SUDS: Primitive Mechanisms for Memory Dependence Speculation

Matthew Frank, C. Andras Moritz, Benjamin Greenwald,
Saman Amarasinghe and Anant Agarwal
MIT-LCS
Cambridge, MA 02139
`mfrank@lcs.mit.edu`

Abstract

As VLSI chip sizes and densities increase, it becomes possible to put many processing elements on a single chip and connect them together with a low latency communication network. In this paper we propose a software system, *SUDS* (Software Un-Do System), that leverages these resources using speculation to exploit parallelism in integer programs with many data dependences. We demonstrate that in order to achieve parallel speedups a speculation system must deliver memory request latencies lower than about 30 cycles. We give a cost breakdown for our current working implementation of SUDS that has a memory request latency that is nearly able to meet this goal.

In addition, we identify the three primitive runtime operations that are necessary to efficiently parallelize these programs. The subsystems include (1) a fast communication path for true dependences within the program, (2) a method for renaming variables that have anti and output dependences and (3) a memory dependence speculation mechanism to guarantee that parallel accesses to global data structures don't violate sequential program semantics. We find that these three subsystems do not interact, so that they can be implemented separately. Each subsystem is then simple enough that it can be built in software using only minimal hardware support. In this paper we focus on the memory dependence subsystem and demonstrate that it can be implemented using a simple but effective low-cost protocol.

1 Introduction

As we move towards the technology that will permit a billion transistors on a chip, computer architects need to face three converging forces. These include the need to keep internal chip wires short so that clock speed will scale with feature size, the economic constraints of quickly verifying new designs, and changing application workloads that emphasize stream-based multimedia computations. A Raw machine [22, 37] is a software-exposed VLSI architecture comprising a mesh-connected set of tiles each with a processing element and a portion of the on-chip memory. These architectures require only short wires, are much simpler to design than today's superscalars, and provide efficient parallelism for multimedia and signal processing applications.

In this paper we propose SUDS (Software Un-Do System), a software system that provides runtime support for parallelizing integer programs on Raw processors. Unlike multimedia and DSP applications, which can be easily parallelized because they tend to contain few dependences, integer programs are characterized by having a large number of dependences at all levels. In order to parallelize an integer program the system must provide runtime mechanisms for dealing with these dependences. In this paper we characterize the problems with dependences and examine mechanisms for handling them. We propose dividing the runtime dependence-handling system into three subsystems, each of which can then be addressed using simple software mechanisms on a Raw architecture.

Three subsystems are required because dependences constrain parallelism in three different ways. Most importantly, the true dependences in the program form a critical path that must be processed sequentially. In addition, reuse of static variable names in the program causes write-after-read and write-after-write dependences at runtime. Finally, there are *potential* dependence arcs in the program that are caused by random accesses to global data structures. These dependences may or may not exist at runtime given the input data to the program, and the existence of a dependence arc can only be determined at runtime.

SUDS parallelizes code using a technique similar to the Multiscalar architecture [9, 31]. In SUDS each loop is parallelized by cyclically distributing loop iterations across the system's processing elements. In order to simplify the runtime protocols SUDS breaks the execution of the program into *chunks*. The processing elements each run a single iteration of the chunk in parallel and then all the nodes synchronize. True dependences between loop iterations are forwarded in just a few cycles using the mesh interconnect between neighboring processing elements.

SUDS takes a software intensive approach to renaming anti and output dependences. A compiler is used to perform privatization analysis on all loop variables. Those that are found to be privatizable are placed on a special local stack. The runtime system creates a low-cost cactus stack by simply allowing each processing element to build a private copy of the local stack in its own local memory.

Finally, SUDS can speculate across memory dependences that can not be analyzed at compile time. SUDS dedicates a subset of the system elements to act as "speculative memories". All memory requests are sent through these nodes. The speculative memories validate the dependence speculations and hold a log of write requests which can be rolled back in the case of a mis-speculation. In order to simplify the validation and speculation protocols SUDS occasionally synchronizes all the nodes. In the results section we show that this additional synchronization results in only minimal load-imbalance overheads.

This paper makes two contributions. First, we show encouraging empirical results that indicate that with a reasonable engineering effort software based speculation systems might achieve parallel speedups. In addition we demonstrate how to build a simple, yet effective, low-cost memory dependence speculation protocol. This simple protocol is made possible because we separate the dependence handling problem into three primitive subproblems, each of which can be implemented independently of the others.

**MIT/LCS Technical Memo,
LCS-TM-591,
January 6, 1999.**

The rest of this paper is organized as follows. The next section describes the programming model that SUDS supports and the basic hardware and compiler support required by SUDS. Section 3 describes the techniques SUDS uses to forward true dependences, rename false dependences and speculate across potential dependences. Section 4 contains both a detailed cost breakdown of the basic mechanisms, and an empirical study of how memory latencies affect the available parallelism. Section 5 compares SUDS to previous memory dependence speculation systems. Section 6 concludes.

2 Support for SUDS

SUDS parallelizes loops by cyclically distributing the loop iterations across the system's processing elements. The main focus of this paper is the runtime support that SUDS provides for handling data dependences, described in Section 3. In this section we describe the environment in which SUDS runs. The parallelization system is based on the assumption that the underlying hardware includes certain mechanisms. Primarily these include the assumption that it provides a MIMD like parallel programming model, where there are multiple threads running in parallel and communicating using message passing. The particular hardware for which SUDS is targeted is discussed in Section 2.1.

The parallelization model that our system supports is based on loops. In the current version of our system, the programmer is responsible for identifying which loops the system should attempt to parallelize. This is done by marking the loops in the source code. The parallelization techniques we use work with any loop, even "do-across" loops, loops with loop carried dependences, loops with non-trivial exit conditions and loops with internal control flow. The system will attempt to parallelize any loop even if the loop contains no available parallelism due to data or control dependences. Our future plans include a system to automatically identify loops that are likely to contain significant amounts of parallelism.

A potential performance limitation of our system is that the compiler does not currently do renaming on global scalars. This can severely limit the available parallelism in programs written using programming styles that communicate extensively using global variables. We discuss this limitation in more detail in Section 3.4. For the programs we investigate in this paper, which are written in an "object-oriented" programming style, we have not had a problem with this limitation. We are currently working on a compiler algorithm that uses the results of an existing pointer analysis system to perform compiler renaming even for programs that rely exclusively on global variables.

2.1 Raw machines

The hardware we are targeting is a highly parallel single chip VLSI architecture. The processor as a whole is made up of an interconnected set of tiles (Figure 1). 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. Designs of this type have been called *Raw architectures* because they expose control of all of the communication resources directly to the compiler [22, 37].

This architecture is excellent for signal processing applications because it provides enormous amounts of compute bandwidth, and many independent memory ports. A Raw machine also provides

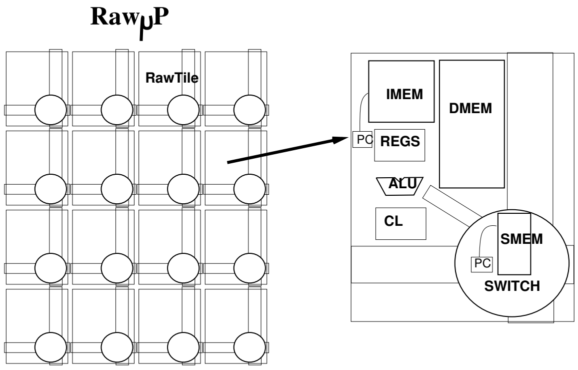


Figure 1: RawμP composition. The system is made up of multiple tiles. Each tile contains a processor pipeline with an integrated network interface and instruction and data memories.

three features that are invaluable for building a dependence handling system. First, the low latency communication path between tiles is important for transferring true-dependences that lie along the critical path. Second, 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 to performing memory dependence speculation while other tiles actually perform computation for the application. Finally, the many independent memory ports available on a Raw machine allow the high bandwidth required for supporting both a cactus stack and parallel access to global data structures.

3 Design

This section describes the design of the SUDS system. Achieving the best total system performance requires exposing enough application parallelism to use all of the system resources while minimizing overheads in the runtime system. We have achieved this goal using a number of techniques. First, we have tried to do as much work in the compiler as possible to relieve unnecessary runtime costs. For example, most of the work of memory renaming in our system is done in the compiler. In addition, when choosing protocols we have leaned towards those that have low overhead per invocation, rather than those that expose the most application parallelism. Finally, we have found that the handling of data dependences can be divided into three primitive runtime operations, fast-forwarding of true dependences, renaming of false dependences, and memory dependence speculation for potential dependences. The separation of renaming from memory dependence speculation has exposed many simplifications in the underlying system.

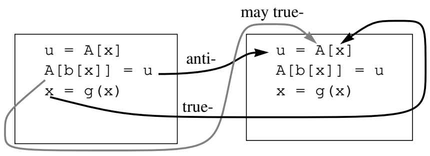
3.1 Chunk based work distribution

The SUDS system partitions the processing elements of the underlying hardware system into two groups. Some portion of the elements are dedicated as *compute* nodes. The rest are dedicated as *memory* nodes. One of the compute nodes is designated as the *master* node, the rest are designated as *workers* and sit in a dispatch loop waiting for commands from the master. The master node is responsible for running all the sequential code.

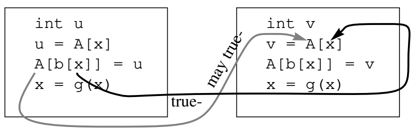
When a parallel loop is encountered the master is responsible for telling all the workers which loop to run. Each of the compute nodes is responsible for running a single iteration of the loop. We call the set of iterations running in parallel a *chunk*. The compute nodes each run a single loop iteration, and then all the nodes synchronize through the master node. While this synchronization

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

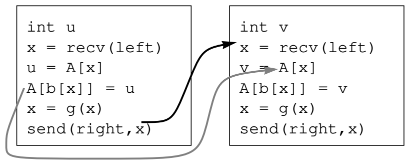
(a) Example loop nest



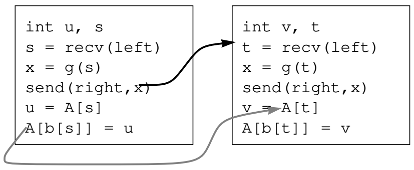
(b) Dependences in two adjacent iterations



(c) After renaming to eliminate the anti-dependence



(d) After forwarding the true-dependence



(e) After optimizing to reduce critical path length

```
int u, s
 s = recv(left)
 x = g(s)
 send(right,x)
 send(&A[s]&mask,{ld,&A[s]})
 u=recv(&A[s]&mask)
 send(&A[b[s]]&mask,{st,&A[b[s]],u})
```

```
int v, t
 t = recv(left)
 x = g(t)
 send(right,x)
 send(&A[t]&mask,{ld,&A[t]})
 v=recv(&A[t]&mask)
 send(&A[b[t]]&mask,{st,&A[b[t]],v})
```

```
while (true) {
 {type, pid, address, st_data} = recv()
 if (type == LD) {
   if (pid >= last_writer[address]) {
     last_reader[address] = pid
     send(pid, data[address])
   } else {
     MIS-SPECULATION DETECTED!
   } else if (type == ST) {
   if (pid >= last_reader[address]) {
     if (pid >= last_writer[address]) {
       last_writer[address] = pid
       data[address] = st_data
     }
     /* (pid < last_writer)
            => IGNORE BY THOMAS WRITE RULE */
   } else {
     MIS-SPECULATION DETECTED!
   }
  }
}
```

```
while (true) {
 {type, pid, address, st_data} = recv()
 if (type == LD) {
   if (pid >= last_writer[address]) {
     last_reader[address] = pid
     send(pid, data[address])
   } else {
     MIS-SPECULATION DETECTED!
   } else if (type == ST) {
   if (pid >= last_reader[address]) {
     if (pid >= last_writer[address]) {
       last_writer[address] = pid
       data[address] = st_data
     }
     /* (pid < last_writer)
            => IGNORE BY THOMAS WRITE RULE */
   } else {
     MIS-SPECULATION DETECTED!
   }
  }
}
```

(f) After using SUDS to eliminate the may true dependence.
The psudo-code for two SUDS memory nodes are also given.

Figure 2: An example of how SUDS parallelizes a simple loop.

produces some load imbalance it also dramatically simplifies the protocols for checkpointing of true dependences and the validation of memory dependence speculations as described in Sections 3.3 and 3.5.

The application address space is cyclically distributed across the memory nodes at the word (32 bit) granularity. During sequential sections of the program the memory nodes simply process each memory request and reply with either load data or a store acknowledgement. During parallel loops memory requests may arrive out of order, and the memory nodes must both validate that these requests don't violate sequential order and provide facilities for rolling back to a consistent state in the case that a dependence violation is detected.

3.2 Example

Figure 2 gives a step by step demonstration of how our compiler transforms a loop so that the runtime system can exploit the available parallelism. The loop contains a true dependence on the variable `x`, an anti-dependence on the variable `u` and a potential dependence on the accesses to the global array `A`. The system is based on the idea that the loop iterations will be distributed cyclically along the compute nodes.

First the compiler peforms renaming on the private variable `u` to eliminate the anti-dependence (Figure 2(c)). Second the compiler inserts communication instructions so that compute node on the left can forward the value of variable `x` to the compute node on the right (Figure 2(d)). Finally we perform an optimization, described in Section 3.3 to reduce the critical path length between the receive of variable `x` and the corresponding send operation in the same iteration (Figure 2(e)).

Figure 2(f) shows how the code might be mapped onto four processing elements. The two elements on the top act as compute nodes, while the two processing elements on the bottom act as memory nodes, and run the memory dependence validation protocol described in Section 3.5.

3.3 Forwarding true dependences

The task of identifying loop carried true dependences is carried out by the compiler in our system. Currently, our compiler uses standard data flow analysis techniques to identify scalar loop carried dependences [26]. Any scalar variables modified within the loop nest, need to be either privatized by renaming (see Section 3.4) or forwarded to the next iteration. If the compiler finds that there is an inter-iteration dependence on a particular variable it inserts explicit communication instructions into the code. The compiler uses an analysis similar to that used by T.N. Vijaykumar for the Multiscalar [36] to identify the optimal placement of communication instructions.

At runtime the master node checkpoints all the loop carried dependences so that if a mis-speculation occurs the computation can be rolled back to a consistent state. Since only the master checkpoints this cost can be amortized over a number of loop iterations. The drawback of this approach is that when a mis-speculation does occur we may need to rollback slightly further than necessary. So far we have not found this to be a problem. As discussed in Section 4, in the programs we have examined, the ratio of misspeculations per chunk is low enough that it does not constrain parallelism.

Currently, our compiler only performs dependence analysis for scalars. For affine array accesses a compiler could perform array dependence analysis to identify array based dependences. Note that this compiler analysis can be optimistic since forwarding non-dependent data does not violate sequential semantics of the program. This is optimistic in the sense that it permits the register allocation and low-latency forwarding of the data values in question, rather than completing a more expensive round trip request to one of the memory nodes. However, to reduce the critical path cost, the compiler needs to identify a minimal set of values to be forwarded.

Another important compiler optimization is the reduction of critical path lengths of true dependence chains. Many typical loops use the forwarded values early in an iteration while preventing early forwarding by not updating the values until late in the iteration. In the worst case, this will completely sequentialize the loop. However, in many cases it is possible to calculate and forward the new value as soon as the value from the previous iteration is received. The compiler can reduce the critical path length by making a copy of the dependent variable, then modifying the variable, and finally transforming all the uses of the variable to uses of the copy.

3.4 Renaming

Many scalars, arrays and other data structures that are used to propagate values within a loop iteration are normally defined outside the scope of the loop body. This will create anti- and output-dependences across the iterations, forcing the loop to be serialized. Compiler analysis can identify these variables by performing scalar privatization and array data-flow analysis. body [28, 23, 24, 34]. Privatizable variables are allocated locally in each processor and never communicated outside.

This local allocation is performed by creating a second *safe-stack* on which privatizable variables may be placed. At runtime each compute node maintains its own local safe-stack and all privatizable variables are accessed off of this stack. This has two benefits. First, this provides a cheap form of renaming. Each privatizable variable can now be independently addressed on each compute node. There is an additional caching benefit in that variables that memory references to the safe-stack are completely local to the compute node, and do not need to be communicated to the memory nodes.

3.5 Memory dependence speculation

The memory dependence speculation system is in some ways the core of the system. It is the fall back dependence mechanism that works in all cases, even if the compiler can not analyze a particular variable. Since only a portion of the dependences in a program can be proved by the compiler to be privatizable or loop carried dependences, a large fraction of the total memory traffic will be directed through the memory dependence speculation system. As such it is necessary to minimize the latency of this subsystem.

The method we use to validate memory dependence correctness is based on Basic Timestamp Ordering [4] a traditional transaction processing concurrency control mechanism. As shown in Figure 3, each processing element which is dedicated as a memory dependence node contains three data structures in its local memory. The first is an array which is dedicated to storing actual program values. The next is a small hash table which is used as a *timestamp cache* to validate the absence of memory conflicts. Finally, there is a *log* which contains a list of the hash entries that are in use and the original data value from each memory location that has been modified. At the end of each chunk of parallel iterations the log is used to either commit the most recent changes permanently to memory, or to roll back to the memory state from the begining of the chunk.

The validation protocol works as follows. Each memory location has two timestamps associated with it, one indicating the last time a location was read (`last-read`) and one indicating the last time a location was written (`last-written`). As each load request arrives, its timestamp (`read-time`) is compared to

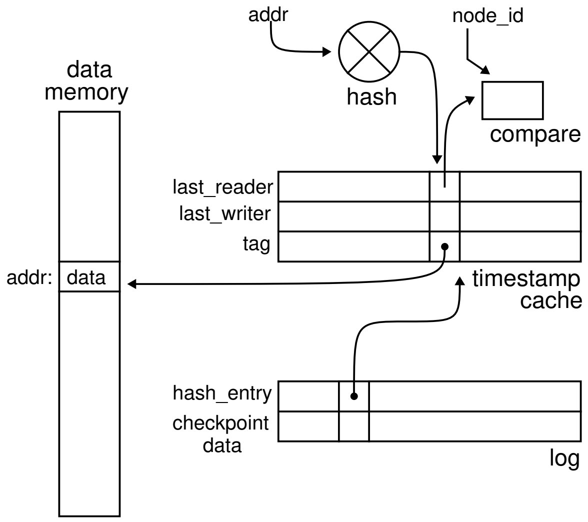


Figure 3: Data structures used by the memory dependence speculation subsystem.

the `last-written` stamp for its memory location. If `read-time` $\geq$ `last-written` then the load is okay and `last-read` is updated to `read-time`, otherwise the system flags a mis-speculation and aborts the current chunk.

On a store request, its timestamp (`write-time`) is compared first to the `last-read` stamp for its memory location. If `write-time` $\geq$ `last-read` then the store is okay, otherwise the system flags a mis-speculation and aborts the current chunk. So that a store can be rolled back in the case of a later abort, the old value of the memory location is copied into the log before the new store request is executed.

We have implemented an optimization on store requests that is known as the Thomas Write Rule [4]. This is basically the observation that if `write-time` $<$ `last-written` then the value being stored by the current request has been logically over-written without ever having been consumed, so the request can be ignored. If `write-time` $\geq$ `last-written` then the store is okay and `last-written` is updated as `write-time`.

The fact that SUDS synchronizes the processing elements between each chunk of loop iterations permits us to simplify the implementation of the validation protocol. In particular the synchronization point can be used to commit or roll back the logs and reset the timestamp to 0. Because the timestamp is reset we can use the requester's physical node-id as the timestamp for each incoming memory request.

In addition, the relatively frequent log cleaning means that at any point in time there are only a small number of memory locations that have a non-zero timestamp. To avoid wasting enormous amounts of memory space storing 0 timestamps, we cache the active timestamps in a relatively small direct-mapped hash table. Each hash table entry contains a pair of `last-read` and `last-written` timestamps and a cache-tag to indicate which memory location owns the hash entry.

As each memory request arrives, its address is hashed. If there is a hash conflict with a different address the validation mechanism conservatively flags a mis-speculation and aborts the current chunk. If there is no hash conflict the timestamp ordering mechanism is invoked as described above.

Log entries only need to be created the first time a chunk touches a memory location, at the same time an empty hash entry is allocated. Future references to the same memory location do not need to be logged, as the original memory value has already been copied to the log.

Operation	Cost
Dispatch	12
Validation	29
Log allocation	9
Message Reply	17
Log clean	7
Total	74

Table 1: Amortized cost breakdown for a load operation. Subsequent loads to the same address do not require additional log allocation or cleaning and therefore require only 58 cycles.

In the common case the chunk completes without suffering a mis-speculation. At the synchronization point at the end of the chunk, each memory node is responsible for cleaning its logs and hash tables. It does this by walking through the entire log and deallocating the associated hash entry. The deallocation is done by resetting the timestamps in the associated hash entry to 0.

If a mis-speculation is discovered during the execution of a chunk, then the chunk is aborted and a consistent state must be restored. Each memory node is responsible for rolling back its log to the consistent memory state at the end of the previous chunk. This is accomplished by walking through the entire log, copying the checkpointed memory value back to its original memory location. The hash tables are cleaned at the same time.

In the next section we give a breakdown of the costs of the memory dependence speculation system we have implemented, and show that these costs are nearly low enough to achieve parallel speedups.

4 Results

In this section we present results demonstrating that there is hope of building a software based speculation system that can achieve parallel speedups. In Section 4.1, we present a detailed cost breakdown of our working implementation of the SUDS memory dependence speculation subsystem. In our current implementation the total worst-case overhead per load operation is about 75 cycles. In Section 4.2, we present simulation results that demonstrate that for the applications we are considering, this latency is within a factor of about 2 of what is required to achieve parallel speedups. That is to say, these applications still demonstrate speedups with memory latencies of up to about 30 cycles. First, we show how the applications speed up given an unrealistically perfect memory system, one where every memory operation takes exactly one cycle. Then we show how each application performs as we vary the memory latency.

4.1 Cost breakdowns

In this section we break down the costs for each memory dependence speculation operation in our current implementation of the SUDS runtime system. The memory dependence speculation module was originally written in C. For this study we compiled the code with the SunPro Compiler version 4.2 at optimization level 5, and performed some further hand optimization. We then hand counted the number of cycles for the resulting code assuming a simple 5 stage scalar pipeline. This kind of pipeline is representative of the kind of processing element we expect to be available on each tile of a Raw processor [22, 37].

Operation	Cost
Rollback	10
Log clean	7
Total	17

Table 2: Amortized cost breakdown for a rollback operation.

Table 1 shows the breakdown in costs for a load operation. For each incoming request the system first needs to dispatch on the request type. The dispatch code has been optimized in favor of load operations because these operations are on the critical path. For a load request the dispatch requires 12 cycles.

The total cost for both validation and the actual memory access is 29 cycles. If this is the first access to the memory location since the last log cleaning operation, then the system must allocate a new log entry at the cost of 9 cycles. This cost is not incurred on subsequent accesses to the same memory location. Finally, a reply message is constructed and launched to the processing element that made the original load request at a cost of 17 cycles.

Later when the chunk of parallel iterations finishes, the system must incur an additional cost to clean the logs for each memory location touched during the parallel chunk. This cleaning operation costs 7 cycles per log entry.

In sum a load request requires a total of 74 cycles of processing from the memory node. For a second load request to the same memory location the log entry neither needs to be allocated or later cleaned, so these additional load requests require only a total of 58 cycles of processing from the memory node.

Table 2 shows the cost of performing a rollback operation in the case that a mis-speculation occurs and the chunk of parallel iterations needs to be rolled back. This operation is performed in bulk for all the memory addresses in the log, so the message dispatch cost can be amortized over a large amount work. For each entry in the log the checkpointed memory value needs to be copied back to memory at a cost of 10 cycles. In addition the hash entry associated with each log entry must be cleaned at an additional cost of 7 cycles. The total cost of rolling back to a consistent memory state after a mis-speculation is then the sum of these two costs, 17 cycles per memory location touched.

Discussion There are several methods that we are investigating to reduce the average latency of each load request. The first is to modify the protocol so that the actual data value is returned to the requester before validation is performed. We believe that this optimization ould reduce the latency of each load operation to about 30 cycles, but it would not change the bandwidth requirements placed on the memory nodes. A problem with changing the protocol in this way is that it requires the master node to perform an extra synchronization tep with the memory nodes at the end of each chunk to determine whether a mis-speculation has occured.

Another technique that has been proposed by other researchers for reducing both the average latency per memory request and also the bandwidth requirements on the memory is to place a small cache at the processing elements to exploit locality in the memory reference stream. A problem with this approach is keeping the caches coherent with one another, and there has been a great deal of active research in this area [8, 13, 32, 20, 17, 14]. We are investigating a technique that would allow SUDSto cache read-mostly values by allowing the system to "permanently" mark an address in the timestamp cache.

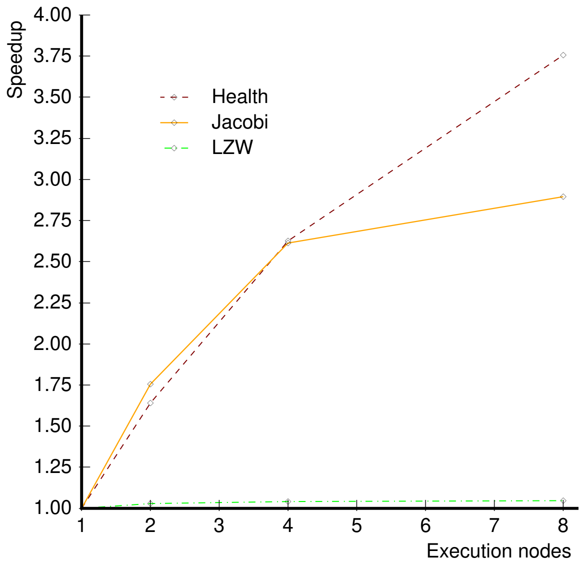


Figure 4: Ideal speedups as the number of compute nodes is varied from 1 to 8 if the memory dependence speculation system were able to deliver a result in a single cycle. On 8 compute nodes the speedup for Health is 3.756, the speedup for Jacobi is 2.895 and the speedup for LZW is 1.046.

Application	Rate of Rollback
Health	2.3%
Jacobi	17%
LZW	3.8%

Table 3: Rate at which parallel chunks must be rolled back due to mis-speculation on a system with 8 compute nodes and 16 memory nodes. These numbers include both memory dependence mis-speculations and branch mis-speculations.

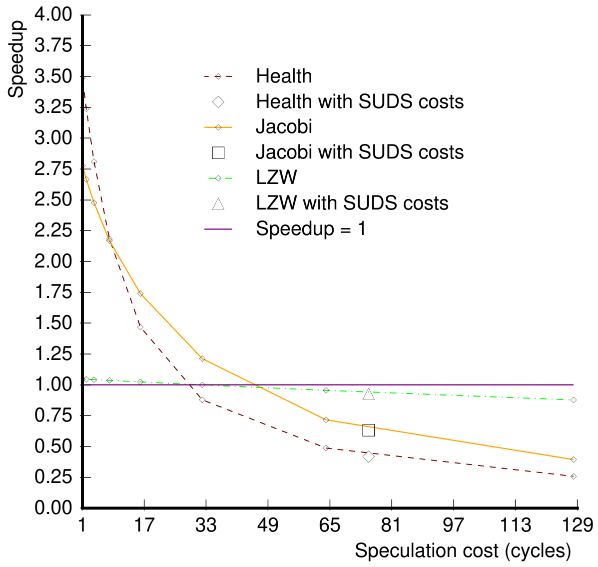


Figure 5: The speedups on a system with 8 compute nodes and 16 memory nodes as the cost of handling a speculative memory request in the memory nodes is varied from 1 cycle to 128 cycles. The 74 cycle cost for a memory operation in SUDS is marked on the graph. The crossover point for achieving speedups is 30 cycles for Health, 46 cycles for Jacobi and 32 cycles for LZW. It may be possible to reduce the average cost of memory in SUDS to below 30 cycles by using a combination of optimized protocols and a caching system at the compute nodes.

4.2 Application results

In this section we examine the question of how fast the average global memory operation needs to be in order to achieve parallel speedups. We find that in order to deliver speedups the memory system must have an average load latency of under 30 cycles. Although our system currently has a load latency of about 75 cycles we are encouraged that we may be able to achieve speedups with a reasonable amount of further engineering. As discussed above, between the hope of further optimizing the protocol at the memory nodes and the promise of reducing latency by putting a small cache at each of the compute nodes, we believe that an average latency of 30 cycles is achievable.

We have conducted a study on 3 applications, `health`, `jacobi` and `LZW`. `Health` is one of the pointer intensive benchmarks from the Olden Benchmark Suite [5]. It performs an event driven simulation of a health care system. Its main data structure is a tree of linked lists. `Jacobi` is a dense matrix implementation of the Jacobi relaxation algorithm. While this algorithm is easily parallelized by a compiler, in these experiments we use only the SUDS compiler, which currently does not perform any array dependence analysis. Our final benchmark is `LZW`, an implementation of the LZW decompression algorithm [38]. As well as performing random accesses into several arrays, this application is additionally constrained by having a large number of true dependences on its critical path. For these experiments we ran each application with a relatively small data set. We ran the `health` benchmark with 341 hospitals, the `jacobi` benchmark on a 50x50 matrix and the `LZW` benchmark with a 750 byte compressed english text.

To collect some basic performance numbers we developed a simple direct execution simulator. We use a compiler based tool, similar to `pixie`, to insert instruction counting code into each application program, and then we link together the application code and a complete implementation of the runtime system which has been ported to run on a set of UNIX processes which communicate via UNIX pipes. The communication code in the runtime system is hand annotated to update the instruction counters with latencies similar to those that might be found on a Raw processor. One of the nice features of a simulation system like this is that we can very easily change the costs of various portions of the runtime system, simply by changing the amounts that the profiler charges for each runtime operation. Another nice feature of this system is that it has allowed us to use the full suite of UNIX programming tools to debug and tune the protocols in the runtime system.

First we used our simulator to produce "ideal" speedup curves. These are shown in Figure 4. Here we have set the cost for each request to the memory dependence speculation system to 1 cycle. In this way we are able to measure the maximum speedup that each of these applications can achieve, given an unrealistically fast memory system. The graph shows that when the programs are run on a system with 8 compute nodes the speedup for `health` is 3.756 over sequential execution. The speedup for `jacobi` is 2.895 and the maximum achievable speedup for `LZW` is 1.046. The speedup for the `LZW` application is limited by Amdahl's law because the parallel loop accounts for only about a third of the total sequential program execution time.

Table 3 shows that for `health` and `LZW` the system has very few mis-speculations per parallel chunk. The `jacobi` application exhibits a relatively large number of mis-speculations. These mis-speculations are all due to branch mis-speculations because the 50x50 input matrix doesn't fit evenly onto 8 compute nodes. The result is that the first rows of the matrix are processed in 6 parallel chunks. When processing the final chunk six of the eight compute nodes in the final chunk register a branch mis-speculation because they are attempting to execute beyond the loop limit. An input matrix that mapped evenly onto 8 nodes would not exhibit this problem.

Figure 5 shows how the parallel speedup is affected by the cost of each load operation. For this experiment we held the number of compute nodes constant at 8, and the number of memory nodes constant at 16. Then we varied the simulated latency of each request to the memory nodes between 1 cycle (the ideal case) and 128 cycles. The graph shows how the speedup for each application degrades as the memory latency increases. For reference the graph also shows the line (speedup = 1) at which each application crosses over from getting parallel speedup to slowdown. The crossover point for `health` occurs at 30 cycles per memory request. The `LZW` benchmark crosses over at 32 cycles per memory request and `jacobi` achieves parallel speedup up to 46 cycles per memory request. The graph also indicates the slowdowns that are achieved by the current implementation of SUDS with an average memory latency of 74 cycles.

5 Related work

Timestamp based algorithms have long been used for concurrency control in transaction processing systems. The memory dependence validation algorithm used in SUDS is most similar to the "basic timestamp ordering" technique proposed by Bernstein and Goodman [4]. More sophisticated multiversion timestamp ordering techniques [30] reduce the number of false dependencies detected by the system at the cost of a more complex implementation. Optimistic concurrency control techniques [21], like SUDS, take the converse approach, optimizing for the case that operations do *not* conflict.

Memory dependence speculation is even more similar to virtual time systems, such as the Time Warp mechanism [16] used extensively for distributed event driven simulation. This technique is very much like multiversion timestamp ordering, but in virtual time systems, as in SUDS, the assignment of timestamps to tasks is dictated by the sequential program order. In a transaction processing system, each transaction can be assigned a timestamp whenever it enters the system. More specifically after detecting a conflict a transaction processing system restarts whichever transaction detects the conflict, giving it a higher transaction number in the process. SUDS must restart the thread with the later transaction number.

Knight's Liquid system [19, 18] used a method more like optimistic concurrency control [21] except that timestamps must be pessimistically assigned *a priori*, rather than optimistically when the task commits, and writes are pessimistically buffered in private memories and then written out in serial order so that different processing elements may concurrently write to the same address. The idea of using hash tables rather than full maps to perform independence validation was originally proposed for the Liquid system.

Knight also pointed out the similarity between cache coherence schemes and coherence control in transaction processing. The Liquid system used a bus based protocol similar to a snooping cache coherence protocol [12]. SUDS uses a scalable protocol that is more similar to a directory based cache coherence protocol [6, 2, 1] with only a single pointer per entry, sometimes referred to as a Dir1B protocol.

The ParaTran system for parallelizing mostly functional code [33] was another early proposal that relied on speculation. ParaTran was implemented in software on a shared memory multiprocessor. The protocols were based on those used in Time Warp [16], with checkpointing performed at every speculative operation.

SUDS is most directly influenced by the Multiscalar architecture [9, 31]. The Multiscalar architecture was the first to include a low-latency mechanism for explicitly forwarding dependencies from one task to the next. This allows the compiler to both avoid the expense of completely serializing do-across loops and also permits register allocation across task boundaries. The Multiscalar validates memory dependence speculations using a mechanism called an address resolution buffer (ARB) [9, 10] which is similar to a hardware implementation of multiversion timestamp ordering. From the perspective of a cache coherence mechanism the ARB is most similar to a full-map directory based protocol.

The SUDS compiler algorithms for identifying the optimal placement points for sending and receiving loop-carried dependences are similar to those used in the multiscalar [36]. The primary difference is that the Multiscalar algorithms permit some data values to be forwarded more than once, leaving to the hardware the responsibility for squashing redundant sends. The SUDS algorithms guarantee to insert send and receive instructions at the optimal point in the control flow graph such that each value is sent and received exactly once.

More recent efforts have focused on modifying shared memory cache coherence schemes to support memory dependence speculation [8, 13, 32, 20, 17, 14]. SUDS implements its protocols in software rather than relying on hardware mechanisms. In the future SUDS might permit long-term caching of read-mostly values by allowing the software system to "permanently" mark an address in the timestamp cache.

Another recent trend has been to examine the prediction mechanism used by dependence speculation systems. Some early systems [19, 33, 14] transmit all dependencies through the speculative memory system. SUDS, like the Multiscalar allows the compiler to statically identify loop carried dependences which are then forwarded using a separate, fast, communication path. SUDS and other systems in this class essentially statically predict that all memory references that the compiler can not analyze are in fact *independent*. Several recent systems [25, 35, 7] have proposed hardware mechanisms, based on runtime branch prediction, for finding, and explicitly forwarding, additional dependences that the compiler can not analyze.

Memory dependence speculation has also been examined in the context of fine-grain instruction level parallel processing on VLIW processors. The point of these systems is to allow trace-scheduling compilers more flexibility to statically reorder memory instructions. Nicolau [27] proposed inserting explicit address comparisons followed by branches to off-trace fixup code. Huang *et al* [15] extended this idea to use predicated instructions to help parallelize the comparison code. The problem with this approach is that it requires $m \times n$ comparisons if there are m loads being speculatively moved above n stores. This problem can be alleviated using a small hardware set-associative table, called a memory conflict buffer (MCB), that holds recently speculated load addresses and provides single cycle checks on each subsequent store instruction [11]. An MCB has been proposed for inclusion in the Hewlett Packard/Intel IA-64 EPIC architecture [3].

The LRPD test [29] is a software speculation system that takes a more coarse grained approach than SUDS. In contrast to most of the systems described in this section, the LRPD test speculatively block parallelizes a loop as if it were completely data parallel and then tests to ensure that the memory accesses of the different processing elements do not overlap. It is able to identify privatizable arrays and reductions at runtime. A directory based cache coherence protocol extended to perform the LRPD test is described in [39]. SUDS takes a finer grain approach which can cyclically parallelize loops with loop carried dependences and can parallelize most of a loop that has only a few dynamic dependences.

6 Conclusion

In this paper we have investigated the issues of whether it is possible to build a software based speculation system for a Raw processor. Our preliminary results are encouraging. Through simulation we've found that the memory request latencies for our working prototype system are within a factor of 2X the latencies that are required to achieve parallel speedup. We were able to reduce the software system's latencies to this level by dividing the dependence handling problem into three independent subproblems. These include forwarding true dependences, renaming false dependences and providing memory dependence speculation for potential dependences. Moving the problem of renaming into the compiler enabled us to simplify the implementation of the memory dependence speculation subsystem.

We believe that there are caching techniques that can be used to improve the average memory latencies in our system, and we are actively investigating how to integrate caches into our system. We believe that with this additional engineering effort it will be possible to build an all software speculation system which achieves parallel speedups.

References

[1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *15th International Symposium on Computer Architecture*, pages 280–289, Honolulu, HI, May 1988.

[2] James Archibald and Jean-Loup Baer. An Economical Solution to the Cache Coherence Problem. In *11th International Symposium on Computer Architecture*, pages 355–362, Ann Arbor, MI, June 1984.

[3] David I. August, Daniel A. Connors, Scott A. Mahlke, John W. Sias, Kevin M. Crozier, Ben-Chung Cheng, Patrick R. Eaton, Qudus B.

Olaniran, and Wen-Mei W. Hwu. Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture. In *25th International Symposium on Computer Architecture (ISCA-25)*, pages 227–237, Barcelona, Spain, June 1998.

[4] Philip A. Bernstein and Nathan Goodman. Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems. In *Proceedings of the Sixth International Conference on Very Large Data Bases*, pages 285–300, Montreal, Canada, October 1980.

[5] Martin C. Carlisle and Anne Rogers. Software Caching and Computation Migration in Olden. In *Proceedings of the Fifth ACM Symposium on Principles and Practice of Parallel Programming*, pages 29–38, Santa Barbara, CA, July 1995.

[6] Lucien M. Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[7] George Z. Chrysos and Joel S. Emer. Memory Dependence Prediction using Store Sets. In *25th International Symposium on Computer Architecture (ISCA-25)*, pages 142–153, Barcelona, Spain, June 1998.

[8] Manoj Franklin. Multi-Version Caches for Multiscalar Processors. In *Proceedings of the First International Conference on High Performance Computing (HiPC)*, 1995.

[9] Manoj Franklin and Gurindar S. Sohi. The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism. In *19th International Symposium on Computer Architecture (ISCA-19)*, pages 58–67, Gold Coast, Australia, May 1992.

[10] Manoj Franklin and Gurindar S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[11] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wen mei W. Hwu. Dynamic Memory Disambiguation Using the Memory Conflict Buffer. In *Proceedings of the 6th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, pages 183–193, San Jose, California, October 1994.

[12] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *10th International Symposium on Computer Architecture*, pages 124–131, Stockholm, Sweden, June 1983.

[13] Sridhar Gopal, T. N. Vijaykumar, James E. Smith, and Gurindar S. Sohi. Speculative Versioning Cache. In *Proceedings of the Fourth International Symposium on High Performance Computer Architecture (HPCA-4)*, pages 195–205, Las Vegas, NV, February 1998.

[14] Lance Hammond, Mark Willey, and Kunle Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, San Jose, CA, October 1998.

[15] Andrew S. Huang, Gert Slavenburg, and John Paul Shen. Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA)*, pages 200–210, Chicago, Illinois, April 1994.

[16] David R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[17] Iffat H. Kazi and David J. Lilja. Coarse-Grained Speculative Execution in Shared-Memory Multiprocessors. In *International Conference on Supercomputing (ICS)*, pages 93–100, Melbourne, Australia, July 1998.

[18] Thomas F. Knight, Jr. System and Method for Parallel Processing with Mostly Functional Languages. U.S. Patent 4,825,360, issued Apr. 25, 1989 (expired).

[19] Tom Knight. An Architecture for Mostly Functional Languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 88–93, August 1986.

[20] Venkata Krishnan and Josep Torrellas. Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor. In *International Conference on Supercomputing (ICS)*, Melbourne, Australia, July 1998.

[21] H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[22] Walter Lee, Rajeev Barua, , Matthew Frank, Devabhatuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, San Jose, CA, October 1998.

[23] Z. Li. Array Privatization for Parallel Execution of Loops. In *Conference Proceedings, 1992 International Conference on Supercomputing*, DC, July 1992.

[24] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array Data-Flow Analysis and its Use in Array Privatization. In *Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 2–15, Charleston, SC, January 1993.

[25] Andreas Moshovos and Gurindar S. Sohi. Streamlining Inter-operation Memory Communication via Data Dependence Prediction. In *30th Annual International Symposium on Microarchitecture (MICRO)*, Research Triangle Park, NC, December 1997.

[26] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Mateo, 1997.

[27] Alexandru Nicolau. Run-Time Disambiguation: Coping with Statically Unpredictable Dependencies. *IEEE Transactions on Computers*, 38(5):663–678, May 1989.

[28] Karen Pieper. *Parallelizing Compilers: Implementation and Effectiveness*. PhD thesis, Dept. of Computer Science, Stanford University, June 1993.

[29] Lawrence Rauchwerger and David Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 218–232, La Jolla, CA, June 1995.

[30] David P. Reed. Implementing Atomic Actions on Decentralized Data. *ACM Transactions on Computer Systems*, 1(1):3–23, February 1983.

[31] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *22nd International Symposium on Computer Architecture*, pages 414–425, Santa Margherita Ligure, Italy, June 1995.

[32] J. Gregory Steffan and Todd C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, pages 2–13, Las Vegas, NV, February 1998.

[33] Pete Tinker and Morry Katz. Parallel Execution of Sequential Scheme with ParaTran. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 40–51, July 1988.

[34] Peng Tu and David Padua. Automatic Array Privatization. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[35] Gary S. Tyson and Todd M. Austin. Improving the Accuracy and Performance of Memory Communication Through Renaming. In *30th Annual International Symposium on Microarchitecture (MICRO)*, Research Triangle Park, NC, December 1997.

[36] T. N. Vijaykumar. *Compiling for the Multiscalar Architecture*. PhD thesis, University of Wisconsin-Madison Computer Sciences Department, January 1998.

[37] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, September 1997.

[38] Terry Welch. A Technique for High-Performance Data Compression. *IEEE Computer*, 17(6):8–19, June 1984.

[39] Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas. Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors. In *Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, pages 162–173, Las Vegas, NV, February 1998.