ΕΠΛ605: Προχωρημένη Αρχιτεκτονική Υπολογιστών

Γιάννος Σαζειδης

Κεφ. 3 και Appendix H: Static ILP
Static (Compiler Based) Scheduling

Διαβάστε κεφ. 3 και H

Εαρινό Εξάμηνο 2017

Today's Theme and Contents

- Let compiler uncover the ILP
  - Objective:more ilp possibly simpler hardware/faster clock/less power
  - Static ilp can be useful for dynamically scheduled processors
- How:
  - Static (Local) Scheduling
  - Loop Unrolling
- Processor Architecture for Statically scheduled Multiple Issue: VLIW
- IA-64 and Itanium

How to uncover the ILP

- Violate program order and control dependences
- But maintain correctness
  - same dataflow and exception behavior
- Do above effectively AND efficiently!
- Dynamically (hardware based approach)
- Statically (more compiler control – may be some hardware help and support from ISA)

Basic Idea

- The compiler moves/rearranges dependent instructions apart to avoid hazards
- This means:
  - such instructions exist (if not there employ transformations)
- Static ILP applicable to *statically** and dynamically scheduled processors
  - May help simplify hardware to find parallelism (ex. in-order superscalar)
  - the compiler knows implementation details
    - » latency AND superscalarity (issue width)
- What happens if implementation changes?
  - Correctness preserved but may be less benefits
- **Statically scheduled processors: the compiler dictates which instructions can execute together (scheduling done in software) – ISA support*

Compiler Techniques for Exposing ILP

- Pipeline scheduling
  - Separate dependent instruction from the source instruction by the pipeline latency of the source instruction
  - assume simple scalar pipeline
  - If something not ready stall

- Example:

```
for (i=999; i>=0; i=i-1)
  x[i] = x[i] + s;
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Pipeline Stalls

```
Loop:   L.D      F0,0(R1)
        stall
        ADD.D F4,F0,F2
        stall
        stall
        S.D F4,0(R1)
        DADDUI R1,R1,#-8
        stall (assume integer load latency is 1)
        BNE R1,R2,Loop
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Pipeline Scheduling

(Local Scheduling within a BB)

Scheduled code:

```
Loop:   L.D      F0,0(R1)
        DADDUI R1,R1,#-8
        ADD.D F4,F0,F2
        stall
        stall
        S.D F4,8(R1)
        BNE R1,R2,Loop
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Loop Unrolling

- Loop unrolling
  - Unroll by a factor of 4 (assume # elements is divisible by 4)
  - Eliminate unnecessary instructions

```
Loop:   L.D F0,0(R1)
        ADD.D F4,F0,F2
        S.D F4,0(R1) ;drop DADDUI & BNE
        L.D F6,-8(R1)
        ADD.D F8,F6,F2
        S.D F8,-8(R1) ;drop DADDUI & BNE
        L.D F10,-16(R1)
        ADD.D F12,F10,F2
        S.D F12,-16(R1) ;drop DADDUI & BNE
        L.D F14,-24(R1)
        ADD.D F16,F14,F2
        S.D F16,-24(R1)
        DADDUI R1,R1,#-32
        BNE R1,R2,Loop
```

Unroll body of loop several times
How many times???

- note: number of live registers vs. original loop

Loop Unrolling/Pipeline Scheduling

- Pipeline schedule the unrolled loop:

```
Loop:   L.D F0,0(R1)
        L.D F6,-8(R1)
        L.D F10,-16(R1)
        L.D F14,-24(R1)
        ADD.D F4,F0,F2
        ADD.D F8,F6,F2
        ADD.D F12,F10,F2
        ADD.D F16,F14,F2
        S.D F4,0(R1)
        S.D F8,-8(R1)
        DADDUI R1,R1,#-32
        S.D F12,16(R1)
        S.D F16,8(R1)
        BNE R1,R2,Loop
```

Strip Mining

- Unknown number of loop iterations?
  - Number of iterations = *n*
  - Goal: make *k* copies of the loop body
  - Generate pair of loops:
    - » First executes *n* mod *k* times
    - » Second executes *n* / *k* times
    - » "Strip mining"

Trace Scheduling [Ellis 1985]

trace scheduling -

- originally for VLIW architectures, now superscalars also

trace scheduling will work for non-loop situations

takes most common paths in program and schedules instrs

Software speculation/Global Scheduling

© 1999 by Hill, Wood, Sohi, Smith, and Vijaykumar

Trace Scheduling

```
b[i] = "old"
a[i] =
if (a[i] > 0)  then
    b[i] = "new";     common case
else
    X
endif
c[i] =
```

© 1999 by Hill, Wood, Sohi, Smith, and Vijaykumar

Trace Scheduling: Top Level Algorithm

until done

- select most common path - called a trace **HOW??**
- schedule trace across basic blocks
- basic block -
  - code block with single entry point, single exit point
- repair other paths

Static prediction, profile, frequency, path
Which is better the above or dynamic prediction?

© 1999 by Hill, Wood, Sohi, Smith, and Vijaykumar

Trace Scheduling

trace to be scheduled:

```
        b[i] = "old"
        a[i] =
        b[i] = "new"
        c[i] =
        if (a[i] <=0) goto label1
label2:
```

repair code

```
label1:
        restore old b[i]
        X
        recalculate c[i]?
```

© 1999 by Hill, Wood, Sohi, Smith, and Vijaykumar

Static Scheduling: Summary

loop unrolling

- \+ large block to schedule
- \+ reduces branch frequency
- – expands code size
- – have to handle "extra" iterations

Register pressure

Static Scheduling: Summary

trace scheduling

+ works for non-loops

– more complex than unrolling

– does not seem to handle more general cases

- Predicting dependences accurately

- Code pressure

© 1999 by Hill, Wood, Sohi, Smith, and Vijaykumar

Maybe dependences

maybe (ambiguous) memory dependences

e.g.,

```
*ptr1 =                 ----  store instr
 tmp = *ptr2            ---- load instr
 add (tmp+4)            ---- will cause stalls due to
                        ---- dependence on tmp
```

© 1999 by Hill, Wood, Sohi, Smith, and Vijaykumar

Maybe dependences

Is a dependence possible?

```
for (i=0; i<=100; i++)
     A[a * j + b] = A [c * k + d]
```

if dependence exists then GCD(c,a) must divide (d-b)

e.g.,

```
for (i=0; i<=100; i++)
     A[2i+3] = A [2i] + 4;
```

a = 2, b = 3, c = 2, d = 0, GCD (a,c) = 2 and d-b = -3.

so no dependence

- Software speculation for memory dependences
- Possible same as sw based control speculation
- problem are exceptions from instructions not in program order (same for control sw speculation)

Software vs. Hardware

equivalent techniques, differ in applicability

hardware

+ high branch prediction accuracy
+ has dynamic information on latencies like cache misses
+ works for generic, non-loop, irregular code
    - e.g., databases, desktop applications, compilers

– limited reorder buffer size - limited "lookahead"

– high cost/complexity

© 1999 by Hill, Wood, Sohi, Smith, and Vijaykumar

Software vs. Hardware

software

+ can look at large amounts of code - large "lookahead"
+ no hardware cost
+ works for regular code - "fortran codes"
  - e.g., engineering applications, weather prediction
– low branch prediction accuracy - can improve by profiling

Does not have dynamic information on latencies like cache misses

- run code once to figure branches/cache misses
- use a different input, not real input

© 1999 by Hill, Wood, Sohi, Smith, and Vijaykumar

Software vs. Hardware

How did hardware do all our software examples?

unrolling

- branch prediction, renaming

trace scheduling

- prediction + renaming + reorder buffer + squashes
- trace cache

code from the past

© 1999 by Hill, Wood, Sohi, Smith, and Vijaykumar

Hardware/SoftwareTradeoffs

hardware scheduling

+ uses runtime info => increased ILP, flexibility
+ complicated hardware
+ limited scope for finding ILP

software scheduling

- uses only compile-time info
- simple hardware
- broader scope of finding ILP

© 1999 by Hill, Wood, Sohi, Smith, and Vijaykumar

Hardware/SoftwareTradeoffs

Mitigating issues for hardware

compiler can still do higher level scheduling

will hardware control slow clock?

The various compiler techniques allow to violate program order and/or control flow dependences without violating data flow

Applicable to dynamically ooo processors
But more useful for dynamic in-order processors

Static ILP issue: how to deal with imprecise exceptions when using speculation

Statically scheduled processors? (different ISA, uarch)

VLIW: All Software

very long instruction word

implement a number of independent functional units

provide a long instruction word with one operation per FU

instruction latencies are fixed

compiler packs independent instructions into VLIW

- compiler schedules all hardware resources

entire long word issues as a “unit”

result: ILP with simple hardware, simple control, fast clock

LockStep: any hazard stall / NOPs if not enough //ism

© 1999 by Hill, Wood, Sohi, Smith, and Vijaykumar

VLIW: Software

code scheduling in software only

loop unrolling, software pipelining, trace scheduling

architectural support

- deferred interrupts
  - enhanch scheduling opportunities
- predicated execution - e.g., conditional moves
  - less need for hardware prediction
- more registers
  - renaming less important

© 1999 by Hill, Wood, Sohi, Smith, and Vijaykumar

Predicated Execution & Conditional Moves

Convert control dependences to data dependences

if (a=0) s=t; a-R1 s-R2 t-R3

```
  bnez    R1,L
  addu    R2,R3,0
L:
```

```
cmovz R2,R3,R1
```

Above for all itypes is called predication…

Ax+(1-x) < B => x < (B-1)/(A-1)

A misprediction penalty

B latency to execute both paths

+/-?

```
bnz r1, L1
lw r2,0(r3)
add r2,r2,1
sw r2,0(r3)
j L2
L1:
lw r2,0(r4)
add r2,r2,-1
sw r2,0(r4)
L2:
```

```
cz.lw r2,0(r3), r1
cz.add r2,r2,1, r1
cz. sw r2,0(r3), r1
cnz.lw r2,0(r4), r1
cnz.add r2,r2,-1, r1
cnz.sw r2,0(r4), r1
```

$3x + 9(1-x) \; ? \; 6$

$9-6x \; ? \; 6$

$3 < 6x$

$0.5 < x$

Speculative Loads

Bypass stores speculative - repair code in case of mispeculation
Use an address buffer

1. LookUp Table: updated by address of speculative load

2. Updated by addresses of intervening stores

3. Check instruction: that no store conflicted and release Entry

check instruction is inserted at the place of original instruction

IA64 - Based on VLIW (EPIC) Architectural Approach

- Registers
  - 128 x 64 bit GPR
  - 128 x 82 bit FPR
  - 64 x 1 bit predicates
  - 8 x 64 bit Branch Registers
  - system registers
  - Register Stack Engine
- Instructions
  - Bundle - 3 instructions (total 128 bits)
  - 5 bit template and 3x41 instructions
- Instruction Group
  - sequence of independent instructions (can be as long as it needs but ends with a stop bit)
  - stop bit part of template
- 5 instructions classes: Alu, Ioalu, Move, Fp και Br

IA64 - Based on EPIC Architectural Approach

- Predication
  - many predicate registers
  - compare instructions that set two predicates
  - almost all instructions can be predicated
- Speculation
  - control and data (memory)
  - deferred exceptions
    - » exception flag propagated
    - » either caught by a non-speculative check instruction or a store
    - » two types of checks for memory speculation
      - Reload (idempotent), jump to fixup routine

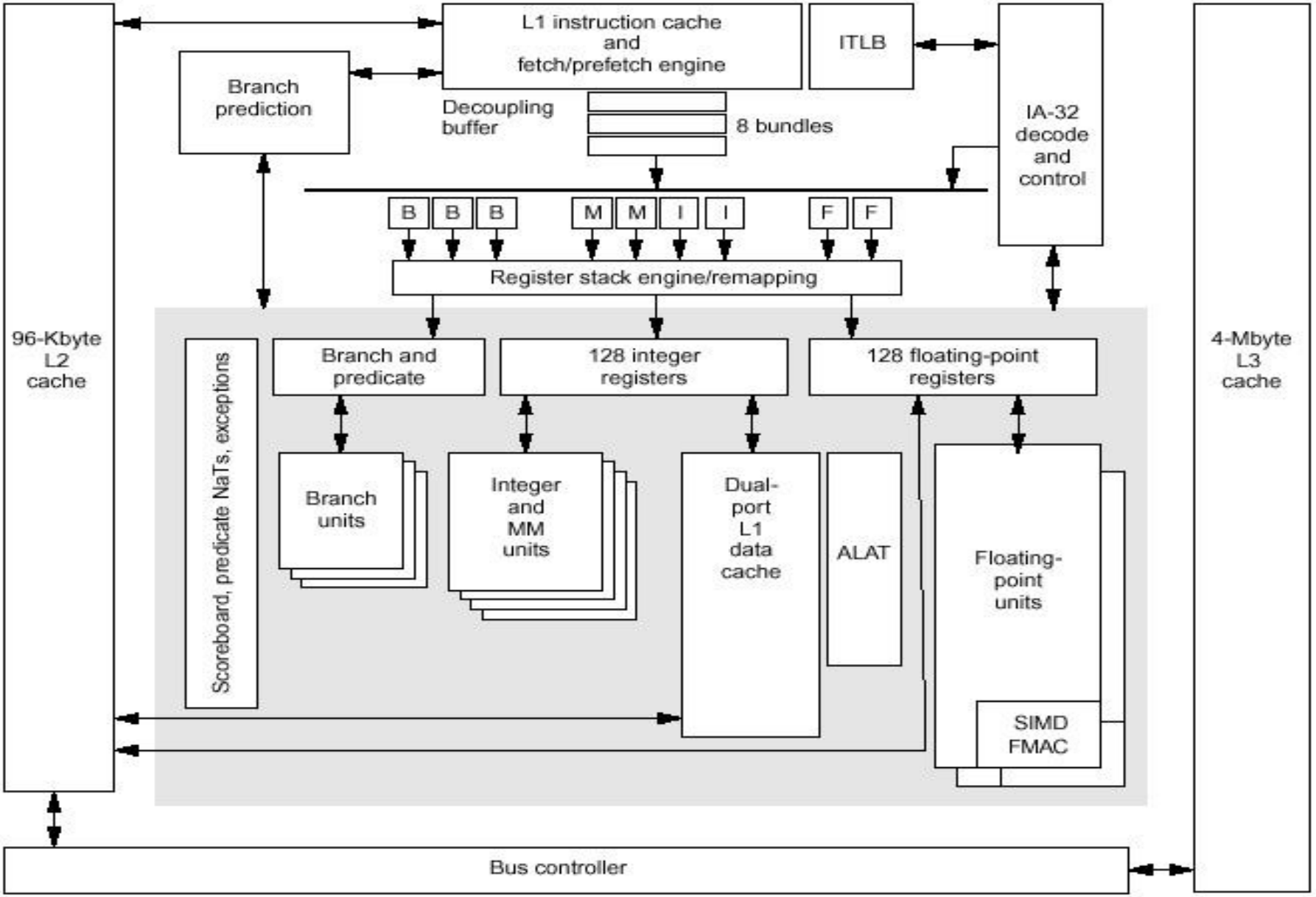
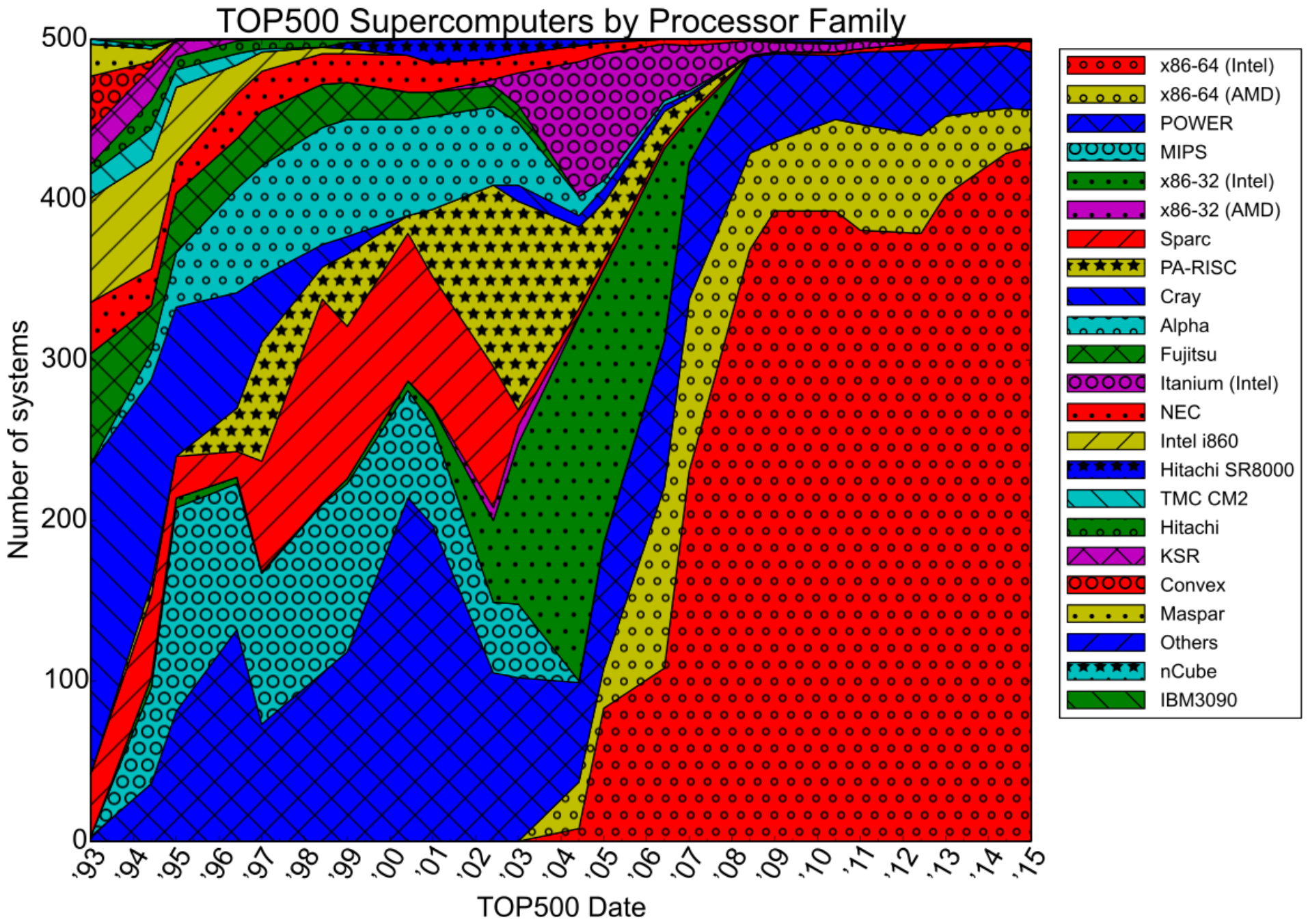


Figure 4. Itanium processor block diagram.


TOP500 Supercomputers by Processor Family
Number of systems
0
100
200
300
400
500
'93
'94
'95
'96
'97
'98
'99
'00
'01
'02
'03
'04
'05
'06
'07
'08
'09
'10
'11
'12
'13
'14
'15
TOP500 Date
x86-64 (Intel)
x86-64 (AMD)
POWER
MIPS
x86-32 (Intel)
x86-32 (AMD)
Sparc
PA-RISC
Cray
Alpha
Fujitsu
Itanium (Intel)
NEC
Intel i860
Hitachi SR8000
TMC CM2
Hitachi
KSR
Convex
Maspar
Others
nCube
IBM3090

Conclusion/Future

- Static ILP very useful transformations to increase performance
  - Applicable to all processors
  - More beneficial to simple processors and statically scheduled (VLIW)
- VLIW
  - For general purpose: does not seems work
  - Works well for scientific
  - For embedded: big market
- Other compiler trends:
  - Dynamic Compilation/Optimization
  - Virtual Machines (online optimization)
  - Emulation of instructions