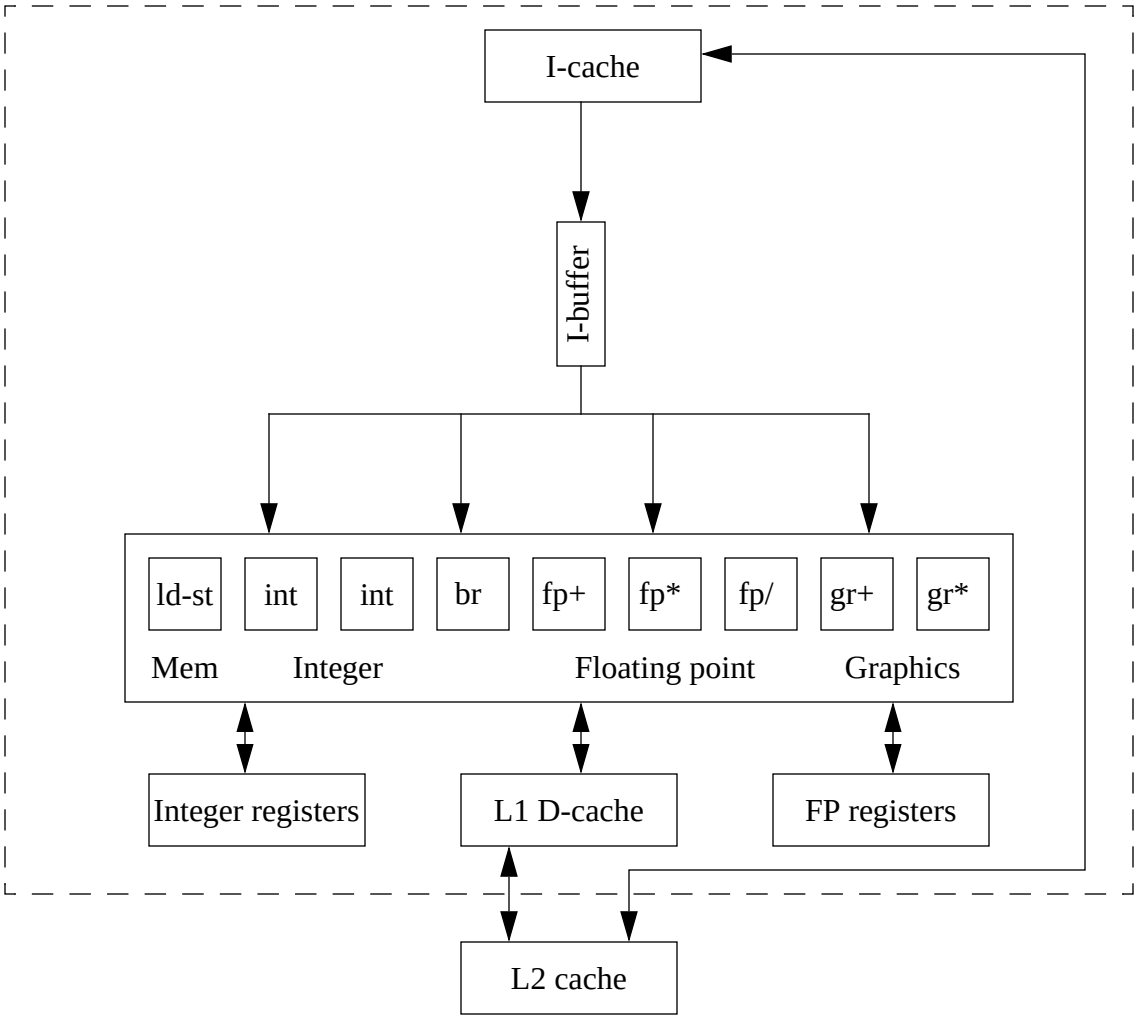
A Parallelizing Compiler for UltraSPARC

Chris Aoki
Peter Damron
Kurt Goebel
Vinod Grover
Xiangyun Kong
Michael Lai
Krishna Subramanian
Partha Tirumalai
Jian-Zhong Wang

Sun Microsystems, Inc.
Mountain View, CA 94043

Part I: The Uniprocessor



Simple model of the UltraSPARC processor

UltraSPARC architectural features

- **Four-way superscalar with nine independent functional units**
- **64-bit V9 architecture**
- **16 kb 2-way instruction cache**
- **16 kb non-blocking direct mapped data cache**
- **9-entry load buffer/8-entry store buffer**
- **Separate 64 entry, fully associative instruction and data TLB's**

Latency and throughput for common instructions

	Latency	Pipelined?	Max. issue
Load	**2 (L1), 8 (L2)**	**Yes**	**1**
Int ALU	**1**	**Yes**	**2**
FP add, mul	**3**	**Yes**	**2 (1 +, 1*)**
FP div, sqrt	**12 (sp), 22 (dp)**	**No**	**1**

Load buffer and non-blocking L1 cache

- **Fully pipelined access to the off chip cache**
- **Allows some codes to operate out of the large L2 cache**
- **Compiler must schedule loads and uses sufficiently apart**
- **Long latencies +High scalarity -------> compiler must extract a lot of ILP**

ld			
use			

UltraSparc: Four issue, latency to L2 = 8 clocks ==> ~31 independent instructions have to be found and scheduled between loads and uses, to operate at the peak rate out of L2

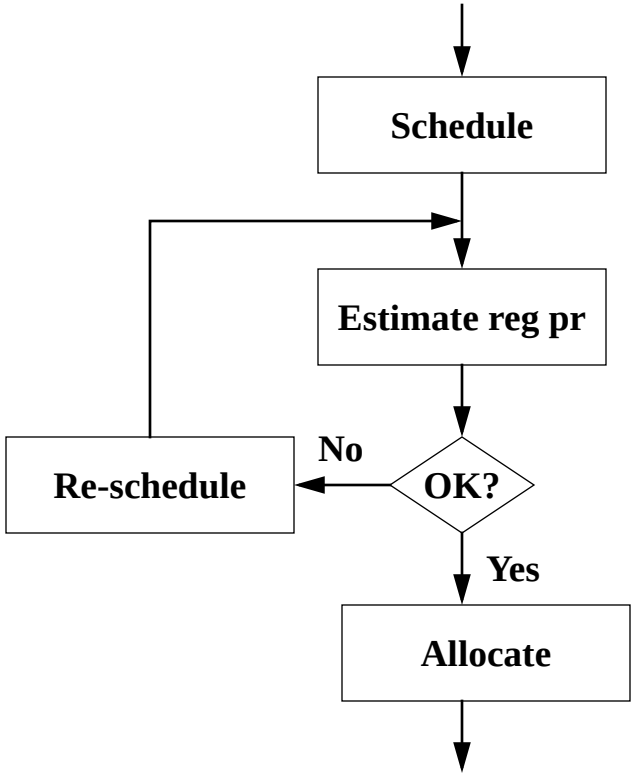
Inner loops are scheduled to operate out of L2.

Inner Loop Scheduling

- **Software pipelining (modulo scheduling) is used**
- **Processor is modelled as a VLIW**
- **Bounds based scheduler - limits imposed by resources and inter-iteration deps**
- **Performs a number of loop optimizations for inner loops**
- **Handles machine specific “rules”**
- **> 1300 inner loops scheduled in SPEC’92**
- **> 90% produce schedules that saturate resource/dependence limits**
- **> 99% are within 10% of the resource/dependence limits**
- **Most loops operate at or close to the predicted rate on the system**

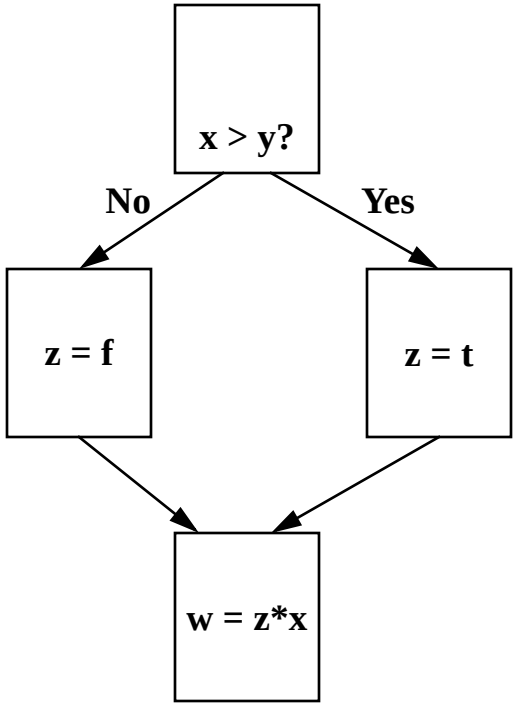
Scheduling and register allocation

- **Scheduler targets high performance initially (register pressure is a secondary metric)**
- **If the pressure is then *estimated* to be high, the schedule is relaxed**
- **Statistics show that the final allocation is very close to the estimated register requirement**
- **Allocator also handles machine specific rules (e.g. ld [addr],%f0 locking both %f0 and%f1)**
- **Less than 10 cases of spills in SPEC'92 loops**
- **Less than 20 schedules compromised due to high register pressure**



Multiple condition codes and conditional moves

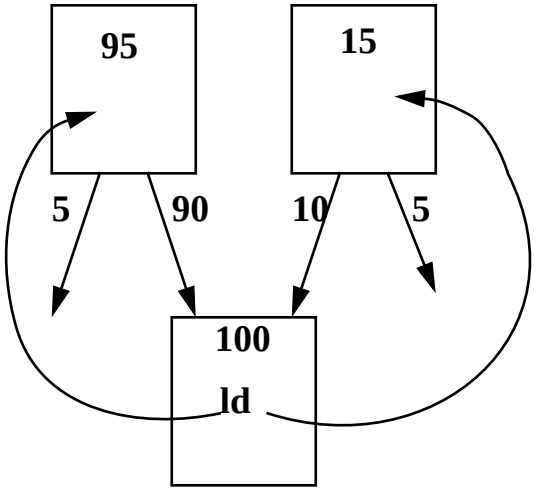
- **Four floating point condition codes are available in SPARC V9**
- **Allows multiple comparisons to execute simultaneously**
- **Can eliminate some branches**
- **Improved scheduling of branch free code**
- **Useful for min, max and other simple cases (e.g. hydro2d)**

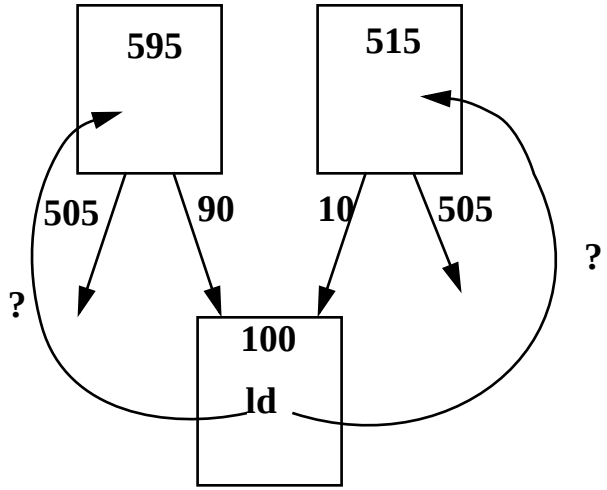



```
(x>y) -> fcci
movgt fcci,t,z
movle fcci,f,z
z*x -> w
```

Non-faulting loads

- **Permits loads to be executed early thereby covering load latency**
- **Relies on code motion (guided by profile information)**
- **Compiler must evaluate profitability, manage register pressure and disambiguate memory references to obtain maximum benefit**





VIS/Multimedia instructions

- **Look like functions to the programmer**
- **Compiler allocates registers and schedule instructions**
- **All backend optimizations are performed including modulo scheduling**

```
for (i = 0; i < times; ++i) {
   dsrc1 = s1[i];
   dsrc2 = s2[i];

   dadd10 = vis_fexpand(vis_read_hi(dsrc1));
   dadd11 = vis_fexpand(vis_read_lo(dsrc1));
   dadd20 = vis_fexpand(vis_read_hi(dsrc2));
   dadd21 = vis_fexpand(vis_read_lo(dsrc2));

   resu = vis_fpack16(vis_fpadd16(dadd10, dadd20));
   resl = vis_fpack16(vis_fpadd16(dadd11, dadd21));
   dres = vis_freg_pair(resu, resl);

   ((vl_d64 *) da_align)[i] = dres;
}
```

```
.L900000142:
     fpadd16%f34,%f32,%f36
     add  %g4,2,%g4
     add  %o5,16,%o5
     fexpand%f9,%f32
     cmp  %g4,-1
     add  %o1,16,%o1
     ldd  [%o7],%f8
     fexpand%f5,%f34
     add  %o7,16,%o7
     ldd  [%o1-16],%f4
     fpack16%f36,%f31
     fpadd16%f32,%f34,%f34
     fexpand%f6,%f32
     fpack16%f34,%f31
     fexpand%f2,%f34
     std  %f0,[%o5-16]
     fpadd16%f32,%f34,%f36
     fexpand%f7,%f32
     .........
```

Block load and store instructions

- **Can load (store) 64 bytes of data into (from) 8 double registers with one instruction**
- **Does not pollute the caches**
- **Useful for data initializations, copy etc.**
- **Also used in libraries**

Part II: Compiling for Multiprocessor Systems

- **Automatic Parallelization**
  - Loop Level (arbitrary nests)
  - Targets shared-memory MP systems
  - Scientific Codes
  - Parallelization Directives
    - Loop Level
    - Scheduling Control
    - Variable Types
  - Extensions for Aliasing
    - restrict keyword
    - no side effect pragma

- **Loop Transformations**
  - Interchange
  - Fusion
  - Reductions (scalar and array)
  - Vector code generation

Automatic Parallelization

- **Based on dependence analysis**

- **Profitability Analysis**

  - Estimate the amount of work performed by the loop nest (uses profile information if available)
  - Need about 2500 cycles worth of work for break-even on Ultra systems
  - Generate multi-version or straight parallel code

Automatic Parallelization (contd.)

- **An example of parallel code generation**

```
do i = 1, n
     a(i) = a(i) + b(i)*c(i)
end do
```

```
void pfunction(ap,bp,cp,l,u)
     double *ap, *bp, *cp;
     int l, u
{
int j
for (j = l; j <=u; j++) {
     ap[j] = ap[j] + bp[j]*cp[j];
}
}
```

- The pfunction executed by multiple threads managed by the runtime library.
- The work distribution performed by the master thread
- The user and the compiler could control the work distribution strategy
- The chunk loop in the pfunction can be pipelined
- The transformation is done at intermediate level not source level

Loop Transformations

- **Loop Interchange**

  Maximize the parallelism and data locality of the parallelized loop nest

  Transformation done before parallel code generation and on the residual loop in the parallel function

- **Loop Fusion**

  Improves granularity of loops

  Improves the cache behaviour of loops

  e.g. wave5 gets speedup because of this (approximately 8 loops are fused together)

- **Array Privatization**

  Based on region summary and data flow analysis in loop nests

  Each instance of the pfunction gets its own copy of the array

  Very effective on user codes and on some benchmarks in NAS suite

- **Array Reduction**

  Similar to array privatization (based on same analysis)

  Each pfunction gets a separate copy of the array

  e.g. mdljsp2 and mdljdp2 in spec92, and several NAS benchmarks

Loop Transformations (contd.)

- **Scalar Replacement**

  Replaces loads with scalar temporaries
  Works within and across loop iterations
  Can also dead-code-eliminate array stores that are used as temps
  Can also optimize use-to-use array references.

- **Vectorization**

  Based on data dependence analysis
  Discovers and isolates vector trancendental functions in loop nests
  Allocates scalar variables to vector temporaries and inserts calls to the vector library
  Vector library implemented using VIS functions on Ultra systems
  Very effective on certain scientific and commercial applications

Conclusions

- **Compiler exploits parallelism at all levels**

- **Very effective on a wide variety of applications**
  - **Scientific codes scheduled to operate out of the external cache**
  - **> 90% of 1300 inner loops reach resource/dependence limits**
  - **1.25X, 1.4X and 1.5X speedup on SPEC’92 with 2, 3 and 4 processors**
  - **Large, parallelized programs show nearly linear speedup**

- **Optimization algorithms developed in conjunction with hardware design**