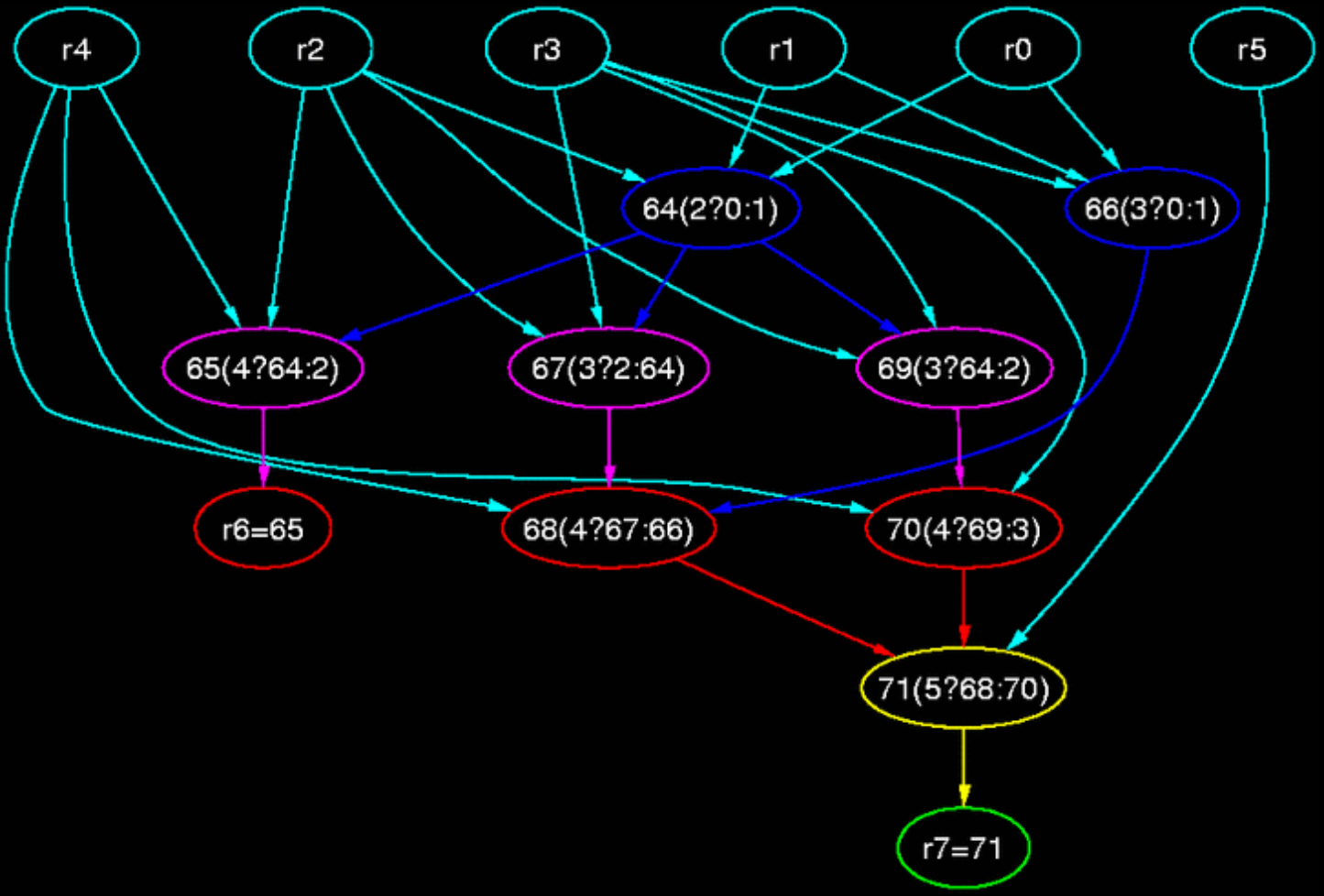
We don't do simple pattern substitutions

We have adapted logic circuit analysis and minimization technology to optimize bit-serial nanocontroller programs

Thus far, 4 M.S. worth of work on this

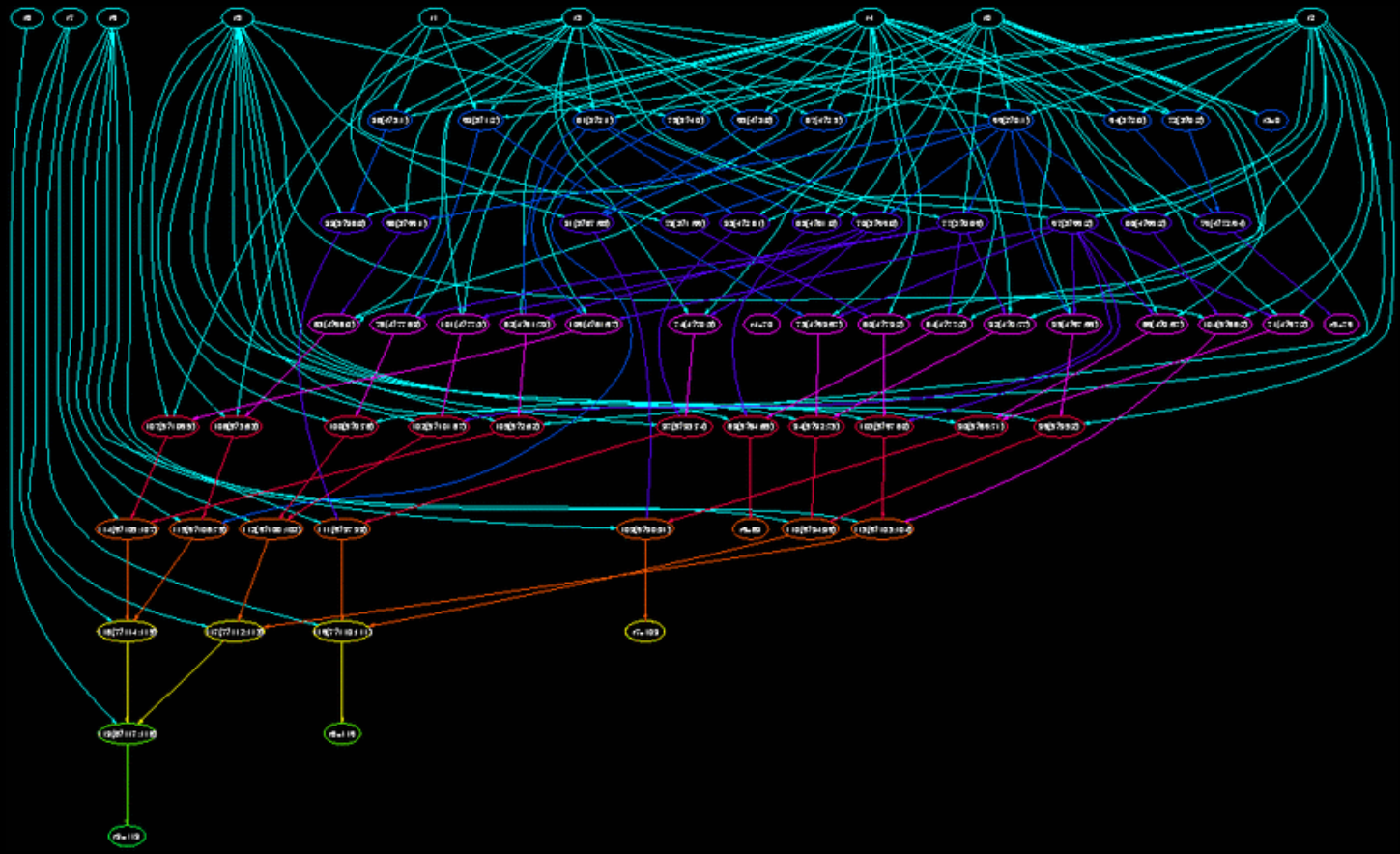
Our techniques primarily extend those of **Bryant** and **Karplus** involving normalization of Binary Decision Diagrams (BDDs)...

Bryant's Normal Form



A Larger Example

```
int:8 a; a = a * a;
```



Preliminary Results

Compiler speed is not a problem

The normal form transformations perfectly recognize even word-level identities identities

```
int a,b; a=a-b; a=a+b;
```

generates no code!

Complexity of XOR-based math is awkward for more than about 12-bit precision

```
int:12 a,b; a=a*b;
```

generates 156,392 ITEs!

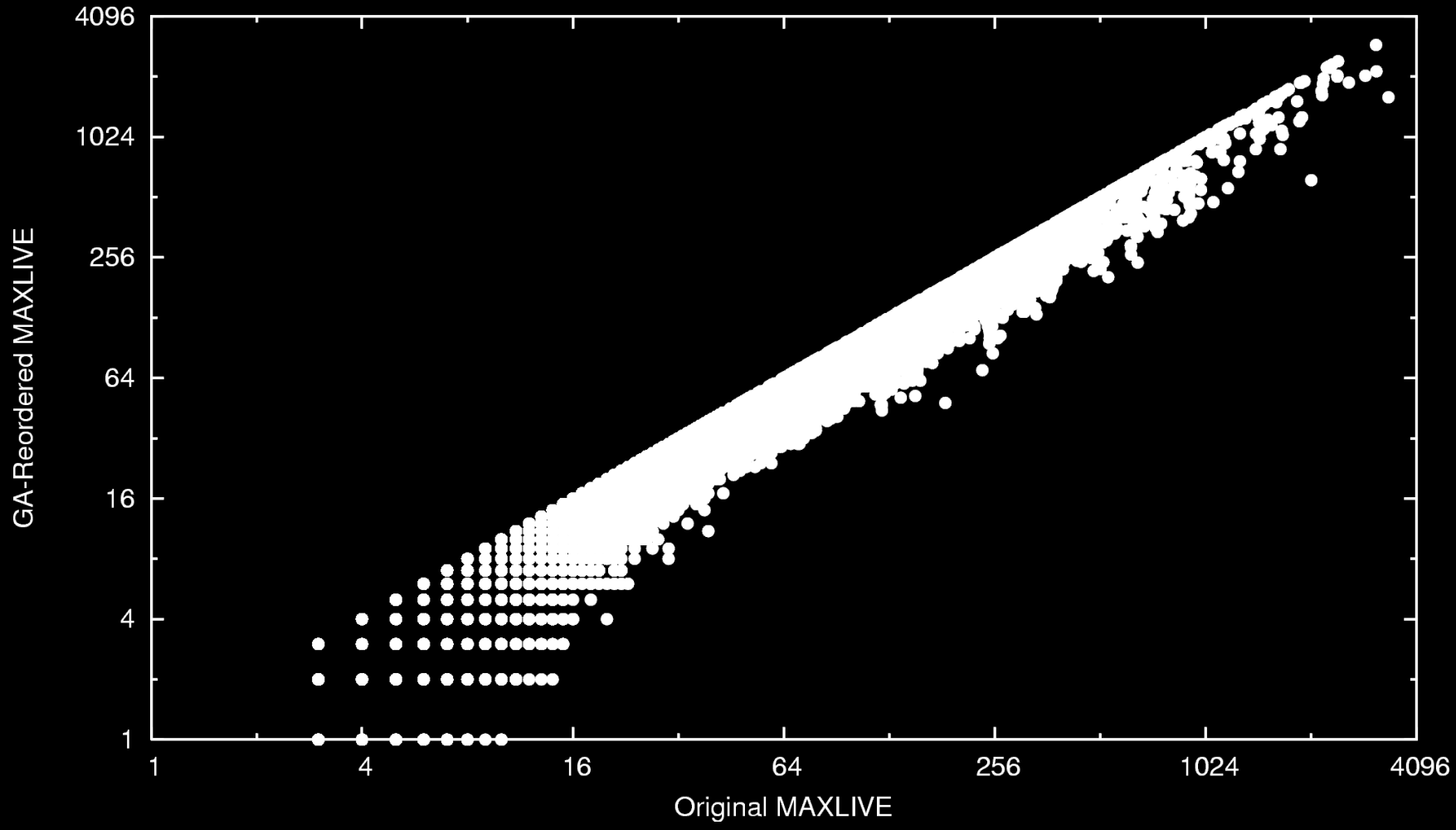
How many temporary registers do we need?

The MAXLIVE Problem (solved in 2005/2006)

Register Allocation was the killer problem:
1-bit operations increase DAG complexity
Trinary ops increase DAG complexity
Basic blocks often needed 1000s of registers
Needed to dramatically **reduce MAXLIVE**
Developed two new techniques: GA & SUN-GA

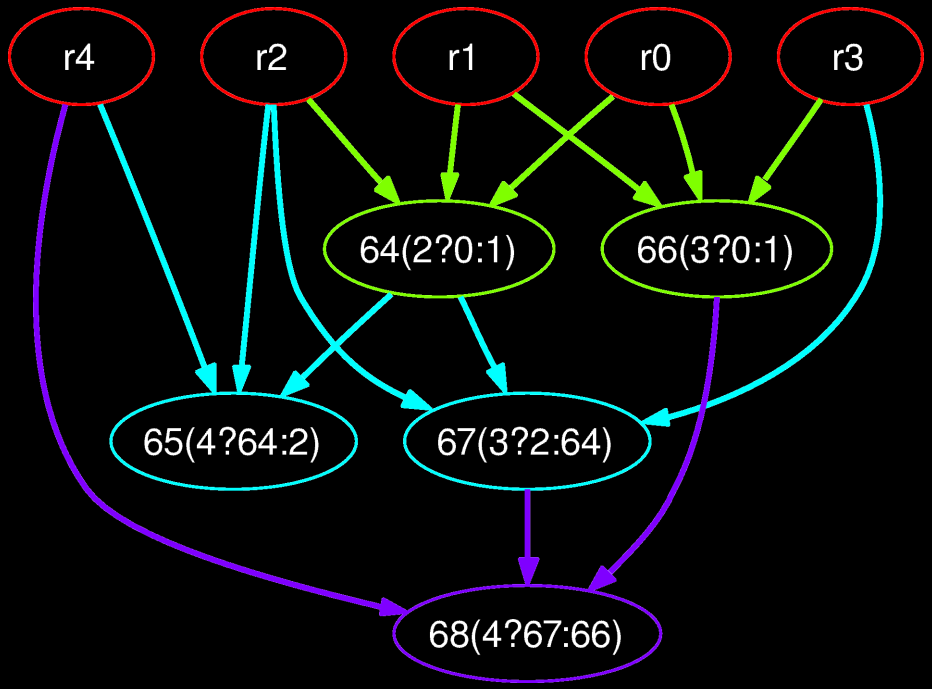
UNIVERSITY OF KENTUCKY

GA-Reordered MAXLIVE Vs. Original MAXLIVE

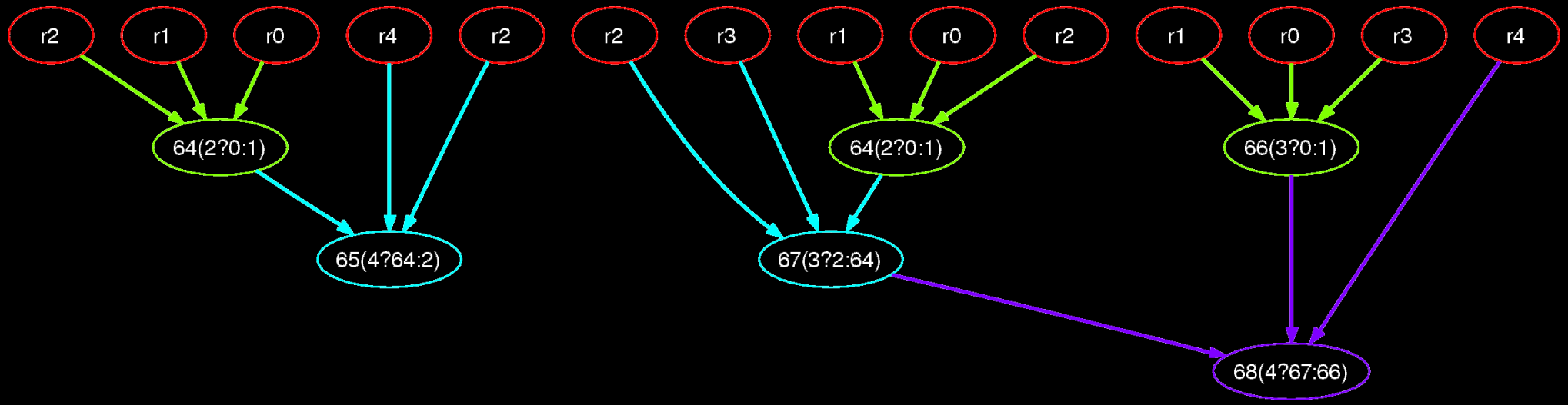


Kentucky UNBRIDLED SPIRIT

DAGs To Trees: A Sample DAG



DAGs To Trees: The Corresponding Trees



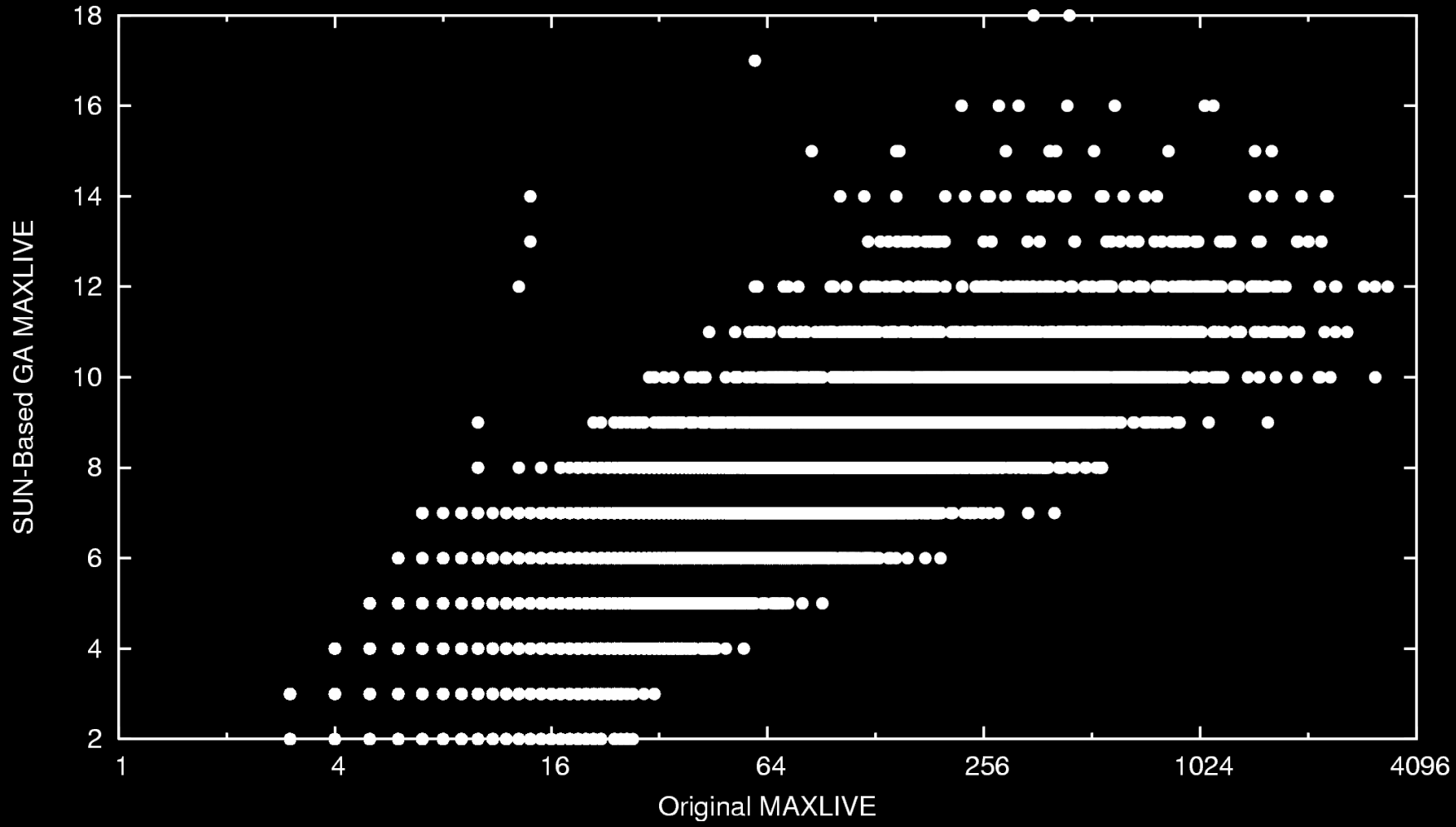
SUN-GA Experimental Results

Results from 32,912 accepted test cases... the same ones used for the reordering GA, so direct comparison of results is valid

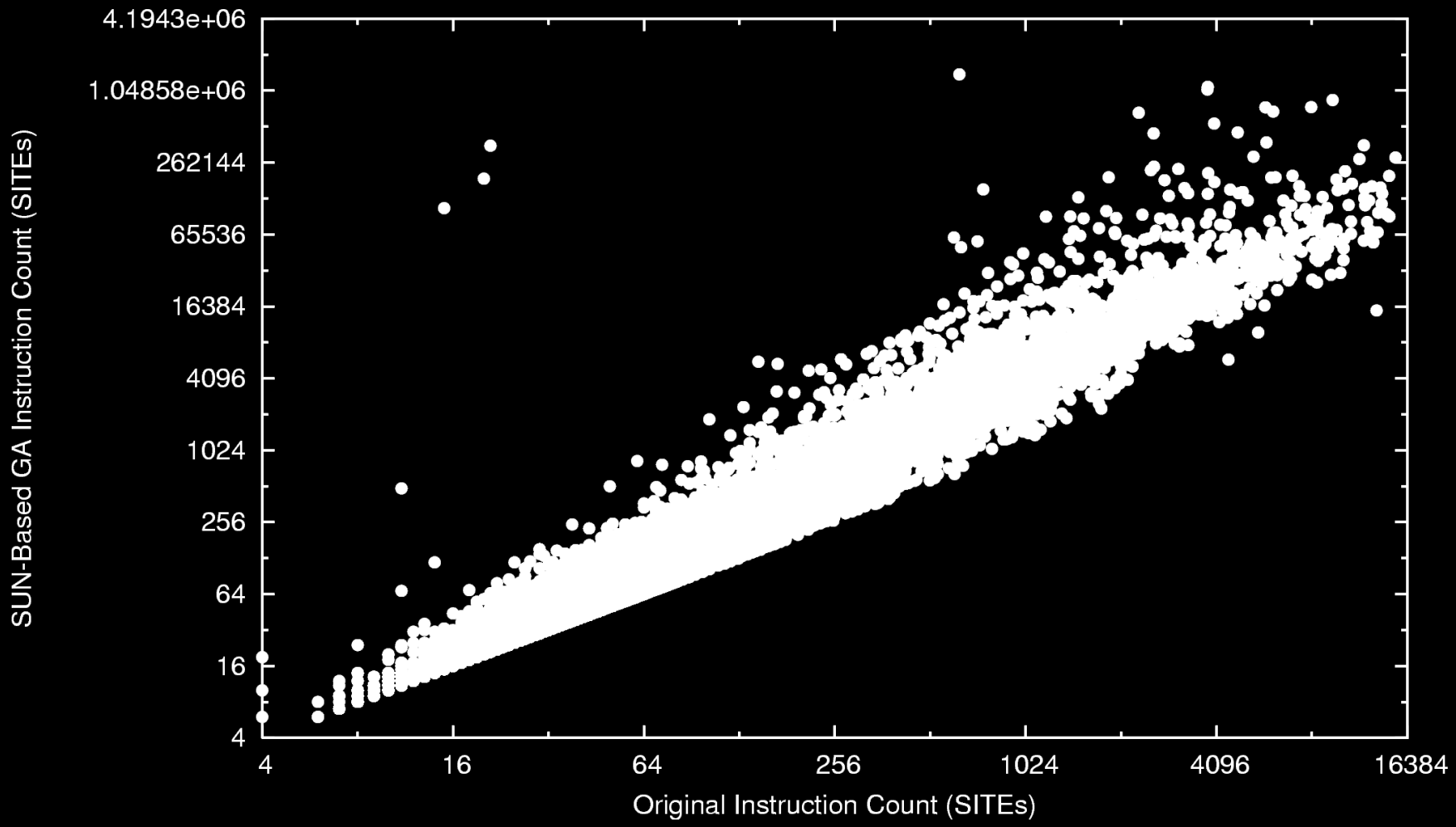
The goal was to minimize MAXLIVE, secondarily minimizing number of SITEs

Execution time was limited to about 1 minute per test case on an Athlon XP

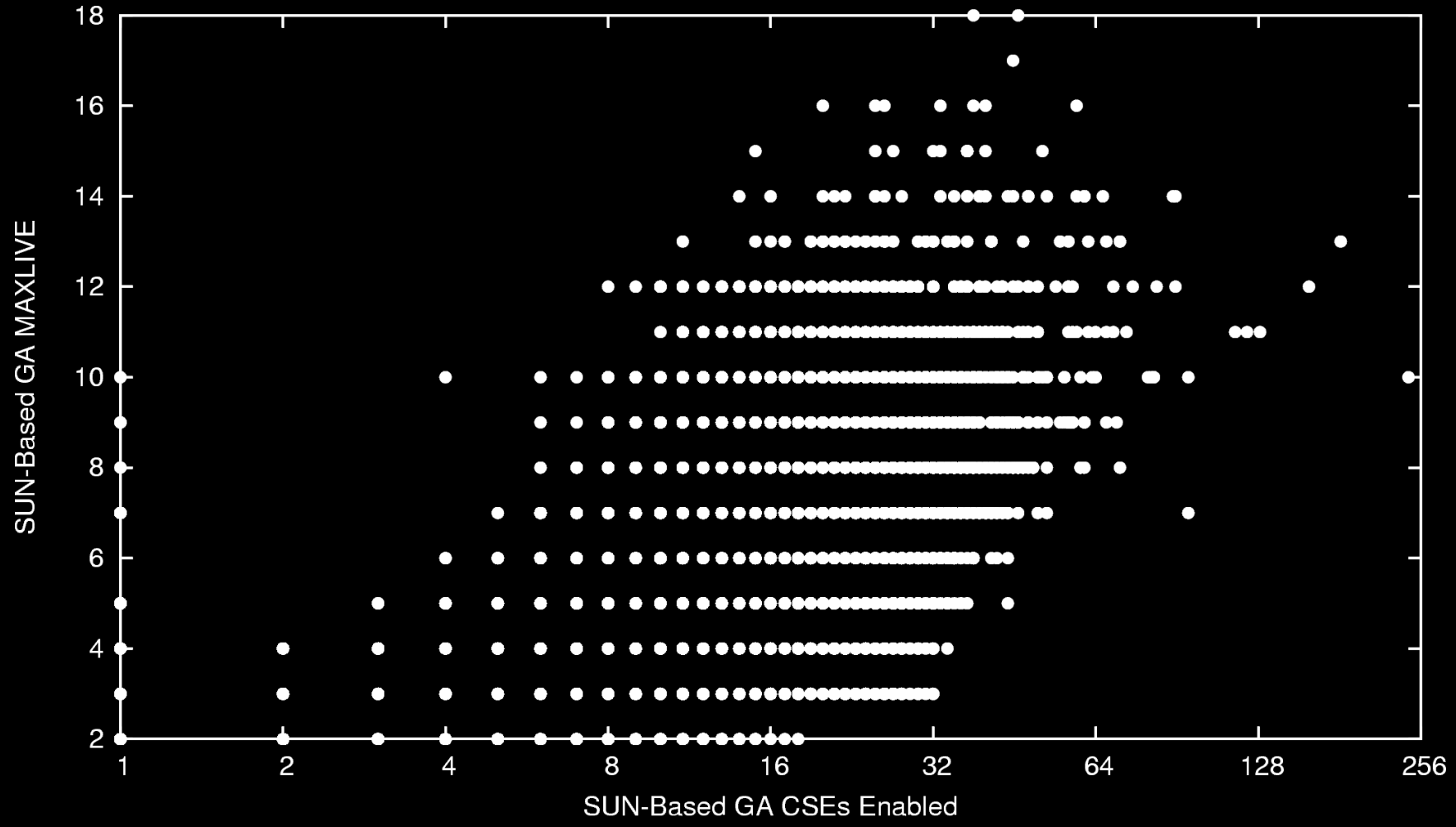
SUN GA Vs. Original MAXLIVE



SUN GA Vs. Original SITEs



SUN GA MAXLIVE Vs. CSEs Enabled



MAXLIVE Problem Solved

The Reordering GA wasn't good enough

Aggressive SUN-Based GA works:

- 8X increase in SITEs was common, worst was 15,309 and became 1,431,548
- MAXLIVE reduction also was huge, from a maximum over all test cases of 3,409 to 18 (a 189:1 improvement!)

Fortunately, targeting a specific MAXLIVE can greatly reduce SITE count

Current & Future Work

Physical implementation

"Logic under devices" issues

Details of the control hierarchy

Local and global I/O hardware details

Additional compiler improvements

Switched analog nanocontrollers?

"Killer applications" -- this month, we filed IP involving an application that could obsolete both still and video cameras....

UNIVERSITY OF KENTUCKY

Questions?

Kentucky
UNBRIDLED SPIRIT
TM