Subroutine Profiling Results for the CPU2006 Benchmarks

Reinhold P. Weicker
Retired, SPEC CPU Subcommittee Member 1990 – 2006
Contact: reinhold.weicker@t-online.de

John L. Henning
Sun Microsystems
Contact: john dot henning at acm dot org

Subroutine Profiling

Subroutine profiling is a well-known performance tool. For application or system programmers, it determines "hot spots" where the program spends most of its time, and where careful rewriting can most help performance. For compiler authors, it can give information about programming style in such hot spots, and can indicate where compiler improvements may be useful. For hardware designers and analysts, it can be the starting point to explain performance behavior.

During selection and porting of the CPU2006 benchmarks, subroutine profiling was performed routinely for test versions of the suite. It influenced the selection of benchmarks. For example, SPEC uses benchmark profiles to help determine weak spots in benchmark program candidates: Does a program spend a large part of its time in subroutines that are in some way badly or unusually written, or too easily optimized by a narrowly focused method? In SPEC CPU subcommittee jargon, this was sometimes referred to as the danger of "low-hanging fruit". However, those making the selection were aware that some real-life programs do exhibit a high degree of locality. Therefore, a peaky profile does not alone disqualify a benchmark candidate.

During development of CPU2006, SPEC considered many types of profiles, including low and high optimization, 32 and 64-bit, Unix and Windows, and on various hardware.

Profile variability

It is understood that profiling results can vary. They are not independent from the hardware and the software on which the program is running. Even for the same instruction set and the same compiler, times for individual instructions may vary between implementations. For example, Load instructions may have a short or a long latency, or their latency may be hidden by clever prefetching into caches. Results for 64-bit executables may be different from those for 32-bit executables. Results may change depending on library functions. And, perhaps most importantly, optimizations performed by the compiler may greatly influence the code that is executed. If inlining is involved, some subroutines visible at low optimization levels may completely disappear at higher optimization, with their execution times subsumed into the calling subroutine's time.

Therefore, any reference set of profiles must be regarded merely as a beginning for analysis: from the profiles provided here, one can get a sense of the complexity of a program, or the extent to which a profile is concentrated in a single area. Reference profiles may come in handy when deciding on an initial focus area: "if I want to learn about the performance of perl, I might as well start with `S_regmatch`."

Methods

As mentioned above, at high optimization routines that are visible in the source code may become invisible at run time.. Therefore, the profiles presented here use only moderate optimization (**`-O`**). All binaries are 32-bit.

In cases where a benchmark has several input sets (e.g. 400.perlbench), profiling results are given for each individual invocation of the benchmark binary; in the same order here as when they are run. It is not surprising (and even, to some degree, intended) that different input files often cause the program to exercise different paths, and that the subroutine distribution may vary considerably between invocations. Information about the benchmark input sets may be found at [2].

Each table lists only the 20 highest-scoring subroutines. Routines using less than 1% are not reported. A leading underscore (_) indicates a library subroutine. C++ benchmarks have very long routine names; these have been ruthlessly truncated in the interest of space. Note that the routines **`_mcount`** and **`mcount_single`** are not part of the benchmarks; they are due to profile collection overhead.

All benchmarks were compiled using the same major version of the compiler and the same optimization level. But it should be noted that profile collection used two methods.

(1) For most of the benchmarks, the sources were compiled using a recent compiler (Sun Studio 11) with the switch **`-p`**, which causes the insertion of data-collection code into the executable. The benchmarks were then run on a SPARC-64 system under Solaris 5.8. The standard Unix **`prof`** utility was then used to display the collected data.

(2) For some benchmarks, the **`-p`** method was not preferred, either because the overhead from profiling code (**`mcount`**) exceeded 20%, or because of software compatibility issues. Instead, benchmarks were compiled using a later patch level of the same compiler as in method 1 (without **`-p`**), and were then run on an UltraSPARC system under Solaris 5.10 using the **`collect`** command [1]. The **`er_print`** utility was used to display the collected data. With this method, there was still some profiling overhead. It is seen below as **`take_deferred_signal`**, which is associated with time in lock-protected code, typically memory allocation routines. But in all cases overhead was less than with method 1. The benchmarks that used method 2 were: 400.perlbench, 403.gcc, 436.cactusADM, 445.gobmk, 447.dealII, 453.povray, 458.sjeng, 464.h264ref, 471.omnetpp, 473.astar, and 483.xalancbmk.

For some benchmarks, informal observations / notes / opinions are added at the end of the profiling result tables. Of course, they reflect personal opinions and should not be taken as objective results or as SPEC-endorsed opinions.

Results for the Integer Benchmarks CINT2006

400.perlbench (C program)

Invocation 1 (about 49% of overall execution time)

36.5	S_regmatch
8.9	S_find_byclass
6.8	S_regtry
6.0	Perl_leave_scope
4.6	Perl_runops_standard
2.5	take_deferred_signal
1.9	Perl_save_alloc
1.7	S_hv_fetch_common
1.6	Perl_sv_setsv_flags
1.4	Perl_pp_entersub
1.1	Perl_pp_match

400.perlbench, invocation 2 (about 24% of time)

18.6	take_deferred_signal
12.0	S_regmatch
4.3	Perl_sv_setsv_flags
3.6	Perl_sv_free
3.5	lmutex_lock
3.2	Perl_regexec_flags
3.1	Perl_sv_clear
2.8	Perl_leave_scope
2.8	Perl_sv_upgrade
2.6	Perl_sv_setpvn
2.3	Perl_sv_grow
2.2	Perl_runops_standard
1.9	S_regtry
1.7	Perl_newSVsv
1.6	malloc
1.5	free
1.5	Perl_newSVpvn
1.5	memcpy
1.4	S_regrepeat

400.perlbench, invocation 3 (about 26% of time)

56.6	S_regmatch
6.1	take_deferred_signal
4.7	Perl_leave_scope
3.7	S_regtry
3.7	S_reginclass
2.3	S_find_byclass
2.3	Perl_runops_standard
1.7	memcpy
1.6	Perl_save_alloc

401.bzip2 (C program)

Invocation 1 (about 19% of overall execution time)

16.3	mainSort
14.4	BZ2_decompress
13.1	mainGtU
11.5	mainQSort3
8.3	copy_input_until_stop
6.0	sendMTFValues
5.9	generateMTFValues
5.6	_mcount
4.6	mainSimpleSort
4.6	unRLE_obuf_to_output_FAST
3.3	mcount_single
1.3	bsW
1.3	_memcpy
1.0	copy_output_until_stop

401.bzip2, invocation 2 (about 8% of time)

28.6	fallbackSort
13.6	BZ2_decompress
12.5	generateMTFValues
9.9	mainSort
8.2	mainGtU
6.3	fallbackQSort3
4.1	sendMTFValues
3.3	_mcount
2.4	copy_input_until_stop
2.4	unRLE_obuf_to_output_FAST
2.0	fallbackSimpleSort
1.9	mcount_single
1.4	copy_output_until_stop
1.1	mainSimpleSort
1.1	bsW

401.bzip2, invocation 3 (about 13% of time)

41.2	fallbackSort
12.7	fallbackSimpleSort
11.8	mainGtU
8.8	fallbackQSort3
5.7	BZ2_decompress
4.8	_mcount
3.5	generateMTFValues
3.4	mainSort
2.7	mcount_single
1.7	sendMTFValues
1.2	copy_input_until_stop
1.1	unRLE_obuf_to_output_FAST

401.bzip2, invocation 4 (about 20% of time)

21.9	BZ2_decompress
15.3	mainSort
12.1	generateMTFValues
8.3	mainGtU
7.8	copy_input_until_stop
7.8	sendMTFValues
6.2	mainQSort3
4.1	_mcount
3.2	unRLE_obuf_to_output_FAST
3.2	mainSimpleSort
2.5	mcount_single
1.8	bsW
1.8	copy_output_until_stop
1.3	_memcpy

401.bzip2, invocation 5 (about 23% of time)

34.1	mainGtU
14.3	mainSort
10.3	mainQSort3
7.8	BZ2_decompress
7.0	copy_input_until_stop
5.1	unRLE_obuf_to_output_FAST
4.3	_mcount
4.0	generateMTFValues
4.0	mainSimpleSort
2.9	mcount_single
2.5	sendMTFValues
1.0	_memcpy

401.bzip2, invocation 6 (about 15% of time)

14.8	mainSort
14.1	BZ2_decompress
12.2	mainGtU
9.6	mainQSort3
9.3	copy_input_until_stop
6.4	generateMTFValues
5.6	sendMTFValues
5.4	_mcount
4.4	fallbackSort
4.1	unRLE_obuf_to_output_FAST
4.0	mainSimpleSort
3.1	mcount_single
1.2	bsW
1.1	_memcpy
1.1	copy_output_until_stop
1.0	fallbackQSort3

403.gcc (C program)

Invocation 1 (about 8% of overall execution time)

21.8	reg_is_remote_constant_p
11.4	memset
5.1	clear_table
3.1	splay_tree_splay_helper
3.1	compute_transp
3.0	bitmap_operation
2.6	sbitmap_union_of_diff
2.3	bitmap_element_allocate
2.2	htab_traverse
2.1	single_set_2
1.5	delete_null_pointer_checks_1
1.3	init_alias_analysis
1.2	canon_rtx

403.gcc, invocation 2 (about 11% of time)

9.1	memset
4.8	ggc_mark_rtx_children_1
3.9	ggc_set_mark
2.8	bitmap_operation
2.5	htab_traverse
2.2	bitmap_element_allocate
2.1	ggc_mark_rtx_children
2.0	reg_is_remote_constant_p
1.9	init_alias_analysis
1.6	cse_insn
1.5	for_each_rtx
1.4	clear_table
1.4	mark_set_1
1.3	ggc_alloc
1.3	note_stores
1.2	constrain_operands
1.0	ggc_mark_trees
1.0	reg_scan_mark_refs

403.gcc, invocation 3 (about 11% of time)

24.8	memset
12.2	clear_table
3.5	compute_transp
3.3	bitmap_operation
2.7	sbitmap_union_of_diff
2.3	delete_null_pointer_checks_1
2.3	bitmap_element_allocate
1.8	htab_traverse
1.7	compute_dominance_frontiers_1
1.1	canon_rtx
1.1	loop_regs_scan
1.0	reg_is_remote_constant_p
1.0	ggc_set_mark

403.gcc, invocation 4 (about 8% of time)

17.4	memset
13.6	clear_table
5.9	compute_transp
3.8	canon_rtx
2.8	htab_traverse
2.6	bitmap_operation
2.2	delete_null_pointer_checks_1
2.2	sbitmap_union_of_diff
2.1	find_base_term
2.1	bitmap_element_allocate
1.6	compute_dominance_frontiers_1
1.5	rtx_equal_for_memref_p
1.1	ggc_mark_rtx_children_1
1.0	init_alias_analysis
1.0	memrefs_conflict_p
1.0	ix86_find_base_term

403.gcc, invocation 5 (about 9% of time)

18.1	memset
11.8	clear_table
8.7	compute_transp
5.2	htab_traverse
3.2	delete_null_pointer_checks_1
3.1	bitmap_operation
2.3	sbitmap_union_of_diff
2.2	bitmap_element_allocate
1.9	compute_dominance_frontiers_1
1.7	canon_rtx
1.0	init_alias_analysis
1.0	side_effects_p

403.gcc, invocation 6 (about 13% of time)

18.7	memset
11.0	clear_table
8.0	compute_transp
5.0	htab_traverse
3.5	delete_null_pointer_checks_1
3.2	bitmap_operation
2.4	sbitmap_union_of_diff
2.2	bitmap_element_allocate
1.8	compute_dominance_frontiers_1
1.2	canon_rtx
1.2	reg_is_remote_constant_p
1.1	init_alias_analysis

403.gcc, invocation 7 (about 20% of time)

27.1	reg_is_remote_constant_p
11.1	memset
7.4	htab_traverse
5.4	clear_table
4.3	fixup_var_refs_1
4.0	delete_null_pointer_checks_1
3.3	fixup_var_refs_insns
3.0	single_set_2
2.8	bitmap_operation
2.0	fixup_var_refs_insn
2.0	bitmap_element_allocate
1.9	sbitmap_union_of_diff
1.7	compute_dominance_frontiers_1
1.6	try_combine

403.gcc, invocation 8 (about 16% of time)

19.6	memset
11.0	reg_is_remote_constant_p
10.7	clear_table
5.1	compute_transp
4.9	bitmap_operation
4.3	delete_null_pointer_checks_1
3.0	bitmap_element_allocate
2.6	sbitmap_union_of_diff
1.5	canon_rtx
1.4	htab_traverse
1.3	single_set_2
1.1	splay_tree_splay_helper
1.1	find_base_term

403.gcc, invocation 9 (about 4% of time)

8.9	memset
4.2	ggc_set_mark
4.1	ggc_mark_rtx_children_1
2.1	constrain_operands
2.1	init_alias_analysis
2.0	for_each_rtx
2.0	cse_insn
1.9	ggc_mark_rtx_children
1.8	htab_traverse
1.4	take_deferred_signal
1.4	ggc_alloc
1.3	find_reloads
1.3	note_stores
1.2	bitmap_operation
1.2	ggc_mark_trees
1.0	mark_set_1
1.0	reg_scan_mark_refs
1.0	propagate_one_insn
1.0	record_reg_classea

Various versions of GCC have been in all SPEC CPU suites so far, overall flat profile. The high percentage for memset is a concern but as a compiler creates and destroys its various data structures, it seems understandable.

429.mcf (C program)

42.0	primal_bea_mpp
23.4	refresh_potential
12.2	_mcount
8.5	replace_weaker_arc
3.6	price_out_impl
2.9	mcount_single
1.9	update_tree
1.7	bea_is_dual_infeasible
1.7	primal_iminus
1.2	sort_basket

This benchmark is sensitive to memory latency. For 64 bit, the two top subroutines reverse their positions.

445.gobmk (C program)

Invocation 1 (about 13% of overall execution time)

5.7	undo_trymove
4.6	fastlib
4.3	order_moves
3.8	scan_for_patterns
3.5	incremental_order_moves
3.3	do_play_move
3.1	do_trymove
3.0	compute_connection_distances
2.5	remove_liberty
2.4	assimilate_string
2.1	remove_neighbor
2.1	get_next_move_from_list
2.1	hashtable_clear
2.0	extend_neighbor_string
1.9	approxlib
1.9	do_push_owl
1.9	is_self_atari
1.8	hashtable_search
1.6	propose_edge_moves

445.gobmk invocation 2 (about 34% of time)

5.8	undo_trymove
4.9	fastlib
4.6	order_moves
3.6	incremental_order_moves
3.4	do_trymove
3.4	do_play_move
3.3	compute_connection_distances
3.1	scan_for_patterns
2.9	remove_liberty
2.9	do_matchpat
2.5	assimilate_string
2.5	accumulate_influence
2.4	extend_neighbor_string
2.2	remove_neighbor
2.1	approxlib
1.9	is_self_atari
1.8	hashtable_search
1.7	chainlinks2
1.7	count_common_libs

445.gobmk, invocation 3 (about 21% of time)

24.5	do_matchpat
13.0	hashtable_clear
3.5	compute_connection_distances
3.4	undo_trymove
3.3	accumulate_influence
2.7	scan_for_patterns
2.2	order_moves
2.2	fastlib
1.8	incremental_order_moves
1.8	do_play_move
1.7	do_trymove
1.5	assimilate_string
1.5	remove_liberty
1.3	remove_neighbor
1.3	update_liberties
1.2	extend_neighbor_string
1.0	approxlib
1.0	is_self_atari

445.gobmk, invocation 4 (about 13% of time)

```
6.0   undo_trymove
4.8   fastlib
4.3   order_moves
3.9   compute_connection_distances
3.8   incremental_order_moves
3.6   hashtable_clear
3.5   do_trymove
3.5   do_play_move
2.9   remove_liberty
2.6   do_matchpat
2.6   assimilate_string
2.3   scan_for_patterns
2.2   remove_neighbor
2.2   extend_neighbor_string
2.1   approxlib
2.0   is_self_atari
1.8   hashtable_search
1.6   count_common_libs
1.5   chainlinks2
```

445.gobmk, invocation 5 (about 18% of time)

```
6.7   undo_trymove
6.1   compute_connection_distances
4.6   fastlib
4.4   order_moves
4.2   do_matchpat
3.8   do_play_move
3.7   incremental_order_moves
3.6   do_trymove
3.4   remove_liberty
2.8   assimilate_string
2.8   extend_neighbor_string
2.7   accumulate_influence
2.3   remove_neighbor
1.9   approxlib
1.9   scan_for_patterns
1.8   is_self_atari
1.7   is_suicide
1.7   hashtable_clear
1.7   create_new_string
```

456.hmmer (C program)

Invocation 1 (about 30% of overall execution time)

```
97.0   P7Viterbi
```

456.hmmer, invocation 2 (about 70% of time)

```
93.1   P7Viterbi
 2.1   FChoose
 1.4   sre_random
 1.3   _mcount
```

The peaky profile (subroutine P7Viterbi) might be considered a concern with this program. But it is a large subroutine, and the author of hmmer is well aware of its importance for performance. Several alternative versions are provided in the source files for this subroutine. SPEC has chosen one that is intended to preserve the level playing field, by not providing an unfair advantage to any particular implementation.

458.sjeng (C program)

```
18.7   std_eval
 9.4   setup_attackers
 8.4   gen
 4.9   remove_one
 4.8   order_moves
 4.2   is_attacked
 4.1   QProbeTT
 4.1   search
 3.3   make
 3.2   push_slidE
 3.1   unmake
 2.9   ProbeTT
 2.8   Pawn
 2.8   checkECache
 2.2   rook_mobility
 2.0   add_move
 1.6   bishop_mobility
 1.5   check_legal
 1.5   see
```

462.libquantum (C program)

```
56.2   quantum_toffoli
27.6   quantum_sigma_x
12.9   quantum_cnot
 1.4   quantum_gate1
```

The peaky profile (top subroutine: Only 28 lines of source code) and the fact that the program has a very high cache miss ratio / exhibits large memory pressure may create an incentive for optimizations to reduce memory pressure. In particular since the program is from a research environment, one would need to verify that such optimizations benefit other programs as well.

464.h264ref (C program)

Invocation 1 (about 9% of overall execution time)

```
30.9   SetupFastFullPelSearch
14.9   SubPelBlockMotionSearch
13.2   FastFullPelBlockMotionSearch
 5.2   SetupLargerBlocks
 4.2   UMVLine16Y_11
 4.1   SATD
 3.5   dct_luma
 3.4   FastPelY_14
 3.3   FastLine16Y_11
 1.3   memcpy
 1.0   get_mb_block_pos
 1.0   Mode_Decision_for_4x4IntraBlocks
```

464.h264ref, Invocation 2 (about 9% of tme)

```
35.6   memcpy
15.6   SetupFastFullPelSearch
 6.9   SubPelBlockMotionSearch
 6.4   FastFullPelBlockMotionSearch
 5.4   dct_luma
 2.7   SetupLargerBlocks
 2.1   UMVLine16Y_11
 1.9   SATD
 1.9   biari_encode_symbol
 1.6   FastPelY_14
 1.6   FastLine16Y_11
 1.4   Mode_Decision_for_4x4IntraBlocks
 1.2   get_mb_block_pos
 1.1   OneComponentChromaPrediction4x4
```

464.h264ref, Invocation 3 (about 82% of time)

```
 40.9   memcpy
 15.0   SetupFastFullPelSearch
  9.1   FastFullPelBlockMotionSearch
  6.4   SubPelBlockMotionSearch
  4.6   dct_luma
  2.5   SetupLargerBlocks
  1.8   SATD
  1.7   FastLine16Y_11
  1.6   FastPelY_14
  1.3   Mode_Decision_for_4x4IntraBlocks
  1.1   OneComponentChromaPrediction4x4
```

Remark: There may be some concern about the high percentage for _memcpy. Perhaps it is unavoidable for this application (video compression).

471.omnetpp (C++ program)

```
 14.2   cMessageHeap::shiftup
 10.0   take_deferred_signal
  5.8   cSubModIterator::operator++
  5.1   cGate::deliver
  4.2   cObject::setOwner
  3.5   EtherMAC::handleMessage
  3.2   cModule::findGateconst
  2.8   cOutVector::record
  2.5   cSimulation::selectNextModule
  2.3   cObject::~cObject
  2.2   cFileOutputVectorManager::record
  2.0   cSimpleModule::scheduleAt
  1.8   cMessageHeap::insert
  1.6   cMessage::operator=
  1.6   strcmp
  1.6   cMessage::cMessage
  1.5   cSimpleChannel::deliver
  1.5   cArray::get
  1.5   std::_Rb_global::_M_increment
```

473.astar (C++ program)

Invocation 1 (about 48% of overall execution time)

```
 24.2   regmngobj::getregfillnum
 17.6   regwayobj::makebound2
 14.2   regwayobj::isaddtobound
 14.2   way2obj::releasepoint
 12.6   wayobj::makebound2
  6.0   way2obj::addtobound
  3.8   way2obj::isaddtobound
  3.4   way2obj::releasebound
```

473.astar, invocation 2 (about 52% of time)

```
 28.5   wayobj::makebound2
 22.1   way2obj::releasepoint
 11.7   way2obj::addtobound
 11.2   regmngobj::getregfillnum
  8.9   regwayobj::makebound2
  6.1   regwayobj::isaddtobound
  5.5   way2obj::isaddtobound
  4.2   way2obj::releasebound
```

Moderately peaky profile, relatively small program size: the top 3 routines in invocation 1 total less than 50 lines of code.

483.xalancbmk (C++ program)

```
 10.5   std::__find
  9.8   xercesc_2_5::ValueStore::contains
  9.2   xercesc_2_5::XMLString::stringLen
  5.0   xercesc_2_5::BaseRefVectorOf::elementAt
  4.9   take_deferred_signal
  3.8   xalanc_1_8::ReusableArenaAllocator::destroyObject
  3.4   xercesc_2_5::ValueVectorOf::elementAt
  3.3   memcpy
  2.6   xercesc_2_5::XMLString::equals
  1.7   xalanc_1_8::ReusableArenaAllocator::allocateBlock
  1.7   xercesc_2_5::ValueStore::isDuplicateOf
  1.7   xalanc_1_8::ReusableArenaBlock::ownsObject
  1.6   xalanc_1_8::XalanReferenceCountedObject::removeReference
  1.4   xalanc_1_8::VariablesStack::findEntry
  1.3   xalanc_1_8::XPath::executeMore
  1.1   std::__find_if
  1.0   xalanc_1_8::XalanReferenceCountedObject::addReference
  1.0   xalanc_1_8::FunctionSubstring::execute
  1.0   xercesc_2_5::BaseRefVectorOf::elementAt
```

A very large program (in terms of lines of code), flat profile.

Results for the Floating-Point Benchmarks CFP2006

410.bwaves (Fortran program)

```
75.7   mat_times_vec_
14.2   bi_cgstab_block_
 6.1   shell_
 2.2   jacobian_
```

Peaky profile, top subroutine is quite small, overall a small program (Fortran77).

416.gamess (Fortran program)

Invocation 1 (about 23% of overall execution time)

```
19.9   dirfck_
13.3   forms_
11.7   genral_
 7.8   xyzint_
 7.3   genr70_
 3.8   rt123_
 3.3   shells_
 3.2   __exp
 3.0   _mcount
 2.5   twoei_
 2.5   dspdfs_
 2.0   mcount_single
 1.9   grdg80_
 1.7   ijprim_
 1.4   sp0s1s_
 1.3   tq0s1s_
 1.2   dabclu_
 1.2   zqout_
 1.1   intj2_
 1.1   jdxyzs_
```

416.gamess, invocation 2 (about 14% of time)

```
37.9   dirfck_
25.3   forms_
10.8   xyzint_
 6.6   genral_
 3.5   dspdfs_
 2.5   zqout_
 1.8   rt123_
 1.8   shells_
 1.3   ijprim_
 1.0   _mcount
 1.0   __exp
 1.0   dabclu_
```

416.gamess, invocation 3 (about 63% of time)

```
32.6   twotff_
21.7   forms_
10.0   dirfck_
 7.5   genral_
 7.3   xyzint_
 4.8   dirtrn_
 2.2   rt123_
 1.5   zqout_
 1.4   shells_
 1.2   __exp
 1.1   dspdfs_
 1.1   _mcount
 1.0   ijprim_
```

Somewhat peaky profile. Large, popular program but very old programming style with many standards violations.

433.milc (C program)

```
16.1   mult_su3_na
13.2   mult_su3_nn
 8.8   mult_su3_mat_vec
 8.7   mult_adj_su3_mat_vec
 7.0   scalar_mult_add_su3_matrix
 5.6   _mcount
 4.7   _memset
 4.5   su3mat_copy
 4.0   uncompress_anti_hermitian
 3.8   su3_projector
 3.6   mult_su3_an
 3.1   su3_adjoint
 2.8   u_shift_fermion
 2.1   mcount_single
 1.6   mult_su3_mat_vec_sum_4dir
 1.6   mult_adj_su3_mat_4vec
 1.5   add_su3_matrix
 1.2   eo_fermion_force
 1.0   scalar_mult_add_su3_vector
```

434.zeusmp (Fortran program)

```
37.4   hsmoc_
16.9   lorentz_
10.1   _mcount
 6.3   mcount_single
 5.3   __f95_sign
 3.9   momx3_
 2.7   momx2_
 2.4   momx1_
 2.4   tranx3_
 1.9   tranx1_
 1.7   tranx2_
 1.6   forces_
 1.6   avisc_
 1.5   newdt_
 1.1   ct_
```

Somewhat peaky program. The peak subroutine is well documented and seems to be carefully written. However, it makes use of COMMON and EQUVALENCE which are now considered old-fashioned Fortran programming style.

435.gromacs (C/Fortran program)

```
66.2   inl1130_
 8.8   ns5_core
 5.1   _mcount
 2.2   mcount_single
 1.9   put_in_list
 1.6   do_update_md
 1.5   inl1120_
 1.2   fsettle_
 1.2   inl1100_
 1.2   inl0100_
```

The top subroutine, and several others, are machine generated, not written directly by a human programmer.

436.cactusADM (C/Fortran program)

```
99.9   bench_staggeredleapfrog2_
```

Although this is a very peaky profile, the routine is moderately long: 810 lines, and the activity is spread out across it. It is machine-generated code.

437.leslie3d (Fortran program)

```
16.5   fluxk_
16.0   fluxj_
13.7   extrapj_
13.2   extrapi_
13.0   extrapk_
12.8   fluxi_
11.0   update_
 3.2   setbc_
```

Remark: One Fortran source module only.

444.namd (C++ program)

```
All routines below are from ComputeNonbondedUtil::
13.5   calc_pair_energy_fullelect
12.3   calc_pair_fullelect
10.0   calc_pair_energy
 9.6   calc_pair_energy_merge_fullelect
 9.3   calc_pair_merge_fullelect
 9.1   calc_pair
 7.4   calc_self_energy_fullelect
 6.6   calc_self_fullelect
 6.4   calc_self_merge_fullelect
 5.2   calc_self_energy
```

All of the above routines are actually instantiated from a single template in a `.h` file, which caused some challenges for profiling tools. The common source may cause some risk of low-hanging fruit, but the activity is distributed over several different loops in the file.

447.dealII (C++ program)

```
16.3   ConstraintMatrix::add_line
12.3   LaplaceSolver::Solver::assemble_matrix
 9.5   MappingQ1::compute_fill
 8.9   SparseMatrix::vmult
 5.7   std::_Rb_global::_M_increment
 5.0   contract
 4.7   std::__advance
 3.0   ConstraintMatrix::add_entry
 2.4   FiniteElementBase::compute_2nd
 2.3   Tensor::Tensor
 2.2   SparseMatrix::precondition_Jacobi
 1.4   Tensor::operator*
 1.4   std::_Rb_tree,std::less,std::allocator::insert_unique
 1.3   Tensor::operator+=
 1.2   QProjector::DataSetDescriptor::operator
 1.2   MappingQ::apply_laplace_vector
 1.1   Tensor::operator*=
```

450.soplex (C++ program)

Invocation 1 (about 48% of overall execution time)

```
17.1   SSVector::assign2productFull
13.9   _mcount
 6.8   CLUFactor::initFactorMatrix
 6.5   SPxFastRT::maxDelta
 5.6   SSVector::setup()
 5.1   _memset
 4.9   CLUFactor::vSolveUrightNoNZ
 3.7   SPxSteepPR::selectLeaveX
 3.2   CLUFactor::solveLleftNoNZ
 2.7   SSVector::clear()
 2.4   Vector&soplex::Vector::multAdd
```

450.soplex, invocation 2 (about 52% of time)

```
32.3   SVector::operator*
15.0   _mcount
12.6   SSVector::assign2productFull
 8.9   SPxSteepPR::selectEnterX
 4.6   Vector&soplex::Vector::multAdd
 4.1   SPxSteepPR::entered4X
 3.0   single
 2.6   SSVector::clear()
 2.6   SoPlex::test
 2.2   DataHashTable::index
 2.1   SSVector::setup()
```

453.povray (C++ program)

```
13.9   pov::Intersect_Sphere
12.6   pov::Intersect_Plane
 8.6   pov::All_CSG_Intersect_Intersections
 6.1   pov::Check_And_Enqueue
 4.9   pov::All_Sphere_Intersections
 4.4   pov::All_Plane_Intersections
 3.8   pov::DNoise
 3.2   pov::Inside_Plane
 3.0   pov::Inside_Object
 2.6   pov::Intersect_Quadric
 2.4   pov::Ray_In_Bound
 2.3   pov::priority_queue_insert
 2.1   pov::Priority_Queue_Delete
 1.9   pov::Intersect_BBox_Tree
 1.8   pov::Noise
 1.8   pov::Intersect_Light_Tree
 1.6   pov::Inside_Quadric
 1.5   pov::Intersection
 1.5   pov::compute_lighted_texture
```

454.calculix (C/Fortran program)

```
53.6   e_c3d_
18.1   DVdot33
 4.7   DVaxpy
 2.4   Chv_updateS
 2.4   _mcount
 2.1   Network_findAugmentingPath
 1.5   mcount_single
 1.1   __pow
```

The profile appears somewhat peaky, but top routine, e_c3d is long (1110 lines of code). It is from the application proper. The DV and Chv routines are from "SPOOLES", a public domain solver library included with the benchmark source. The inclusion of SPOOLES makes the directory tree somewhat complicated.

459.GemsFDTD (Fortran program)

```
21.7   update_mod.updatee_homo_
21.4   update_mod.updateh_homo_
18.4   upml_mod.upmlupdatee_
17.0   upml_mod.upmlupdateh_
14.4   nft_mod.nft_store_
 1.5   setexception
 1.1   _mcount
 1.0   huygens_mod.huygense_
```

465.tonto (Fortran program)

```
14.8   shell2_module.make_ft_1_
13.3   shell1quartet_module.form_esfs_no_rm_
 9.4   _mcount
 8.4   mcount_single
 3.1   __exp
 3.0   shell1quartet_module.make_esss
 2.9   gaussian2_module.make_ft_component_
 2.7   _libc_threads_interface
 2.6   shell1quartet_module.make_esfs_
 2.3   __z_exp
 2.2   crystal_module.sum_ft_ints_
 2.1   __sincos
 2.1   __k_sincos
 1.9   shell1quartet_module.make_ssfs
 1.8   __rem_pio2
 1.4   shell1quartet_module.make_r_jk_abcd_
 1.3   _malloc_unlocked
 1.2   shell1quartet_module.form_esps_no_rm_
 1.2   shell2_module.normalise_ft_
 1.1   rys_module.get_weights2_t2_
```

470.lbm (C program)

```
99.3   LBM_performStreamCollide
```

Very peaky profile. Top subroutine is 95 LOC. Very small overall static size, from a research environment. Together with the high memory pressure, the peaky profile and the small size may create incentives for special-case optimizations.

481.wrf (C/Fortran program)

```
16.0   module_advect_em.advect_scalar
 6.3   _mcount
 5.5   module_small_step_em.advance_uv_
 4.8   __powf
 4.6   module_small_step_em.advance_w_
 4.4   module_big_step_utilities_em.calc_cq_
 3.6   module_small_step_em.advance_mu_t_
 3.1   mcount_single
 3.1   module_em.rk_update_scalar_
 2.9   module_small_step_em.calc_p_rho_
 2.8   module_advect_em.advect_u_
 2.6   module_small_step_em.small_step_prep_
 2.6   module_advect_em.advect_w_
 2.5   module_small_step_em.sumflux_
 2.5   module_advect_em.advect_v_
 1.9   module_big_step_utilities_em.zero_tend_
 1.9   module_bl_ysu.ysu2d_
 1.7   __f95_signf
 1.7   module_big_step_utilities_
       em.horizontal_pressure_gradient_
 1.6   module_big_step_utilities_em.rhs_ph_
```

482.sphinx3 (C program)

```
37.9   mgau_eval
24.5   vector_gautbl_eval_logs3
 9.6   _mcount
 8.8   subvq_mgau_shortlist
 2.8   approx_cont_mgau_frame_eval
 2.5   mcount_single
 2.1   mdef_sseq2sen_active
 2.0   _memset
 1.8   dict2pid_comsenscr
 1.3   logs3_add
 1.2   approx_mgau_eval
```

Somewhat peaky profile, top subroutine is about 90 LOC

Some Concluding Remarks

One of the initial goals for CPU2006 was that the program should not spend more than 5% of its time outside of the supplied source code. If the benchmark profile is concentrated in the supplied code, there is more clarity as to what is being tested, and one may reduce the use of platform-specific, narrowly targeted code as may be found in some highly tuned platform libraries.

It can easily be seen that this 5% threshold was not met in a number of cases. Each case has led to a discussion in the subcommittee. When the program survived the selection process nonetheless, it was because of arguments such as (a) memory management operations (allocate, free, copy, clear) are commonly used in real life applications, for example by contemporary codes that instantiate and destroy objects as needed. (b) exponentiation is commonly used in quantum chemistry; that's simply a fact of life for the application area; (c) although a benchmark may show library time on one tested platform, that may be an artifact of that platform, rather than the fault of the code. In several cases, percentages varied widely between operating systems.

During development of CPU2006, benchmarks with "peaky" profiles (i.e. those with a high percentages for some subroutine) were of special interest. The subroutines in question were checked for aspects such as size (number of source lines), programming style, and cache misses. Sometimes, these observations and a review of the program's documentation indicated that a high locality just cannot be avoided for particular application areas. On the other hand, if compiler optimizations for the top subroutines of a program speed up a particular program to an unexpected degree, readers of benchmark results should check carefully whether such optimizations are useful for a larger set of programs. Note that SPEC requires that optimizations not be too narrowly targeted; see the CPU2006 Run Rules, [3], especially section 1.4.

Authors' Roles

The first author (RW) collected profiles through many baselevels of CPU2006 and provided frequent commentary to the subcommittee. During development of CPU2006, he served as the primary source of questions about peaky profiles. For this paper, he collected profiles for the majority of the benchmarks, namely those using method (1) in the "Methods" section, above. He wrote the first draft of this paper.

During CPU2006 development, the second author (JH) profiled many baselevels of the suite to determine whether the 5% criterion was met, and encouraged discussion about exceptions. For this paper, he contributed the profiles that used method (2), provided additional commentary, and edited the final draft.

Disclaimer

Opinions expressed in this article are personal opinions and do not necessarily reflect either SPEC or an employer's official policy.

References

[1] For information on collect, er_print, and related utilities see http://developers.sun.com/sunstudio/analyzer_index.html

[2] John L. Henning (ed.), "SPEC CPU2006 Benchmark Descriptions", Computer Architecture News, Volume 34, No. 4, September 2006.

[3] www.spec.org/cpu2006/docs/runrules.html