SUDS: Software Based Memory Speculation for Raw Microprocessors

Matt Frank MFRANK@LCS.MIT.EDU
MIT-LCS Computer Architecture Group

1. Introduction[1]

Programming a parallel computer is hard work. One solution to this problem is to allow the programmer to write sequential programs and then let the system speculate that the program is parallel. At runtime, the system executes a chunk of the program in parallel. Next, the system checks whether the parallel execution produced a result consistent with sequential semantics. If the parallel execution was correct the system moves on to the next chunk of the program and repeats the process. Otherwise, the execution is rolled back to the state at the beginning of the chunk, and the chunk is rerun sequentially.

SUDS (Software Un-Do System) is a memory speculation system for Raw microprocessors. The SUDS system includes both a software runtime system for managing speculative parallelism and a compiler that finds opportunities for renaming. Because the compiler eliminates most of the work of renaming, the software runtime system is efficient enough to achieve parallel speedups.

SUDS is designed to run on *Raw microprocessors*. A Raw microprocessor is a single chip VLSI architecture, made up of an interconnected set of tiles. Each tile contains a simple RISC-like pipeline, instruction and data memories and is interconnected with other tiles over a pipelined, point-to-point mesh network. The network interface is integrated directly into the processor pipeline, so that the compiler can place communication instructions directly into the code. The software can then transfer data between the register files on two neighboring tiles in just 4 cycles (Lee et al., 1998).

2. Example

Figure 1 shows an example of a simple loop with non-trivial dependences. SUDS partitions Raw's tiles into two groups. Some portion of the tiles are designated *worker* nodes, the rest are designated *memory* nodes. SUDS parallelizes loops by cyclically distributing the loop iterations across the worker nodes. We call the set of iterations running in parallel, one iteration per worker node, a *chunk*.

[1]Presented at the MIT Student Oxygen Workshop, July 16, 2001

```
for (i = 0; i<N; i++)
  u = A[b[x]]
  A[c[x]] = u
  x = g(x)
```

Figure 1. An example loop.

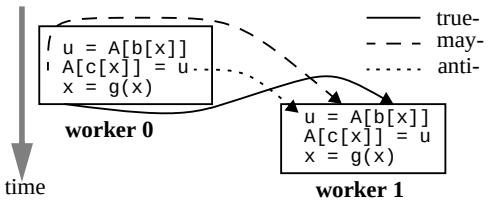

Figure 2. SUDS runs one iteration of the loop on each worker node. In this case the dependences between iterations limit the available parallelism.

Figure 2 shows an initial attempt at parallelizing the loop on a machine with two workers. The figure is annotated with the dependences that limit parallelism. The variable `x` creates a *true-dependence*, because the value written to variable `x` by worker 0 is used by worker 1. The read of variable `u` on worker 0 causes an *anti-dependence* with the write of variable `u` on worker 1. Finally, the reads and writes to the `A` array create *may-dependences* between the iterations. The pattern of accesses to the array `A` depends on the values in the `b` and `c` arrays, and so can not be determined until runtime. Without any further support, any of these three dependences would force the system to run this loop sequentially.

Figure 3 shows the loop after two compiler optimizations have been performed. First, the variable `u` has been renamed `v` on worker 1. This eliminates the anti-dependence. Second, on both worker 0 and worker 1, temporary variables, `s` and `t`, have been introduced. This allows worker 0 to create the new value of variable `x` earlier in the iteration, reducing the length of time that worker 1 will need to wait for the true-dependence. The final remaining dependence is the may-dependence on the accesses to array `A`.

This remaining may-dependence is monitored at runtime.

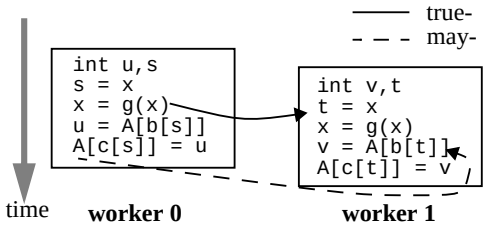

Figure 3. After renaming the anti-dependence is eliminated and the critical path length of the true-dependence is shortened.

The system executes the array accesses in parallel, even though this may cause them to execute out of order. Each of these speculative memory accesses is sent to one of the memory nodes. The runtime system at the memory nodes checks that the accesses are independent. If not, execution is temporarily halted, the system state is restored to the most recent checkpoint and several iterations are run sequentially to get past the mis-speculation point. Because the system is speculating that the code contains no memory dependences, this technique is called *memory dependence speculation* (Franklin & Sohi, 1996).

3. Design

Each of the three dependence types discussed in the example, anti-, true- and may-dependences, are handled by a different SUDS subsystem. SUDS handles anti-dependences by compiler based renaming. The SUDS runtime provides support for renamed variables by allocating a local stack on each worker. SUDS handles true-dependences at runtime by explicitly checkpointing them and then forwarding the data from worker to worker through Raw's point-to-point interconnect. The remaining may-dependences are those that the compiler is unable to analyze further. They are handled at the memory nodes using a runtime memory dependence validation protocol based on basic timestamp ordering (Bernstein & Goodman, 1980).

Since SUDS can handle anti- and true- dependences more efficiently than may-dependences, the goal of the SUDS compiler is to move as many objects as possible into the more efficient categories. It does this using three main compiler optimizations. *Privatization* uses dataflow analysis to identify objects whose live ranges do not extend outside the body of a loop. It also identifies several kinds of true-dependences and loop invariant objects. Additional support is provided for renaming non-scalar objects by taking advantage of scoping information. *Critical path reduction* improves program parallelism in the face of true-dependences, as shown in Figure 3. We introduce additional temporary variables that will hold the old value of the object while the new value is computed and forwarded to other, waiting, workers. *Register promotion* performs partial redundancy elimination on load and store instructions (Cooper & Lu, 1997), reducing the number of may-dependence requests sent to the memory nodes.

4. Status

The SUDS system has been operational for several months. The system runs on a cycle accurate behavioral simulation of a Raw microprocessor. Several sparse-matrix and linked-list style applications run on the system and achieve better than 2x speedups. In the current version of the system, the programmer tells the system which loops to parallelize. SUDS will attempt to parallelize any loop, even "do-across" loops, loops with true-dependences, loops with non-trivial exit conditions and loops with internal control flow.

Raw microprocessors provide a number of features that make them attractive targets for a memory dependence speculation system like SUDS. First, the low latency communication path between tiles is important for transferring true-dependences that lie along the critical path. In addition, the independent control on each tile allows each processing element to be involved in a different part of the computation. In particular, some tiles can be dedicated as worker nodes, running the user's application, while other tiles are allocated as memory nodes, executing completely different code as part of the runtime system. Finally, the many independent memory ports available on a Raw machine allow the bandwidth required for supporting renamed private variables and temporaries in addition to the data structures that the memory nodes require to monitor may-dependences.

References

Bernstein, P. A., & Goodman, N. (1980). Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems. *Proceedings of the Sixth International Conference on Very Large Data Bases* (pp. 285–300). Montreal, Canada.

Cooper, K. D., & Lu, J. (1997). Register Promotion in C Programs. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation* (pp. 308–319). Las Vegas, NV.

Franklin, M., & Sohi, G. S. (1996). ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers, 45*, 552–571.

Lee, W., Barua, R., Frank, M., Srikrishna, D., Babb, J., Sarkar, V., & Amarasinghe, S. (1998). Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems* (pp. 46–57). San Jose, CA.