252 PROJECT SUGGESTIONS SPRING 1990

Tools

P1. Instruction statistics tools for 370 or 80x86. (Exercise 4.22) Some manufacturers have not yet seen the value of measuring instruction set mixes. Maybe you can help them. Pick a machine for which such a tool is not widely available, e.g., the IBM 370 or the Intel 80x86. (There are Sun 386i in the department and an IBM 3090 in the basement of Evans.) Construct one for that machine. Object code translation, as used in the MIPS compiler system [Chow 1986] is preferred because there is little slowdown for collection information. (We may be able get a version from Sun or MIPS for their architectures as the starting point). Otherwise, if the machine has a single-step mode—as in the VAX or 8086—you can use it to create your tool. The ideal test case would be the benchmarks in the text (GCC, Spice, and TeX). There can be several teams working on this project since they can pick several architectures.

P2. Link-time instrumentation for cache simulation. One of the limitations of cache traces is the phenomenal slowdown in execution to collect the information plus the storage space required to save. Borg et al [1990] propose adding statistics collecting instructions to basic blocks at linktime to collect cache information. This allows them to run collect information on any program that runs on their instruction sets for hundreds of times longer than any previous cache studies, with surprising results. While they need to rerun the program to try different cache parameters, it is so much faster that it takes much less time than trace based collection. Write such a linker for the MIPS or SPARC instruction sets and collect cache statistics of your own. Again, there can be several teams working on this project since they can pick several architectures.

P3. Improving DLX tools. The software for DLX (compiler, simulator, assembler) need to be improved. Last semester Ken Sheriff at UCB as part of an advanced course and people at other places worked on fixing the bugs. The main complaint seems to be with the instruction set simulator. The starting point is to collect all the proposed new versions of the DLX compiler and assembler and try to form a single better version. The second step is to modify the MIPS simulator developed by John Ousterhout for CS 60B to match the DLX instruction set. (This simulator is fast and well written, but lacks floating point instructions as well as being different from DLX.) The third step would be to significantly increase the number and size of C programs that compile, assemble, and run on the simulator.

P4. DLX instruction scheduler (Exercise 6.22) Write an instruction scheduler for DLX that works on DLX assembly language, e.g., remove NOPS due to delayed branches and schedule LOADs to reduce interlocks. Evaluate your scheduler using either profiles of programs or with a pipeline simulator. If the DLX C compiler does optimization, evaluate your scheduler's performance both with and without optimization. This optimization would start by improving code within basic blocks, and then schedule beyond beyond basic blocks. Depending on the scope of the difficulty of this project, there may be time to try performing other optimizations of the assembly language or as part of the C compiler.

P5. Adding vector instructions to the DLX simulator (Exercise 7.13) Extend the DLX simulator to be a DLXV simulator including the ability to count clock cycles. Write some short benchmark programs in DLX and DLXV assembly language. Measure the speedup on DLXV, the percentage of vectorization, and usage of the functional units.

P6. Adding vector checking to the DLX compiler (Exercise 7.14) Modify the DLX compiler to include a dependence checker. Run some scientific code and loops through it and measure what percentage of the statements could be vectorized.

Input/Output

P7 Collecting statistics on disk arm motion The often used random seek assumption for disks is debunked in Chapter 9. The purpose of this project is to collect much more evidence by modifying driver routines to record seek distances, in a variety of environments: timesharing, CAD, database, supercomputer, ... The task would be to modify a device driver to trace every access address per disk so that the trace can be processed for seek distributions as well as spatial and temporal hot spots.

P8 Measuring I/O patterns in supercomputers Supercomputer I/O has been characterized as sequential 1MB transfers. (See BUCHER, I. V. AND A. H. HAYES [1980]. "I/O Performance measurement on Cray-1 and CDC 7000 computers," *Proc. Computer Performance Evaluation Users Group, 16th Meeting*, NBS 500-65, 245–254.) Is this (still) an accurate characterization? The starting point would be collecting supercomputer applications known to be I/O intensive and then record the characteristics of their I/O: size, length of sequential accesses, reads vs writes, % of time in I/O, and so on.

P9 I/O overhead experiments Find the instructions per I/O required to deliver 8KB requests in various configurations. If this number is well-modeled, we will better understand the effect of MIPS in increasing throughput when machines have lots of disks. Some interesting comparisons might be:

1. Raw vs. file overhead

2. Cost of DMA for cache invalidation and memory bus utilization

3. Cycle cost of optimizations such as buffer cache management and seek optimizations

3. Instruction counts on different machines

4. The effect of different disk controllers

Most work could be done with trace and editing tricks. *(Suggested by Rich Clewett of Sun Microsystems.)*

P10 I/O performance experiments (Exercise 9.15) Take your favorite computer and write three programs that achieve the following:

1. Maximum bandwidth to and from disks
2. Maximum bandwidth to a frame buffer
3. Maximum bandwidth to and from the local area network

What is the percentage of the bandwidth that you achieve compared to what the I/O device manufacturer claims? Also record CPU utilization in each case for the programs running separately. Next run all three together and see what percentage of maximum bandwidth you achieve for three I/O devices as well as the CPU utilization. Try to determine why one gets a larger percentage than the others.

P11 Generalized scaling formulas Chapter 9 uses these formulas to try to estimate performance when performance of the system is improved when the possibility of overlap exists:

$$\text{Time}_{\text{scaled}} = \frac{\text{Time}_{\text{cpu}}}{\text{Speed-up}_{\text{cpu}}} + \frac{\text{Time}_{\text{i/o}}}{\text{Speed-up}_{\text{i/o}}} - \frac{\text{Time}_{\text{overlap}}}{\text{Maximum}(\text{Speed-up}_{\text{cpu}}, \text{Speed-up}_{\text{i/o}})}$$

(There are also formulas for best case and worst case estimates that are more complicated.) Come up with a simple and elegant formula for being able to predict performance when there are N components to execution time, where $N \geq 3$.

Cost and Performance

P12 Evaluate a "stall" cache There are several ways to fix the problem of loads and stores taking extra cycles in RISC machines that must occur when there is a single 32-bit bus to the outside world. One way is double cycle the bus as done in the MIPS R2000. Another way is to add an instruction cache on chip. Rather that include a complete instruction cache, Sun has come up with an idea called a *stall cache*. The cache only contains the instructions needed during a load or store access, i.e., the instructions that follow the load or store. (If you know what the branch target buffer is on the AMD 29000, this is the same philosophy for a different problem.) The conjecture is that a small fully associative cache (say 32 entries) will substantially reduce the impact of loads and stores while maintaining a single bus. Preliminary results suggest a 10% performance improvement and that random replacement is superior to LRU (see exercise 8.11 on page 494 for an explanation why this might be true.) Ed Frank on Sun Microsystems says he would love to work with a project team to fully analyze the idea with the goal of publishing a paper. *(Suggested by Ed Frank of Sun Microsystems.)*

P13 Distributed system cost What is the most economical way to configure distributed systems in terms of local memory, remote memory (in file system), local disk, remote disk, network bandwidth, and so on. This would require collecting costs of the options and then estimating a performance model. *(Suggested by Andy Bechtolsheim of Sun Microsystems.)*

P14 Benchmarking Cray instruction set architecture vs. newer architectures The Cray X-MP in Evans has hardware that allows it to count the number of instruction executions, clock cycles, and so on. Take the Perfect Club benchmarks (which are known to run on the Cray) and run them on MIPS or SPARC machines, collecting counts of instruction executions. Then take the SPEC benchmarks (which run on MIPS and SPARC) and run them on the Cray. Explore the relative difference in instruction counts with and without vectorization turned on for Cray for C vs Fortran programs. Look at the quality of the code on all machines for C and Fortran to understand the performance differences. It is also illustrative to calculate CPI for each benchmark as well as record execution time. Given the number of programs that to be run, the project can be performed by multiple teams, each going into more analysis of the differences rather than by adding more programs.

P15 Generalized Benchmarks Two popular benchmark kernels, Livermore Loops and Linpack, summarize performance as a single number. While providing some information, this is not enough to understand what is really going on. In addition, for certain types of architectures, performance is a function of input size. However, these two programs measure execution time for only a few input sizes. This project involves rewriting either or both benchmarks to provide more information about processor performance rather than that single number. Benchmarked machines could include the CRAY X-MP. For more information, see Corinna Lee (cori@ginger).

P16 Synthetic SPEC benchmark One of the problems of the SPEC benchmark suite is that they take hours or days to run on a simulator of a new machine. If someone could come up with a program that ran in 10,000 cycles that would accurately predict the SPEC rating they would have the undying admiration of microprocessor designers around the world. The idea would be to measure the SPEC benchmarks by all perspectives and then write an **assembly** language program that has the proper profile for a particular architecture. The idea of write an assembly language program is to prevent targeted compilers from discarding most of the synthetic program. The proper profile include instruction mix, cache miss rate, branch taken frequency, instruction sequence dependences for superscalar machines, “density” of floating point operations, and anything else you can think of to make it realistic. The idea would be to try this program on a variety of models to see how accurately it predicts performance. *(Suggested by Jim Slager of Sun Microsystems.)*