CS184b: Computer Architecture (Abstractions and Optimizations)

Day 11: April 25, 2005
EPIC, IA-64
Binary Translation

Today

- EPIC
- IA-64
- Binary Translation

Scaling Idea

- **Problem:**
  - VLIW: amount of parallelism fixed by VLIW schedule
  - SuperScalar: have to check many dynamic dependencies
- **Idealized Solution:**
  - expose all the parallelism you can
  - run it as sequential/parallel as necessary

Basic Idea

- What if we scheduled an *infinitely* wide VLIW?
- For an N-issue machine
  - for I = 1 to (width of this instruction/N)
    - grab next N instructions and issue

Problems?

- Instructions arbitrarily long?
- Need infinite registers to support infinite parallelism?
- Split Register file still work?
- Sequentializing semantically parallel operations introduce hazards?

Instruction Length

- Field in standard way
  - *pinsts* (from cs184a)
  - like RISC instruction components
- Allow variable fields (syllables) per parallel component
- Encode
  - stop bit (break between instructions)
  - (could have been length…)

Registers

- Compromise on fixed number of registers
  - …will limit parallelism, and hence scalability…
- Also keep(adopt) monolithic/global register file
  - syllables can't control which "cluster" in which they'll run
  - *E.g.* consider series of 7 syllable ops
    - where do syllables end up on 3-issue, 4-issue machine?

Sequentializing Parallel

- Consider wide instruction:
  - `MUL R1,R2,R3  ADD R2,R1,R5`
- Now sequentialize:
  - `MUL R1,R2,R3`
  - `ADD R2,R1,R5`
- Different semantics

Semantics of a "Long Instruction"

- Correct if executed in parallel
- Preserved with sequentialization
- So:
  - read values are from beginning of issue group
  - no RAW hazards:
    - can't write to a register used as a source
  - no WAW hazards:
    - can't write to a register multiple times

Non-VLIW-ness

Register File

- Monolithic register file
- Ports grows with number of physical syllables supported

Bypass

- VLIW
  - schedule around delay cycles in pipe
- EPIC not know which instructions in pipe at compile time
  - do have to watch for hazards between instruction groups
  - ? Similar pipelining issues to RISC/superscalar?
  - Bypass only at issue group boundary
    - maybe can afford to be more spartan?

Concrete Details

(IA-64)

Terminology

- Syllables (their *pinsts*)
- bundles: group of 3 syllables for IA-64
- Instruction group: "variable length" issue set
  - *i.e.* set of bundles (syllables) which may execute in parallel

IA-64 Encoding

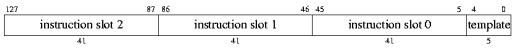


Figure 3-16. Bundle Format

Instruction Type	Description	Execution Unit Type
A	Integer ALU	I-unit or M-unit
I	Non-ALU integer	I-unit
M	Memory	M-unit
F	Floating-point	F-unit
B	Branch	B-unit
L+X	Extended	I-unit

Source: Intel/HP IA-64 Application ISA Guide 1.0

IA-64 Templates

Instruction Type	Description	Execution Unit Type
A	Integer ALU	I-unit or M-unit
I	Non-ALU integer	I-unit
M	Memory	M-unit
F	Floating-point	F-unit
B	Branch	B-unit
L+X	Extended	I-unit

Source: Intel/HP IA-64 Application ISA Guide 1.0

IA-64 Registers

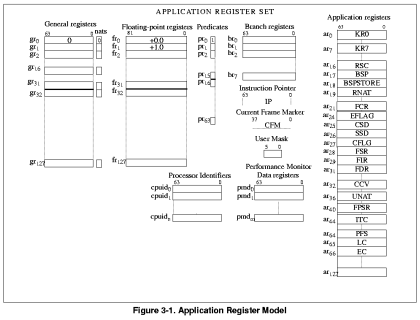


Figure 3-1. Application Register Model

Source: Intel/HP IA-64 Application ISA Guide 1.0

Other Additions

Other Stuff

- Speculation/Exceptions
- Predication
- Branching
- Memory
- Register Renaming

Speculation

- Can mark instructions as speculative
- Bogus results turn into designated NaT
  - (NaT = Not a Thing)
  - particularly loads
  - compare poison bits
- NaT arithmetic produces NaTs
- Check for NaTs if/when care about result

Predication

- Already seen conditional moves
- Almost **every operation** here is conditional
  - (similar to ARM?)
- Full set of **predicate registers**
  - few instructions for calculating composite predicates
- Again, exploit parallelism and avoid loosing trace on small, unpredictable branches
  - can be better to do both than branch wrong

Branching

- Unpack branch
  - branch prepare (calculate target)
    - added branch registers for
  - compare (will I branch?)
  - branch execute (transfer control now)
- sequential semantics w/in instruction group
- indicate static or dynamic branch predict
- loop instruction (fixed trip loops)
- multiway branch (with predicates)

Memory

- Prefetch
  - typically non-binding?
- control caching
  - can specify not to allocate in cache
    - if know use once
    - suspect no temporal locality
  - can specify appropriate cache level
- speculation

Memory Speculation

- Ordering limits due to aliasing
  - don't know if can reorder a[i], a[j]
    - a[j]=x+y;
    - C=a[i]*Z;
  - might get WAR hazards
- Memory speculation:
  - reorder read
  - check in order and correct if incorrect
  - Extension of VLIW common case fast / off-trace patchup philosophy

Memory Speculation

- store(st_addr,data)
- load(ld_addr,target)
- use(target)

- aload(ld_addr,target)
- store(st_adder,data)
- acheck(target,recovery_addr)
- use(target)

Memory Speculation

Before Data Speculation	After Data Speculation

`// other instructions`
`st8 [r4] = r12`
`ld8 r6 = [r8];;`
`add r5 = r6, r7;;`
`st8 [r18] = r5`
	`ld8.a r6 = [r8];; // advanced load`
	`// other instructions`
	`st8 [r4] = r12`
	`ld8.c.clr r6 = [r8] // check load`
	`add r5 = r6, r7;;`
	`st8 [r18] = r5`

Figure 4-2. Data Speculation Recovery Using ld.c

If advanced load fails, checking load performs actual load.

Memory Speculation

Before Data Speculation	After Data Speculation

`// other instructions`
`st8 [r4] = r12`
`ld8 r6 = [r8];;`
`add r5 = r6, r7;;`
`st8 [r18] = r5`
	`ld8.a r6 = [r8];;`
	`// other instructions`
	`add r5 = r6, r7;;`
	`// other instructions`
	`st8 [r4] = r12`
	`chk.a.clr r6, recover`
	`back:`
	`st8 [r18] = r5`
	
	`// somewhere else in program`
	`recover:`
	`ld8 r6 = [r8];;`
	`add r5 = r6, r7`
	`br back`

Figure 4-3. Data Speculation Recovery Using chk.a

If advanced load succeeds, values are good and can continue; otherwise have to execute patch up code.

Advanced Load Support

- Advanced Load Table
- Speculative loads allocate space in ALAT
  - tagged by target register
- ALAT checked against stores
  - invalidated if see overwrite
- At check or load
  - if find valid entry, advanced load succeeded
  - if not find entry, failed
    - reload …or…
    - branch to patchup code

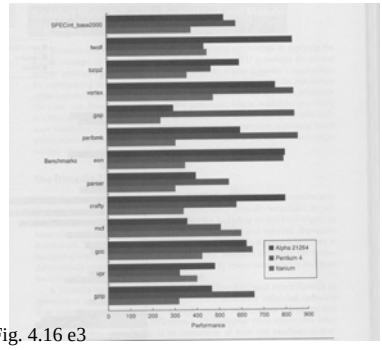
Register “renaming”

- Use top 96 registers like a stack?
- Still register addressable
- But increment base on
  - loops, procedure entry
- Treated like stack with “automatic” background task to save/restore values

Register “renaming”

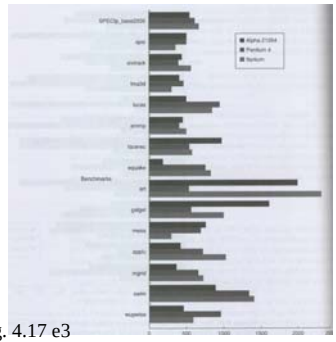
- Application benefits:
  - software pipelining without unrolling
  - values from previous iterations of loop get different names (rename all registers allocated in loop by incrementing base)
    - allows reference to by different names
  - pass data through registers
    - without compiling caller/callee together
    - variable number of registers

Some Data (Integer Programs)



H&P Fig. 4.16 e3

Some Data (FP Programs)



H&P Fig. 4.17 e3

Binary Translation

Skip to ideas

Problem

- Lifetime of programs >> lifetime of piece of hardware (technology generation)
- Getting high performance out of old, binary code in hardware is expensive
  - superscalar overhead…
- Recompilation not viable
  - only ABI seems well enough defined; captures and encapsulates whole program
- There are newer/better architectures that can exploit hardware parallelism

Idea

- Treat ABI as a source language
  - the specification
- Cross compile (translate) old ISA to new architecture (ISA?)
- Do it below the model level
  - user doesn't need to be cognizant of translation
- Run on simpler/cheaper/faster/newer hardware

Complications

- User visibility
- Preserving semantics
  - e.g. condition code generation
- Interfacing
  - preserve visible machine state
  - interrupt state
- Finding the code
  - self-modifying/runtime generated code
  - library code

Base

- Each operation has a meaning
  - behavior
  - affect on state of machin
- stws r29, 8(r8)
  - tmp=r8+8
  - store r29 into [tmp]
- add r1,r2,r3
  - r1=(r2+r3) mod 2^{31}
  - carry flag = (r2+r3>= 2^{31})

Capture Meaning

- Build flowgraph of instruction semantics
  - not unlike the IR (intermediate representation) for a compiler
    - what use to translate from a high-level language to ISA/machine code
  - e.g. IR saw for Bulldog (trace scheduling)

Optimize

- Use IR/flowgraph
  - eliminate dead code
    - esp. dead conditionals
    - e.g. carry set which is not used
  - figure out scheduling flexibility
    - find ILP

Trace Schedule

- Reorganize code
- Pick traces as linearize
- Cover with target machine operations
- Allocate registers
  - (rename registers)
  - may have to preserve register assignments at some boundaries
- Write out code

Details

- Seldom instruction→instruction transliteration
  - extra semantics (condition codes)
  - multi-instruction sequences
    - loading large constants
    - procedure call return
  - different power
    - offset addressing?,
    - compare and branch vs. branch on register
- Often want to recognize code sequence

Complications

- How do we find the code?
  - Known starting point
  - ? Entry points
  - walk the code
  - …but, ultimately, executing the code is the original semantic definition
    - may not exist until branch to...

Finding the Code

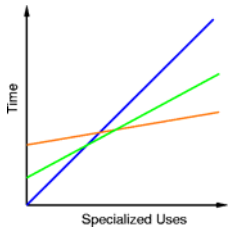
- **Problem:** can't always identify statically
- **Solution:** wait until "execution" finds it
  - delayed binding
  - when branch to a segment of code,
    - certainly know where it is
    - and need to run it
  - translate code when branch to it
    - first time
    - nth-time?

Common Prospect

- Translating code is large fixed cost
  - but has low incremental cost on each use
  - hopefully comparable to or less than running original on old machine
- Interpreting/Emulating code may be faster than "compiling" it
  - if the code is run once
- Which should we do?

Optimization Prospects

- Translation vs. Emulation
  - $T_{trun} = T_{trans} + nT_{op}$
  - $T_{trns} > T_{em\_op} > T_{op}$
- If compute long enough
  - $nT_{op} >> T_{trans}$
  - $\rightarrow$ amortize out load



"Competitive" Approach

- Run program emulated
- When a block is run "enough", translate
- Consider
  - $N_{thresh}\ T_{emop} = T_{translate}$
- Always w/in factor of two of optimal
  - if $N<N_{thresh}$ optimal
  - if $N=N_{thresh}$ paid extra $T_{translate}$ $=2\times$optimal
  - as $N>>N_{thresh}$ extra time amortized out with translation overhead
    - think $T_{translate} \sim = 2T_{translate}$

On-the-fly Translation Flow

- Emulate operations
- Watch frequency of use on basic blocks
- When run enough,
  - translate code
  - save translation
- In future, run translated code for basic block

Translation "Cache"

- When branch
  - translate branch target to new address
  - if hit, there is a translation,
    - run translation
  - if miss, no translation
    - run in emulation (update run statistics)

Alternately/Additionally

- Rewrite branch targets so address translated code sequence
  - when emulator finds branch from translated sequence to translated sequence
  - update the target address of the branching instruction to point to the translated code

Self-Modifying Code

- Mark pages holding a translated branch as read only
- Take write fault when code tries to write to translated code
- In fault-handler, flush old page translation

Precise Exceptions

- Again, want exception visibility relative to simple, sequential model
  - …and now old instruction set model
- Imposing ordering/state preservation is expensive

Precise Exceptions

- Modern BT technique [hardware support]
  - "backup register" file
  - commit/rollback of register file
  - commit on memories
  - on rollback, recompute preserving precise state
    - drop back to emulation?
- …active work on software-only solutions
  - e.g. IBM/WBT'00

Remarkable Convergence?

- Aries: HP PA-RISC→IA-64
  - new architecture
- IBM: PowerPC→BOA
  - ultra-high clock rate architecture? (2GHz)
    - IBM claims 50% improvement over scaling?
    - 700ps = 1.4GHz in 0.18μm
- Transmeta: x86 →Crusoe
  - efficient architecture, avoid x86 baggage

Remarkable Convergence?

- All doing dynamic translation
  - frequency based
- To EPIC/VLIW architectures

Academic Static Binary Translation

Program	Translated Code gcc opt	Translated Code cc opt	Native Code -O0	Native Code -O4
Fibo(40) sec	27.7	28.5	28.6	25.9
bytes	16,512	7,292	16,144	16,152
Sieve(3000) sec	17.8	17.4	18.9	18.6
bytes	16,244	6,548	15,964	15,944
Mbanner(500K) sec	42.5	n/a	80.5	44.8
bytes	22,240		21,524	25,436

Static SPARC to Pentium Translation

[Cifuentes et. al., Binary Translation Workshop 1999]

Academic/Static BT

Program	Translated Code gcc opt	Translated Code cc opt	Native Code -O0	Native Code -O4
Fibo(40) sec	23.0	24.3	41.0	23.0
bytes	24,916	6,680	24,628	24,564
Sieve(3000) sec	26.9	23.9	29.3	24.5
bytes	24,776	6,312	24,552	24,452
Mbanner(500K) sec	53.3	36.9	63.7	26.6
bytes	34,188	21,448	30,652	30,268

Static Pentium to SPARC Translation

[Cifuentes et. al., Binary Translation Workshop 1999]

Academic/Dynamic BT

Test programs	Startup time	Translation time	Without Hot Paths: Execution time without reg caching	Without Hot Paths: Execution time with reg caching	With Hot Paths: Optimisation time	With Hot Paths: Execution time without reg caching	With Hot Paths: Execution time with reg caching	Native gcc compiled
Sieve3000	0.54	0.14	98.25	73.14	0.16	80.98	66.29	29.22
Fibonacci	0.54	0.10	186.17	154.97	0.11	147.56	133.69	41.18
mbanner	0.52	0.34	219.01	146.28	0.37	146.22	126.28	22.85

Table 1: Pentium to SPARC translation (second)

[Ung+Cifuentes, Binary Translation Workshop 2000]

Big Ideas [EPIC]

- Compile for maximum parallelism
- Sequentialize as necessary
  - (moderately) cheap

Big Ideas [IA-64 1]

- Latency reduction hard
  - path length is our parallelism limiter
  - often good to trade more work for shorter critical path
    - area-time tradeoff
  - speculation, predication reduce path length
    - perhaps at cost of more total operations

Big Ideas [IA64 2]

- Local control (predication)
  - costs issue
  - increases predictability, parallelism
- Common Case/Speculation
  - avoid worst-case pessimism on memory operations
  - common case faster
  - correct in all cases

Big Ideas [Binary Trans]

- Well-defined model
  - High value for longevity
  - Preserve semantics of model
  - How implemented irrelevant
- Hoist work to earliest possible binding time
  - dependencies, parallelism, renaming
  - hoist ahead of execution
    - ahead of heavy use
  - reuse work across many uses
- Use feedback to discover common case