X-Ray: Automatic Measurement of Hardware Parameters

Keshav Pingali
pingali@cs.cornell.edu
Department of Computer Science
Cornell University

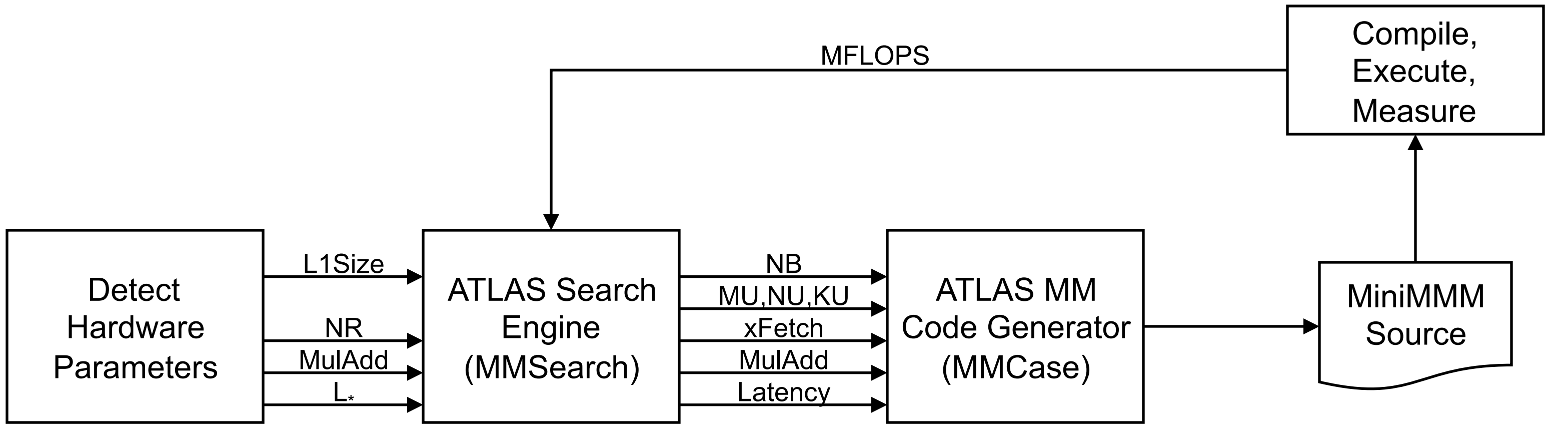
Joint work with:
Kamen Yotov, Paul Stodghill

Organization of talk

- Motivation for X-Ray
  - self-optimizing systems: ATLAS, IBM Autonomic Computing Project
  - need accurate estimates of hardware parameters
- Measurement of hardware parameters
  - CPU Features
    - Frequency
    - Instruction
      - Latency
      - Throughput
      - Existence
    - Number of Registers
  - Memory Hierarchy
    - Multilevel Caches, TLB
      - Associativity
      - Block Size
      - Capacity

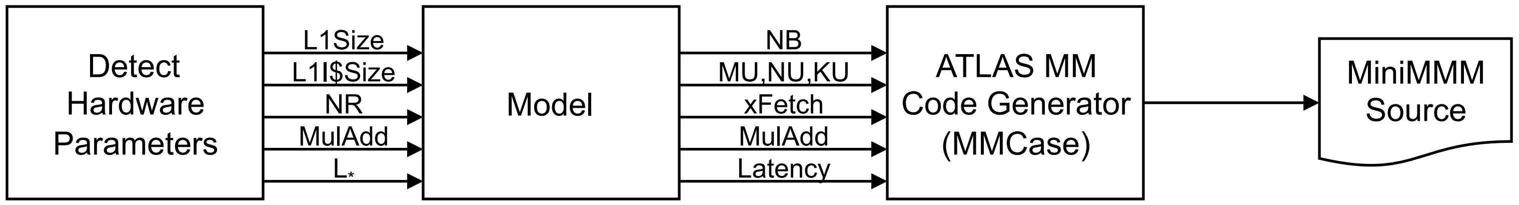
Motivation: self-optimizing systems

- Original ATLAS Infrastructure



MFLOPS
Compile, Execute, Measure
Detect Hardware Parameters
L1Size
NR
MulAdd
L*
ATLAS Search Engine (MMSearch)
NB
MU,NU,KU
xFetch
MulAdd
Latency
ATLAS MM Code Generator (MMCase)
MiniMMM Source

- Model-Based ATLAS Infrastructure

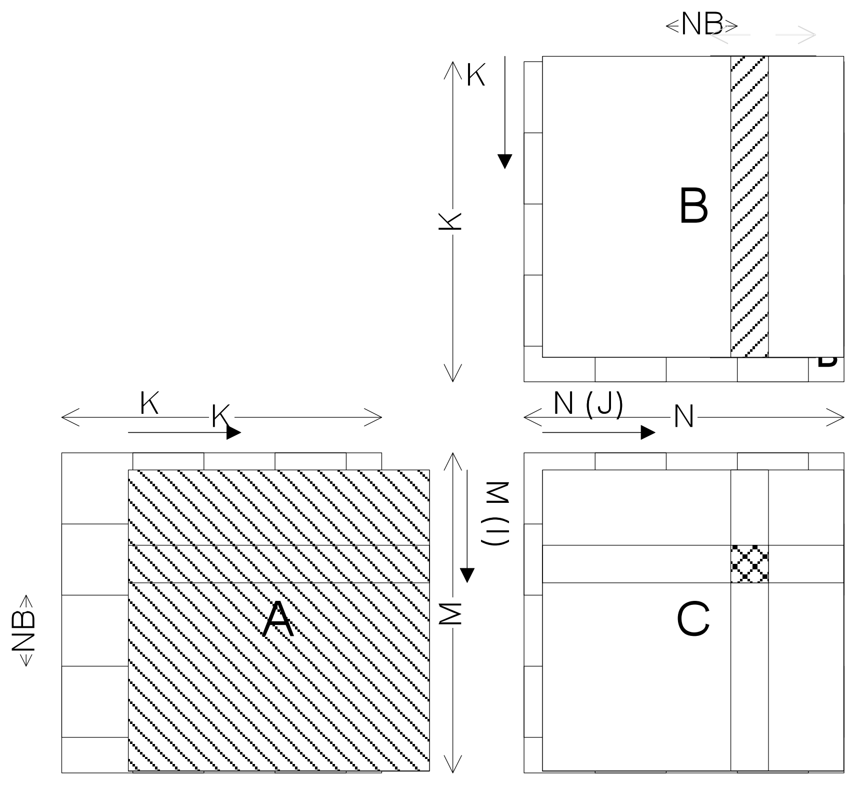


Need for hardware parameter values

- Empirical (search-based) optimization
  - Need hardware parameters to limit search space
  - Hardware parameter values need not be very accurate
- Model-driven optimization
  - Need hardware parameters to estimate optimization parameters
  - Hardware parameter values must be very accurate

Example: Modeling for Tile Size in ATLAS (NB)

- Models of increasing complexity
  - $3 * NB^2 \le C$
    - Whole work-set fits in L1
  - $NB^2 + NB + 1 \le C$
    - Fully Associative
    - Optimal Replacement
    - Line Size: 1 word
  - $\left\lceil \frac{NB^2}{B} \right\rceil + \left\lceil \frac{NB}{B} \right\rceil + 1 \le \frac{C}{B}$ or $\left\lceil \frac{NB^2}{B} \right\rceil + NB + 1 \le \frac{C}{B}$
    - Line Size > 1 word
  - $\left\lceil \frac{NB^2}{B} \right\rceil + 2\left\lceil \frac{NB}{B} \right\rceil + \left(\left\lceil \frac{NB}{B} \right\rceil + 1 \right) \le \frac{C}{B}$ or $\left\lceil \frac{NB^2}{B} \right\rceil + 3NB + 1 \le \frac{C}{B}$
    - LRU Replacement



Why not consult architecture manuals?

- Autonomic systems
  - Require online manuals
- Incomplete
  - Parameters like capacity and line size of off-chip caches vary from model to model
  - Not usually documented in processor manuals
- Moving Target
  - Even same model of computer may be shipped with different cache organizations
- Hardware values vs availability to software
  - (eg) number of hardware registers may not be equal to number of registers available to programs for holding values (register 0 on SPARC)
  - For software optimization, hardware values may not be relevant

X-Ray strategy

- Set of micro-benchmarks in C
- Usage:
  - Download and compile on any architecture
  - Run micro-benchmarks
    - X-Ray deduces parameter values from timing results
- Paradox:
  - Compiler optimizations may confuse timing results
  - Cannot afford to turn off all optimizations though
- Some amount of O/S specific code
  - High-resolution timing routines
  - Super-page allocation
  - Currently support Linux
  - Windows and Solaris in the works

High-level idea: Frequency

- Assumption
  - Integer ADD has latency=throughput=1 cycle
- Plan
  - Measure latency of “ADD r1,r2” in seconds
  - Declare reciprocal as frequency of processor
- Code

```
t = gettime();
r1 += r2;
return gettime() - t;
```

Step by Step: Frequency

```
t = gettime();
r1 += r2;











return gettime() - t;
```

Problem: hard to measure small time intervals accurately

Step by Step: Frequency

```
t = gettime();
while (--R) //R is number of repetitions
  r1 += r2;









return gettime() - t;
```

Problem: loop overhead

Step by Step: Frequency

```
t = gettime();
i = R / U;
while (--i) //loop unrolled U times
{
   r1 += r2;
   r1 += r2;
   .........
   r1 += r2;
}



return gettime() - t;
```

Problem: compiler optimizations

Step by Step: Frequency

```
volatile int v = 0;
```

```
t = gettime();
i = R / U;
switch (v)
{
   case 0: loop:
   case 1: r1 += r2;
   case 2: r1 += r2;
   ..................
   case U: r1 += r2;
   if (--i)
     goto loop;
}
if (!v) return gettime() - t; else use(r1,r2);
```

C Code

```
volatile int v = 0;
volatile int vr = 0;
register int r1 = vr;
register int r2 = vr;
t = gettime();
i = R / U;
switch (v)
{
    case 0: loop:
    case 1: r1 += r2;
    case 2: r1 += r2;
    .................
    case U: r1 += r2;
    if (--i)
       goto loop;
}
if (!v)
    return gettime() - t;
else
{
    vr = r1;
    vr = r2;
}
```

Experimental Results: Frequency

Architecture	Frequency	Error
Pentium III @ 1GHz	981.482	1.85%
Pentium III @ 500MHz	495.025	1.00%
Pentium 4 @ 2.4GHz	4740.582	97.52%
Ahtlon MP @ 1.8GHz	1731.705	3.97%
UltraSparcIII @ 900MHz	898.896	0.12%
Power 3 @ 375 MHz	374.860	0.04%

Instruction latencies

- Similar code can be used to determine latencies of other insttructions
- Use #define to define types and instructions as C code
  - For floating-point double addition:

```
#define F_64 double
#define ADD_64(x,y) ((x)+(y))
```

Latency of instructions

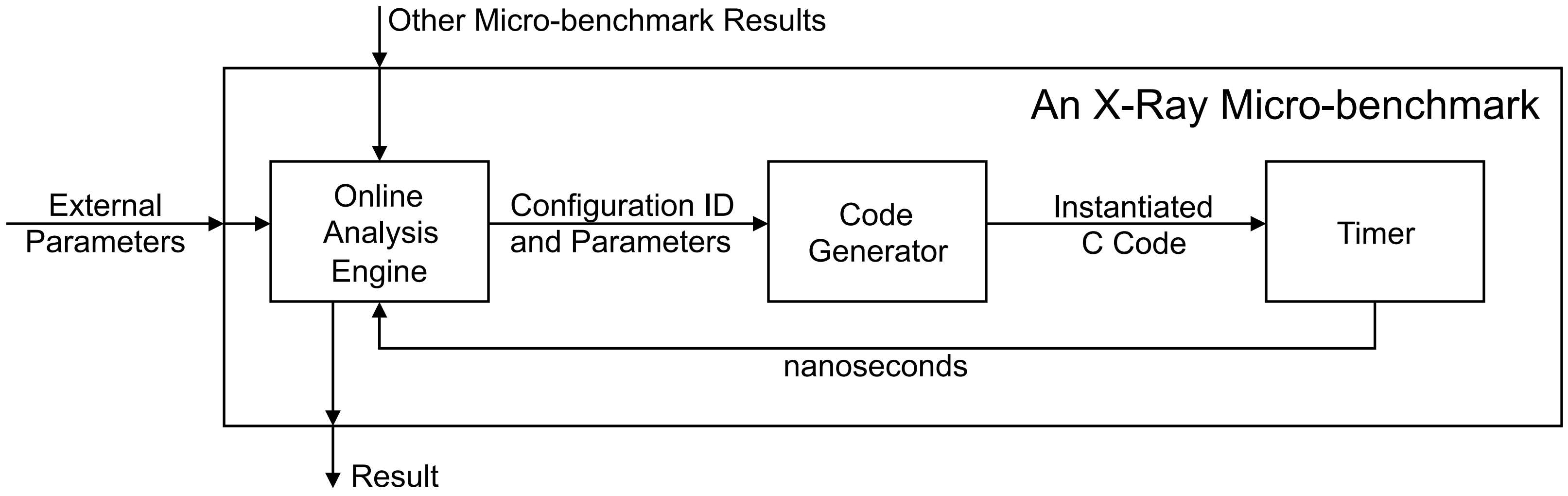
```
volatile int v = 0;
volatile I32 vr = 0;
register I32 r1 = vr;
register I32 r2 = vr;
t = gettime();
i = R / U;
switch (v)
{
    case 0: loop:
    case 1: r1 = ADD_I32(r1, r2);
    case 2: r1 = ADD_I32(r1, r2);
    .................
    case U: r1 = ADD_I32(r1, r2);
    if (--i)
       goto loop;
}
if (!v)
    return gettime() - t;
else
{
    vr = r1;
    vr = r2;
}
```

```
volatile int v = 0;
volatile F64 vr = 0;
register F64 r1 = vr;
register F64 r2 = vr;
t = gettime();
i = R / U;
switch (v)
{
    case 0: loop:
    case 1: r1 = ADD_F64(r1, r2);
    case 2: r1 = ADD_F64(r1, r2);
    .................
    case U: r1 = ADD_F64(r1, r2);
    if (--i)
       goto loop;
}
if (!v)
    return gettime() - t;
else
{
    vr = r1;
    vr = r2;
}
```

Exploit similarity of benchmarks

- Timing shell is identical for both micro-benchmarks
  - Opportunity to reuse code
- Instead of writing a collection of micro-benchmarks
  - implement a micro-benchmark generator
- Input to generator is a configuration
  - Specification of what code should be generated

X-Ray architecture



Measuring Instruction Latency (cont.)

- Build a *Configuration*
  - C=<r1=ADD_F64(r1,r2), F64, 2>
- Execute the *Code Generator* on C
- Measure the *Time* for C
- Compute the result
  - Latency = Frequency * Time

- We will use different configurations to measure different things!

Measuring Throughput

- *Latency* – how often processor can schedule dependent instructions of certain type
- *Throughput* – how often processor can schedule independent instructions of certain type
  - Also *Initiation Interval* (II)
- Intuition

```
r1=ADD_F64(r1, r0)              r1=ADD_F64(r1, r0)
r1=ADD_F64(r1, r0)              r2=ADD_F64(r2, r0)
r1=ADD_F64(r1, r0)              r3=ADD_F64(r3, r0)
r1=ADD_F64(r1, r0)              r1=ADD_F64(r1, r0)
.....................         .....................
r1=ADD_F64(r1, r0)              r3=ADD_F64(r3, r0)
```

- Labels on instructions ensure that compiler does not move or fuse instructions

Experimental Results: Latency and Throughput ADD_F64 and MULTIPLY_F64

Architecture	Add		Multiply	
	L	II	L	II
Pentium III @ 1GHz	3.015	1.015	4.987	2.000
Pentium III @ 500MHz	2.985	1.000	5.046	2.000
Pentium 4 @ 2.4GHz	9.934	2.018	13.93	3.967
Athlon MP @ 1.8GHz	4.000	1.000	4.000	1.000
UltraSparcIII @ 900MHz	4.002	1.000	4.003	1.006
Power 3 @ 375 MHz	4.000	0.500	4.000	0.500

Pentium 4 measured values are double the actual values because measured clock frequency is double actual clock frequency

Power 3 has two ALUs so II is 0.5.

Checking Instruction Existence

- Example: Fused Multiply-Add (FMA)
- **`#define FMA_F64(x,y,z) ((x)+(y)*(z))`**
  - Compilers go around missing instructions
- Key idea
  - Use labels to prevent the compiler from using FMA even if it exists
- Code

```
r1=FMA_F64(r1,r2,r3)                 r0=MULTIPLY_F64(r2,r3)
r1=FMA_F64(r1,r2,r3)                 r1=ADD_F64(r1,r0)
....................         <-->    ......................
r1=FMA_F64(r1,r2,r3)                 r0=MULTIPLY_F64(r2,r3)
r1=FMA_F64(r1,r2,r3)                 r1=ADD_F64(r1,r0)
```

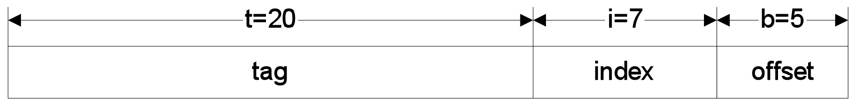
Number of Registers

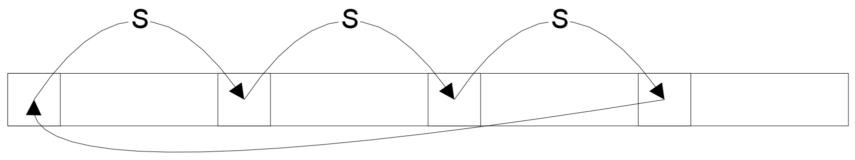
- Plan
  - Write code that uses N variables
  - Performance edge detection
    - Are all N variables register allocated?
- Code (N=4)

```
r0=ADD_F64(r0,r3)
r1=ADD_F64(r1,r0)
r2=ADD_F64(r2,r1)
r3=ADD_F64(r3,r2)
r0=ADD_F64(r0,r3)
..................
```

Memory Hierarchy Features

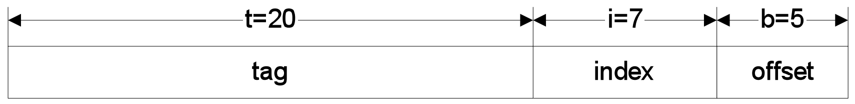
- Walk arrays using different access patterns
  - Traditionally 2^k sized
- Use array of pointers (void*) in order to avoid noise
  - p = *(void**)p;
- Use a single configuration
  - <p=*(void**)p, void *, 1>
- Use different initializations of the array

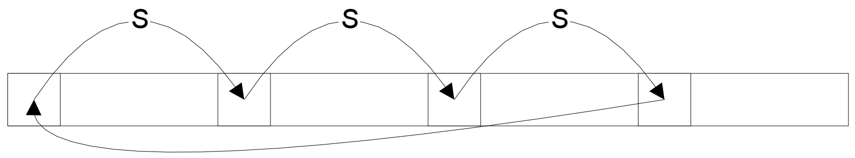




Caches

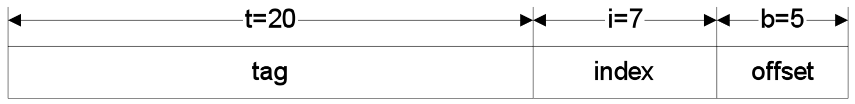
- Parameters
  - Associativity (A)
  - Block Size (B)
  - Capacity (C)
- Stride (S)
  - $S=2^{i+b}$
  - $C=S*A$
- Compact Set
  - Set of addresses, whose data can reside in cache simultaneously

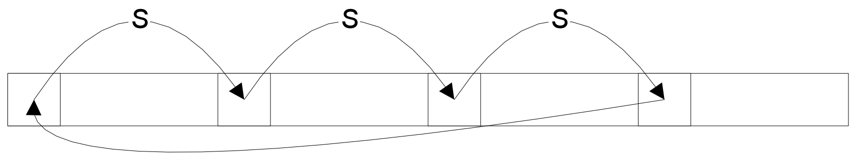




Set Compactness Theorem

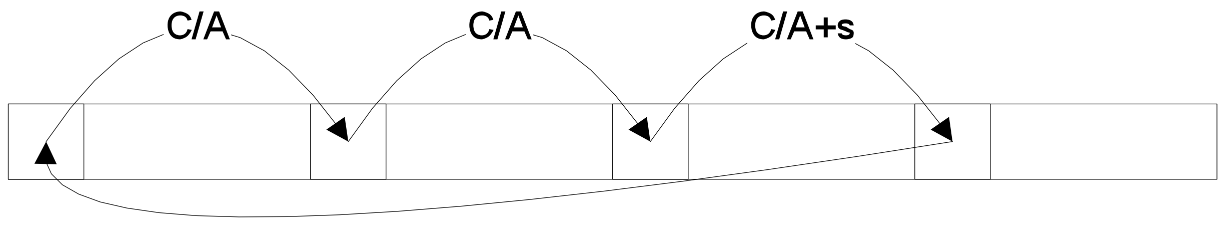
Stride	Count	Compact
2S	A	yes
2S	A+1	no
S	A	yes
S	A+1	no
S/2	2A	yes
S/2	2A+1	no
…		
S/2^k	2^kA	yes
S/2^k	2^kA+1	no





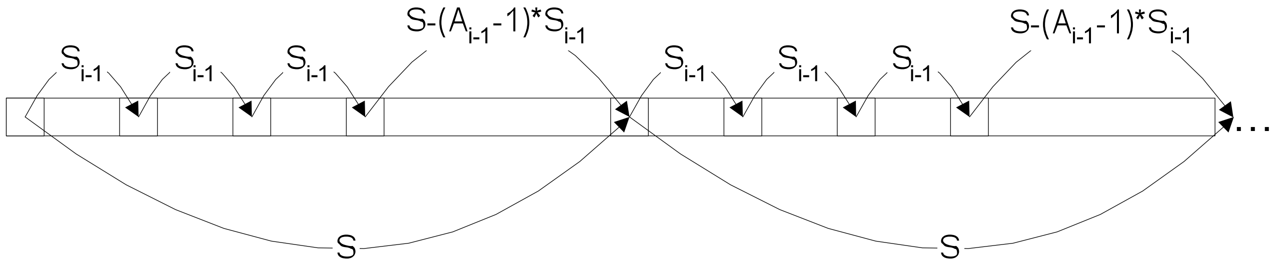
Block Size

- A+1 accesses
  - A with stride S=C/A
  - 1 with stride s
- Compact IFF s<B



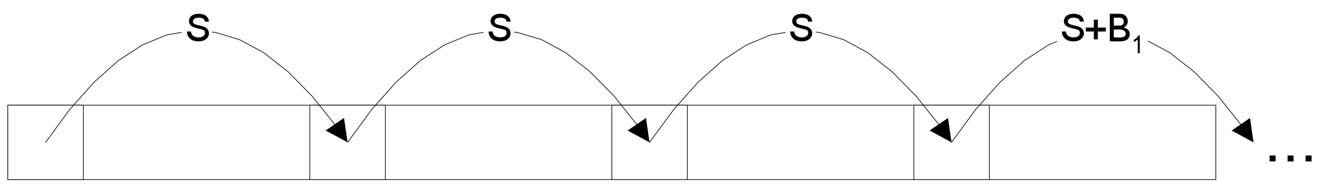
More on Caches

- Deal with hardware prefetching (Power3, Pentium4, etc.)
- Deal with non-inclusion between levels
- Want to measure Ln Cache
  - Need to make sure Ln Cache is accessed
- Non-compact w.r.t. Ln does not generally imply non-compact w.r.t. Ln-1



TLBs

- Goal: Isolated TLB behavior
- Assumptions:
  - L1 Data Cache is at least *Physically Tagged*
  - L1 Data Cache has at least as many blocks as there are page entries in the TLB



Conclusions & Future Work

- Presented precise measurements of many important CPU and Memory Hierarchy properties
- Already in use by our models for BLAS
- Future Work
  - Instruction Cache
  - Out of Order execution
  - Physical Registers
  - Memory Hierarchy
    - Write Mode
    - Replacement Policy
  - Functional Units and (Statement) Scheduling

Functional Units

- Problems
  - CPUs peak pipe utilization is often impossible
  - Hardware bottlenecks prevent sustained peak
    - Number of register read ports
    - Instruction cache throughput
- Try configurations to explore combinations of instructions that can be issued together
- Build sets of “good” configurations and use them for scheduling

SMP / SMT

- Choose a configuration *C* that saturates one type of functional units
  - e.g. The last one tried for Initiation Interval
- SMP
  - Execute increasing number of instances of *C* in parallel (multi-threaded)
  - Stop when no linear speedup
- SMT
  - Execute *C* on both processors
  - Check for 2x speedup