Function Call Synthetic Benchmarks

The function call synthetic benchmark suite consists of a set of small programs that will write large programs with a number and layout of functions defined by input arguments. The different components of the suite differ in the way the functions are linked together, the order they are invoked and whether direct or indirect call instructions are used to invoke the chain of function calls. The functions that get generated are approximately 60 instructions in length, typical of the average size used in large object oriented programs.

The generator creates a set of functions that create a 3 dimensional matrix. It is invoked as:
./build*.sh MAXI_I MAX_J MAX_K
Where I,J,K are the extents of the three dimensional array. The index K is the (maximum) call stack depth. The script will create I * J * K source files called FOO_I_J_K.c and FOO_main.c. It will then compile them and link them into a single static binary and a main program and one shared object per value of K.

The main program loops over the first two indexes calling the top function of the call chain of the third index. The call chains follow an incremental order (ex: FOO_i_j_k calls FOO_i_j_k+1) or, for some of the suite components, the order of the calls through the third index follow a random path. The random numbers used are non repeating and never reference themselves (ran_array[i] !=i).
As can be seen in the spreadsheet (Synthetic_call_benchamrk.xlsx) the statically linked binary (FOO_Static/FOO_att_static, compiled with icc/gcc respectively)) runs in 6 X10E10 cycles on the Intel® Xeon™ 5600 in my lab in Oregon, whioch is running Scientific Linux 5.3. The number of demand driven ifetch hits in L2 (L2_rqsts.ifethc_hit) and L3 (offcore_response.demand_ifetch.local_cache_1) are 2.8xE9 and 1.5XE8 respectively. The code is not retiring any uops for almost 2/3rds of the runtime (uops_retired.stall_cycles). The stalls attributed to ifetch hits in L2 and L3 would be evaluated as 6*2.8XE9 + 35*1.5XE8 = 2.2XE10, a bit more than half of what is observed.

However when the object files are linked into 100 shared objects such defined by including all functions at the same call stack depth (value of K in FOO_i_j_k) such that all calls cross between shared objects (FOO_dynamic/FOO_att_dynamic), then the time increase by 5.75 times to 3.45XE11. This can be attributed to the increase in demand ifetch hits to 9.91XE9, an increase of 67 times! Using a penalty of 35 cycles this would account for 3.47XE11 cycles, which is in reasonably good agreement with the increase in runtime. We note the increase in the rate of counting of br_misp_retired.all_branches which is equal to the increase in the number of branches, an increase entirely due to indirect branches needed for the unconditional jumps between shared objects.

If the functions are generated such that there call targets are "randomly" assigned amongst the 100 values of K, but with no repeating nor self referential calls, the runtime increase from 6XE10 to 1XE11, an increase that can be almost entirely attributed to the increase in L3 demand ifetch hits.

In the binary files are linked in lexical order due to the *.o used in the link command. This results in the calling functions being placed directly before the target fuctions in IP (instruction pointer). Consequently the L2 HW prefetcher is extremely successful in prefetching the cachelines of instructions for the soon to be invoked functions. This suppresses the L3 demand ifetch hits as the lines are in L2 by the time they are needed. This does not happen as effectively when the random order is applied and does not happen at all when the binaries are linked into dynamic shared objects. Thus this test illustrates the benefit of taking advantage of the l2 HW prefetcher for call dominated execution.

Even in the case of the binaries built with shared objects there is some HW prefetching done to good effect. Within each shared object the function of the next call chain follows the current call chain in the default build. Thus the transpose build causes the call chains to be invoked in a non lexical order, FOO_i+1_j_* are invoked after FOO_i_j_*, as opposed to FOO_i_j+1_* as is done in the default. This raises the execution time considerably

Finally, there are directories with the build components for functions that invoke the calls by the use of function pointer arrays. These cause indirect calls to be generated as can be seen from the performance event data in the spreadsheet.

The loop through the 10,000 functions and the 2 to 3 unconditional branches/calls that are invoked per function call clearly exceeds the capacity of the branch predictors for the indirect branches. This is easily seen with the performance events. 100 function deep call chains are not the norm, and the array of calls used for the lab (10 by 10 by 100 deep) is of course completely artificial. It does serve as a useful tool to understand instruction line delivery.