The Jrpm System for Dynamically Parallelizing Java Programs

Summary:

Previous work has shown the benefits of thread level-speculation. These papers built a run-time system to detect the dependencies between the speculatively scheduled threads and had mechanisms to recover from a dependency violation. The ideal candidates for speculative threads were function calls and loops. These papers either identified the prospective candidates either manually or by static techniques using the compiler. The compiler identified potential threads and annotated the binary with the thread information. This paper focuses on identifying threads at runtime. The authors have developed a hardware profiler that profiles the code at runtime and identifies candidate threads and then the code is dynamically recompiled with the speculation information.

Details:

The runtime profiler and threading system is built around a Java runtime environment. On identifying the prospective speculative thread loops based on the information gathered from the profiler, the code is dynamically compiled. This was feasible because of the java runtime system. The java bytecodes were dynamically recompiled into the native architecture.

Runtime profiling is done by instrumenting the code. Some special instructions used by the runtime hardware profiler are added during the compilation. There hardware profiler looks for potential dependencies between threads using timestamp counters. It also looks at the amount of speculative state that needs to be buffered. Candidates are chosen based on this information to incur low overhead during speculative execution. The paper also provides some optimizations that contributed to performance gains significantly. The hardware profiler picked those loads and stores that frequently caused a dependence violation and introduced synchronization between those and loads and stores in the dynamically recompiled code. The authors found that in such cases the overhead of synchronization was much less than that of squashing the thread.

They also provide other optimizations like allocating loop invariants in registers. They also provide mechanisms to compute reduction operations locally and merge them at the end. Another contribution made by the authors is a parallel garbage collector.

The authors show some speedups due to this mechanism and also acknowledge the fact that the hardware profiling which is done during the sequential execution does slow down execution.

Some issues:

In this paper the authors profile the code at runtime at the start of the program. This information is used to capture the threads. Once this done the rest of the execution is done based on this information. This approach might fail if the program exhibits phased behavior. The speculative thread candidates chosen initially may not be good candidates in another phase of the program. The authors should have detected phase changes based on IPC or other means and then re-profiled the code.

The paper does not clearly provide the hardware mechanisms used to detect the load dependencies using time-stamp counters.

Jrpm does not support TLS in the kernel and hence I/O intensive programs that have a lot of system call activity and spend a lot of time in the kernel do not benefit from this mechanism.