Maximizing the Benefits of CMT with Sun's Compilers and Tools

Partha Tirumalai
Distinguished Engineer
Scalable Systems Group, Sun Microsystems

Multicore Expo
Santa Clara, CA, USA
March 21-23, 2006
Introduction to CMT
What is CMT?

- Chip-MultiThreading (CMT) refers to a processor design that allows a single silicon chip to simultaneously execute more than one software thread (instruction stream).

- CMT includes:
  - Multi-core designs
  - Multi-threaded designs
    - Vertical, Horizontal, or other forms of threading in a core
  - Combinations of the above
    - I.e., multiple multi-threaded cores on one chip
Why CMT? - The “old” approach

- Microprocessors have focused on improving single thread performance for the last 25 years
  - Pipelined functional units
  - Multiple-instruction issue
  - Out-of-order execution
  - Hardware prefetching
  - Large, complex cache hierarchies

- Technology trends have made this approach increasingly difficult
  - Complex designs, low efficiency (perf/$, perf/W, ...)

- Complex designs, low efficiency (perf/$, perf/W, ...)

3-issue, 4-stage pipe
Up to 12 ops in flight
Very low efficiency
Rarely operates at peak
Modest CMT Designs

- Largely retain focus on extracting performance from a single thread of execution
  - Use Moore's law to put 2 cores on 1 die
  - Add incremental features like Simultaneous Multi-threading

Benefits

- Leverages investment in old cores already designed
- Good single thread performance
- Some (but limited) gain on multi-threaded workloads
  - Use large SMP's or clusters to handle more threads
- Examples: US IV, Dual-core Opteron
Aggressive CMT Designs

- Reduce focus on single thread performance
  - Recognize memory accesses as the chief bottleneck and tolerate these by exploiting thread level parallelism (TLP)
- Design from scratch for multiple parallel threads
  - Zero or very low thread-switch overhead
  - High associativity, high bandwidth on-chip cache
  - Very high bandwidth to memory
- Benefits
  - Excellent throughput and efficiency (perf/$, perf/W)
- Examples: US T1
CMT Outlook

- Every major vendor in the industry is working on CMT designs
- CMT is expected to be ubiquitous in the near future
  - Multiple threads even in laptops
    - In Apple's MacBook Pro Now!
  - Threads – cheap, everywhere, for everyone
CMT - Synergies Beyond the Chip

- **Hardware**
  - Adequate cache/memory, I/O, and networking bandwidth, plus RAS for large, parallel workloads

- **Operating System**
  - Reliable and scalable OS for optimal management of parallel threads

- **Developer Tools**
  - Compilers and tools to make application development easy and efficient

Focus of this talk: Sun Studio 11 C/C++/Fortran Compilers & Tools
Sun Studio 11 Compilers and Tools

developers.sun.com/sunstudio

Free!  SPARC  Solaris
Sun Studio 11 – From 50K Feet

- Reliable
  - >100M lines of tests routinely run on compute farm
    - Huge, complex, mission-critical applications are built with our compilers

- Standards adherence
  - At the forefront of industry standards
    - C99, IEEE floating point, OpenMP ...

- Advanced optimizations, easy to use
  - Deliver high performance on a wide spectrum of codes
    - Tuned to the latest hardware
Compiler Components

Front End

Machine Independent Optimizer

Optimizing Code Generator

Application

Memory

Caches

CPUs

Tools (Analyzer, Debugger, ...)

Ride the CMT Wave with Sun's Compilers/Partha T.

Multicore Expo, March 21-23. 2006
Optimization & Commentary
What did it do to my code?

- The compiler commentary explains how the source code was optimized
- Build with “-g” added (does not disable optimizations)
- Get commentary with er_src command
- See documentation for details

Improves understanding and helps user optimize

- User can derive hints on further options to use (or not use)
- User can derive hints on adding pragmas that might help
- User can derive hints on what reorganization might help
Example 1 – Loop Scheduling

for (j=1; j<n; j++)
    a[j] = a[j-1] + 4.0*b[j]*c[j] +
          b[j]*b[j] + c[j]*c[j] + 6.0;

L-unknown scheduled with steady-state cycle count = 4
L-unknown unrolled 4 times
L-unknown has 2 loads, 1 stores, 3 prefetches, 4 FPadds, 3 FPmuls, and 0 FPdivs per iteration
L-unknown has 0 int-loads, 0 int-stores, 5 alu-ops, 0 muls, 0 int-divs and 0 shifts per iteration
Source loop below has tag L1
    7. for (j=1; j<n; j++)
    8.    a[j] = a[j-1] + 4.0*b[j]*c[j] +
           b[j]*b[j] + c[j]*c[j] + 6.0;

- 1 load eliminated
- 1 fpmul eliminated
- unrolled 4 times
- optimally scheduled
- resource limit = 4
- dependence limit = 4
- achieved schedule = 4
- 3 prefetches inserted
Example 2 – IPO, Pointers, IF's

```c
void propagate(int *p, int *q, int *r) {
    int x;
    ...
    x = *p;
    ...
    if (x < 50) {
        ...
        split(p,q);
        ...
        if (x < 100) {
            merge(q,r);
        }
    }
    ...
```

- Note x is a local variable
- If it can be proved that:
  - cond1 => cond2
  - x is not modifiable in split
  - x is not modified in r1
  - x is not modified in r2
- then:
  - the second if is eliminated

- Involves
  - Pointer analysis
  - Inter-procedural analysis
  - Conditional relationships

- Complexity and “code rot” can cause such scenarios
- Second conditional optimized away by the compiler
Example 3 – Profile Feedback

```c
switch (var) {
    case 1: ...; break;
    case 2: ...; break;
    ...
    case 47: ...; break;
    ...
    case 59: ...; break;
    ...
    case 250: ...; break;
}
```

- Large switch statement (many cases)
- Without profile feedback
  - Jump table generated
- With profile feedback
  - Data shows a couple of hot cases
  - Other cases are cold
  - Test these two cases first
  - Then use a jump table
  - Results in
    - Faster dispatch of hot cases
    - Hot code uses I-cache better

```
$ cc -fast -xprofile=collect main.c process.c ...
$ a.out
$ cc -fast -xprofile=use main.c process.c ...
$ a.out
```
Example 4 – Whole Program Mode

- Original source has 32 byte struct
- Program malloc's for large vector
- All hot segments touch one field
- Ends up with poor cache behavior
  - 32 byte stride, 25% utilization
- With whole program analysis
  - Compiler splits the vector
  - Generates four vectors
  - Hot segments get 8 byte stride
  - 100% cache block utilization
- Performance is improved

```c
setup(p);
for (i=0; i<STEPS; i++) {
    transform_x(p);
    transform_y(p);
    transform_z(p);
    transform_t(p);
}
report(p);
```
Automatic Parallelization
Automatic Parallelization: Key Points

- Compiler does the parallelization *automatically*
  - Just use the -xautopar option
  - No other user action required

- Automatic parallelization targets loop nests
  - Works synergistically with loop transformations
  - Steadily improving - handles many complex cases now

- Thread count controlled by environment variable

- Two versions generated (if profitability cannot be statically determined)
  - Run time selection between serial and parallel versions
  - Serial version used if work/thread is too low (high overhead)
Example 1 – Matrix Addition

for (i=0; i<m; i++)
    for (j=0; j<n; j++)
        a[i][j] = b[i][j] + c[i][j];

$ cc -xO4 -xautopar -xloopinfo main.o loop.c
"loop.c", line 7: PARALLELIZED, and serial version generated
"loop.c", line 8: not parallelized, not profitable
$ time a.out
    // Default is 1 thread
real    0m37.96s
user    0m36.64s
sys     0m0.97s
$ PARALLEL=2
$ time a.out
    // For a two-way parallel run
real    0m21.99s
user    0m41.07s
sys     0m2.23s

The same binary can run with different numbers of threads.

Turns on auto-parallelization
Prints parallelization information
Example 2 – Multiple Transforms

```c
... first(m,n);
second(m,n);
...

for (j=0; j<n; j++)
  for (i=0; i<m; i++)
    a[i][j] = b[i][j] + c[i][j];

for (i=1; i<m-1; i++)
  for (j=1; j<n-1; j++)
    b[i][j] = 0.5*c[i][j];
```

- Top level routine has two calls
- Loop in first()
- Loop in second()
- Routines inlined
- Loop nest in first() interchanged
- Loop nest in first() peeled
- Fused with loop nest in second()
- Loops parallelized at outer level
- Inner loop pipelined
- Prefetches inserted
- Difficult to understand final code
- Use commentary, other options
OpenMP

www.openmp.org
OpenMP for Parallelization

- It is an industry standard (www.openmp.org)
  - Supported by a large number of compilers
  - OpenMP code is portable
    - Directives can be ignored for serial or unsupported systems
- Requires little programming effort
  - Can start with just a handful of directives
  - Applications can be parallelized incrementally
- Good performance and scalability possible
  - Depends ultimately on the code, compiler, and system
    - Many excellent proof points on Sun's systems
    - CMT-friendly shared-memory parallelism leveraged
The OpenMP Execution Model

Fork-Join Model

Master Thread

Worker Threads

Worker Threads

\{ Serial Region \}

\{ Parallel Region \}

\{ Parallel Region \}
The OpenMP Memory Model

Programmer must decide attributes of variables: shared, private, firstprivate, lastprivate, ...
Example 1 - Loop

for (i=0; i<n; i++)
    a[index[i]] = a[i] + 2;

$ cc -xO4 -xautopar -xloopinfo loop.c
"loop.c", line 8: not parallelized, unsafe dependence (a)

#pragma omp parallel for shared(n,a) private(i)
for (i=0; i<n; i++)
    a[index[i]] = a[i] + 2;

$ cc -xO4 -xopenmp -xloopinfo loop.c
"loop.c", line 9: PARALLELIZED, user pragma used
$ OMP_NUM_THREADS=4    // Controls number of threads
$ a.out
Example 2 - Sections

```c
#pragma omp sections
{
 #pragma omp section
   foo1();
 #pragma omp section
   foo2();
 #pragma omp section
   foo3();
 #pragma omp section
   foo4();
}
```

- It is not just for loops
- Arbitrary pieces easily parallelized
- Must be legal, of course
- In this example:
  - foo1() - foo4() run in parallel
Sun Value-Added Features

- Version 2.5 fully supported
  - Includes support for nested parallelism
- Performance tuned for Solaris and Sun systems
- Idle thread behavior can be controlled
- Static and runtime error checking
- OpenMP debugging using dbx
- OpenMP performance profiling
- Autoscooping
  - The compiler can assist the user with scoping the variables
Original Code – CFD Application

...  
!$omp & rusbpz, rusbmz, rustz, rusqz, rushz, ruzfz, rusbz
!$omp & rwsbpz, rwsbmz, rwstz, rwsqz, rwshz, rwzfz, rwsbz
!$omp & rxsbpz, rxsbmz, rxstz, rxsqz, rxshz, ryzfz, rysbz
!$omp & rysbpz, rysbmz, rystz, rysqz, ryshz, ryzfz, rysbz
!$omp & twsbpz, twsbnz, twstz, twsqz, twshz, twzfz, twsbz
!$omp & rusbpm, rusbm1, ruqtz, rpsqz, rsshz, ppqz, rusbz
!$omp & rwsbpm, rwsbm2, rwshtz, rpsqz, rwszh, ppqz, rwsbzd
!$omp & rxsbpn, rxsbm3, rxstz, rxsqz, rxshz, ryzfz, rysbzd
!$omp & rysbpq, rysbm4, rystz, rysqz, ryshz, ryzfz, rysbzd
!$omp & twsbpo, twsbn5, twstz, twsqz, twshz, twzfz, twsbzd
!$omp & rysbps, rysbm6, rystz, rysqz, koshz, irzfz, rrmwzd
!$omp & twsbpt, twsbn7, twstz, twsqz, twshz, lqzfz, twqqzd
...

!$omp do
    do i = is, ie
        [...] 1949 lines omitted
    end do
!$omp end do
!$omp end parallel
Code with Autoscooping

```c
!$omp parallel DEFAULT(__AUTO)
!$omp do
  do i = is, ie
    [...] 1949 lines omitted
  end do
!$omp end do
!$omp end parallel
```
C Autoscooping Example

```bash
$ cc -fast -g -c -xopenmp -xloopinfo -xvpara loop.c
$ er_src -cc parallel -src loop.c loop.o
...
...
Source OpenMP region below has tag R1
Variables autoscooped as SHARED in R1: b, c, n, a
Private variables in R1: i
Shared variables in R1: a, b, c, n
  8. #pragma omp parallel for default(__auto)

L1 parallelized by explicit user directive
  9.   for (i=0; i<n; i++)
 10.     a[i] = a[i] + 2*b[i] + c[i];
 11. }

• Compiler commentary lists the autoscooping done
```
Libraries
The Basic Math Library

- Includes functions such as exp(), log(), sin(),...
- Part of Solaris, but developed by the compiler team
- Adhere to IEEE 754
- If gradual underflow is not important
  - Use -fns (implied by -fast)
- All C99 functions available in libm
- Vector versions of common functions included in libmvec
- Can be recognized automatically by the compiler
Example 1 – Vector exp()

```
for (i=0; i<n; i++)
a[i] = exp(b[i]);
```

- Normal compile calls exp
- With -xvector, calls __vexp
- Vexp() runs in parallel
- If idle processors available

Loops may be split to enable calling vector functions.
The Sun Performance Library

- Collection of linear algebra routines
  - LAPACK 3.0, BLAS 1-3
  - Standard routines from www.netlib.org
- Enhanced collection in Sun Performance Library
- Highly optimized for Sun systems
- Parallelized versions of the most important functions
- Run time selection of best version for execution platform
- Provides portable high-performance
Example 2 – Matrix Multiply

```fortran
do jj = 1,n,nb
    call zip(b(1,jj),%val(nb))
    do ii = 1,n,na
        call zip(a(1,ii),%val(na))
        do jjj = jj,jj+nb-1,3
            do iii = ii,ii+na-1,3
                call getts()
                call foo(a,b,c,iii,jjj)
                call gette()
            end do
        end do
    end do
end do
call gettp()
end do
```

```bash
$ f90 -fast main.f \ hrtime.o -lsunperf
$ a.out
Mflops:  2640
$ PARALLEL=8
$ a.out
Mflops:  19736
```

- No coding sweat
- No debugging pains
- No tuning headaches
- Great performance!
- Portable!
- And parallel!

+ There's more code behind!

```fortran
call dgemm('T','N',n,n,n,1.d0,a,num,b,num,0.d0,d,num)
```
mediaLib

- Part of Solaris, but developed by the compiler team
- Makes it easy to use available SIMD instructions
  - Collection of library routines provided for:
    - Imaging
    - Signal and Audio
    - Video processing
    - Graphics
    - ... and more
- Portability across SPARC and x64
  - Plain “C” versions exist for additional portability
- Performance tuned for Sun systems
- Parallelized versions available for many functions
Example 3 – FIR Filter

```c
for (n = 0; n < dlen; n ++) {
    tmp = 0;
    for (k = 0; k < flen; k ++)
        tmp += fir[k] * src[n+k];
    dst[n] = (vis_s16) (tmp >> 16);
}

vis_write_gsr(0);
da = (vis_u8 *) dst;
dp = (vis_d64 *) ((vis_u32) da & (~7));
off = (vis_u32) dp - (vis_u32) da;
dend = da + 2 * dlen - 1;
emask = vis_edge16(da, dend);
sa = (vis_u8 *) src;
num = ((vis_u32)dend>>3) - ((vis_u32)da>>3) + 1;
for (n = 0; n < num; n ++) {
    ss = sa;
    rdh = vis_fzero(); rdl = vis_fzero();
    for (k = 0; k < flen; k ++) {
        sp = (vis_d64 *) vis_alignaddr(ss, off);
        s0 = sp[0]; s1 = sp[1];
        sd = vis_faligndata(s0, s1);
        ...
```

- No need to write VIS
- No need to write MMX
- Just use mediaLib functions
- Perf. gain of ~6X (avg.)
- C version exists
- Can be run on any system

+ There's more code!
The Sun Performance Analyzer
Features Overview

- Analyzer – an advanced performance analysis tool
- Intuitive GUI interface
- Clock based statistical profiling
- HW counter based statistical profiling
- Can relate data to function, source, assembly level
- Integrated with compiler commentary
- Dataspace and memory space profiling
- Enhanced OpenMP support
Example 1 – Where did the time go?

<table>
<thead>
<tr>
<th>Excl.</th>
<th>Incl.</th>
<th>Name</th>
<th>User CPU sec</th>
<th>User CPU sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>36.956</td>
<td>36.956</td>
<td>&lt;Total&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36.956</td>
<td>36.956</td>
<td>loop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.</td>
<td>0.</td>
<td>__collector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.</td>
<td>0.</td>
<td>__open</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.</td>
<td>0.</td>
<td>__exithandle</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...compiler commentary here...

8.036 8.036 8. for (i=0; i<n; i++)
## 28.920 28.920 9. a[i] = b[i]/alpha;
...

0.690 0.690 [ 9] 10ce0: ldd ...
## 5.174 5.174 [ 9] 10ce4: fdivd ...
3.522 3.522 [ 9] 10ce8: std ...
0.941 0.941 [ 9] 10cec: ldd ...
## 5.064 5.064 [ 9] 10cf0: fdivd ...

Function view

Source view, with commentary

Assembly view, with line numbers
Example 1 – Tuning

```
$ cc -xO4 -g main.c loop.c
main.c:
loop.c:
$ time a.out
real  0m36.81s
user  0m36.67s
sys   0m0.00s

$ collect -p on a.out
$ analyzer

$ cc -xO4 -fsimple=2 -g main.c loop.c
main.c:
loop.c:
$ time a.out
real  0m10.09s
user  0m10.00s
sys   0m0.00s
```

First run, simple compile

Analyze - “Aha, it's the divide!”

Add -fsimple=2, changes div->multiply, nice speedup
Example 2 – The memory bottleneck

<table>
<thead>
<tr>
<th></th>
<th>Excl. Iinstr_cnt</th>
<th>Incl. Iinstr_cnt</th>
<th>Excl. IEvents</th>
<th>Incl. IEvents</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>L3_miss</td>
<td>2452671487</td>
<td>2452671487</td>
<td>96803274</td>
<td>96803274</td>
<td>&lt;Total&gt;</td>
</tr>
<tr>
<td>loop</td>
<td>2327052822</td>
<td>2327052822</td>
<td>93803181</td>
<td>93803181</td>
<td>loop</td>
</tr>
<tr>
<td>main</td>
<td>125618665</td>
<td>2452671487</td>
<td>3000093</td>
<td>96803274</td>
<td>main</td>
</tr>
<tr>
<td>_start</td>
<td>0</td>
<td>2452671487</td>
<td>0</td>
<td>96803274</td>
<td>_start</td>
</tr>
</tbody>
</table>

...compiler commentary here...

0 0 0 0

11. for (i=0; i<n; i++)

## 2327052822 2327052822 93803181 93803181

12. t += a[index[i]];

9200436 9200436 0 0

[13] 10908: ldd [%o0 + %10], %f4

0 0 0 0

[13] 1090c: fadd %f2, %f10, %f14

## 1154977323 1154977323 93803181 93803181

[13] 10910: sll %g5, 3, %g4
Example 2 – Tuning

$ cc -fast -g main.c loop.c
main.c:
loop.c:
$ time a.out
real  0m17.49s
user  0m16.24s
sys   0m0.97s

$ collect -h Instr_cnt,h,L3_miss,h a.out
$ analyzer

First run, simple compile

Analyze - “Aha, it's the indirect ldd's L3 miss”

Add -xprefetch_level=3, prefetch emitted, nice speedup

$ cc -fast -g -xprefetch_level=3 main.c loop.c
main.c:
loop.c:
$ time a.out
real  0m6.79s
user  0m5.54s
sys   0m0.97s
Example 3 – Database on US T1

- **Symptom:** Slower with 32 threads than 8

- **Analysis:**
  - Most traffic to one memory bank
  - All 32 processes: same VA, different PA, same cache line

- **Solution:**
  - Shatter the hot page into smaller ones
  - Changes mappings to go to different lines

- **Result:**
  - ~6X improvement on US T1 with 32 processes
  - Took 3 hours to get to the “Aha” point with Analyzer
Technology Previews

Some of the items in this section are available from the Sun Download Center now. Please stay tuned for updates regarding the others.
1. Embracing SPARC/gcc Users

- Many developers use gcc
  - Want to use the same compiler for different platforms
  - Use gcc language extensions
  - Familiar with & feel comfortable with gcc
  - Migration to Sun Studio is, or is viewed as being, difficult

Would be nice to bring the features of Sun Studio to these users.
How do we get there?

GCC for SPARC Systems

gcc front-end  Sun IR generator

IPO
Parallelizer
Optimizer
Code Generator
Link time optimizer
Binary optimizer

TOOLS

Studio front-end
Goals of the Project

- Transparent to gcc users
  - Feature compatible with gcc
  - Debuggable with gdb and dbx
- Improved performance
  - Through advanced optimizations tuned to SPARC systems
  - Extra optimizations such as -xipo, -xprefetch, -xprofile
- Higher reliability
2. Enhancing Parallel Debugging

What is a race condition?

```c
for (i=0; i<n; i++)
    a[i] = a[i+1] + b[i];
```

Sequential execution: Results are deterministic
Parallel execution: Results non-deterministic

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
</table>

Example:
Thread 1 executes i=0-4
Thread 2 executes i=5-9
Final value of a[4] wrong!

This is a data race.
Race Detection Tool

- On-the-fly data race detection

- Two phases
  - Instrumentation phase
    - Instruments the executable (if not already instrumented)
  - Execution phase
    - Runs the executable, monitors memory accesses and thread synchronizations, and logs any data race condition detected

- Analysis phase
  - Displays the results gathered
Goals of the Race Detection Project

- Handle a variety of multi-threaded applications
  - Open MP
  - Posix threads
  - Solaris threads
  - Sun/Cray directives
  - Mixtures of the above

- Instrumentation at the IR or binary level

- Considering an open API
  - To support user annotations
3. Automating Tuning & Troubleshooting

- Why automate these tasks?
  - Performance tuning issues
    - Time consuming
    - Requires experienced staff
    - Functionality takes all the time, leaving little room for tuning
  - Debugging issues
    - Time consuming, often more than writing the “first cut” code
    - Requires expensive developer time
    - Main cause of schedule slips
- With CMT, threads are cheap
  - Use computers to ease above tasks
  - ATS (Automatic Tuning System)
The Backbone of ATS - PEC

- Store the intermediate representation (IR) with the executable (object): Portable Executable Code (PEC)
  - The IR can be extracted and reprocessed
Previous Tuning Example with ATS

$ cc -fast -g <pecoption> main.c loop.c
main.c:
loop.c:
$ time a.out

real 0m17.49s
user 0m16.24s
sys 0m0.97s

$ collect -h Instr_cnt,h,L3_miss,h a.out
$ analyzer

$ cc -fast -g -xprefetch_level=3 main.c loop.c
main.c:
loop.c:
$ ats -i '-fast -g -xprefetch_level=3' \
> -keepbin a.out
$ time a.out.1

real 0m6.83s
user 0m5.59s
sys 0m0.96s

First run, simple compile (with pec)

Analyze - “aha, it's the indirect ldd's L3 miss”

What if the source is unavailable? With ATS & PEC, we can still tune the a.out!
Benefits of ATS+PEC

- Can operate on an a.out; source not required
- Faster recompilations because it starts with the IR
- Automatic tuning by searching for the best options
- Profile feedback can be applied by the user
- Can search for and find the offending module in the event of a failure; build a binary with a workaround
- Put those computers to good use instead of drawing circles on the screen!
4. Analyzing & Improving Binaries

- BIT - A tool that operates *reliably* on binaries
- Can instrument and collect information for analysis
- Can create a new binary with improved performance
  - Focusses on rearranging code to better use the I-cache
  - Works best on large, complex applications
- Build with
  - Sun Studio 11
  - Add `-xbinopt=prepare`
  - Use `-O1` or higher optimization level
Analysis with BIT

```
$ cc -fast -xbinopt=prepare *.c -lm  // Build for BIT
...
and.c:
buidl-disjuncts.c:
extract-links.c:
...
$ bit instrument a.out  // Instrument the a.out
$ a.out.instr 2.1.dict -batch < input  // Run a.out.instr
$ bit analyze -a ifreq a.out | more  // Analyze the run
```

Instruction frequencies for whole program:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Executed</th>
<th>(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL</td>
<td>18586774239</td>
<td>(100.0)</td>
</tr>
<tr>
<td>load store</td>
<td>4879117593</td>
<td>(26.3)</td>
</tr>
<tr>
<td>load</td>
<td>3794101701</td>
<td>(20.4)</td>
</tr>
<tr>
<td>store</td>
<td>1085015892</td>
<td>(5.8)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Executed</th>
<th>(%)</th>
<th>Annulled</th>
<th>In Delay Slot</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL</td>
<td>18586774239</td>
<td>(100.0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lduw</td>
<td>2889942469</td>
<td>(15.5)</td>
<td>83993915</td>
<td>629750312</td>
</tr>
<tr>
<td>br</td>
<td>2797374496</td>
<td>(15.1)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>subcc</td>
<td>2222270229</td>
<td>(12.0)</td>
<td>0</td>
<td>329895247</td>
</tr>
</tbody>
</table>
Examining Code Coverage

$ cc -fast -xbinopt=prepare *.c -lm  // Build for BIT
...
and.c:
bUILD-disjuncts.c:
extract-links.c:
...
$ bit instrument a.out  // Instrument the a.out
$ a.out.instr 2.1.dict -batch < input  // Run a.out.instr
$ bit coverage a.out  // Analyze coverage

Creating experiment database test.1.er ...
BIT Code Coverage
Total Functions: 350
Covered Functions: 216
Function Coverage: 61.7%
Total Basic Blocks: 6,041
Covered Basic Blocks: 3,969
Basic Block Coverage: 65.7%
Total Basic Block Executions: 3,955,536,194
Average Executions per Basic Block: 654,781.69
Total Instructions: 27,606
Covered Instructions: 17,680
Instruction Coverage: 64.0%
Total Instruction Executions: 18,866,478,764
Average Executions per Instruction: 683,419.50
5. Simplifying Performance Optimisation

- SPOT – A Simple Performance Optimisation Tool
  - Produces a report on a code's execution
  - Exposes common causes of performance loss
  - Very easy to use

- SPOT reports contain hyperlinked profiles
  - Makes it easy to navigate from performance issue to source to assembly

- For maximum information
  - Add -g (-g0 for C++)
  - Use -O1 or higher
  - Include -xbinopt=prepare
Using SPOT

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ cc -fast -xbinopt=prepare -g *.c -lm // Build the code</td>
<td></td>
</tr>
<tr>
<td>... and.c: build-disjuncts.c: extract-links.c: ...</td>
<td></td>
</tr>
<tr>
<td>$ spot -X a.out 2.1.dict -batch &lt; input &gt; /dev/null // Run SPOT</td>
<td></td>
</tr>
</tbody>
</table>

- SPOT runs the code many times
- Approx. 20X longer (in this mode)
- Invokes other tools to collect data
- Generates comprehensive report
## SPOT Report Example

### Application stall information (using ripc)

---

**NOTE:** Time reported under D$ miss also includes the time spent in L2 cache misses

<table>
<thead>
<tr>
<th>UltraSparc</th>
<th>ticks</th>
<th>sec</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispatch0_IC_miss</td>
<td>78503999</td>
<td>0.065</td>
<td>0.3%</td>
</tr>
<tr>
<td>Dispatch0_br_target</td>
<td>323039451</td>
<td>0.267</td>
<td>1.1%</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DTLB_miss</td>
<td>11859795</td>
<td>0.010</td>
<td>0.0%</td>
</tr>
<tr>
<td>Re_DC_miss</td>
<td>6402197760</td>
<td>5.296</td>
<td>22.1%</td>
</tr>
<tr>
<td>Re_EC_miss</td>
<td>305913838</td>
<td>0.253</td>
<td>1.1%</td>
</tr>
<tr>
<td>Re_PC_miss</td>
<td>99</td>
<td>0.000</td>
<td>0.0%</td>
</tr>
<tr>
<td>Re_RAW_miss</td>
<td>56567038</td>
<td>0.047</td>
<td>0.2%</td>
</tr>
<tr>
<td>Re_FPU_bypass</td>
<td>0</td>
<td>0.000</td>
<td>0.0%</td>
</tr>
<tr>
<td>Rstall_storeQ</td>
<td>678244916</td>
<td>0.561</td>
<td>2.3%</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Stalltime</td>
<td>11545566208</td>
<td>9.551</td>
<td>39.8%</td>
</tr>
</tbody>
</table>

---

| Total CPU Time     | 29010841600 | 24.000 | 100.0% |
| Total Elapsed Time | 24.000      | 24.000 | 100.0% |
| Instr               | 18990794752 |        |       |
| IPC                 | 0.655       |        |       |

---

See full report

- Provides
  - HW info.
  - Build details
  - Stalls/IPC
  - Profiles
  - Traps data
  - Coverage
  -...

---

Ride the CMT Wave with Sun's Compilers/Partha T.  
Multicore Expo, March 21-23. 2006

---

62
Summary

❑ Sun has industry leading compilers and tools for CMT

❑ Components are thread-aware and work synergistically

❑ Reliable, with advanced optimizations and parallelization

❑ Excellent multi-threaded analysis and debugging tools

  ❑ Compiler commentary, automatic parallelization, OpenMP, tuned libraries, performance analyzer, ...

❑ Exciting new tools and enhancements are under-way

❑ Embracing the SPARC/gcc community

❑ Race detection in multi-threaded programs

❑ Automation of the tuning and troubleshooting process

❑ …
More Information & Resources

- OpenSPARC information & community
  www.opensparc.net
- CoolTools site
  cooltools.sunsorce.net
- Sun Forums (including developer forum)
  forum.sun.com
- Contacting Developer Tools Marketing
  Developer911@sun.com
- Sun Studio Product site
  sun.com/sunstudio11
- Sun Studio Developer site
  developers.sun.com/sunstudio
Thank You!

Partha .Tirumalai@Sun.Com