A Software Framework for Supporting General Purpose Applications on Raw Computation Fabrics

Matthew Frank, Walter Lee, Saman Amarasinghe
MIT LCS
Cambridge, MA
{mfrank,walt,saman}@lcs.mit.edu

ABSTRACT

This paper presents SUDS (Software Un-Do System), a data speculation system for Raw processors. SUDS manages speculation in software. The key to managing speculation in software is to use the compiler to minimize the number of data items that need to be managed at runtime. Managing speculation in software enables Raw processors to achieve good performance on integer applications without sacrificing chip area for speculation hardware. This additional area can instead be devoted to additional compute resources, improving the performance of dense matrix and media applications.

1. INTRODUCTION

The rapid growth in the resources available on a chip, approaching a billion transistors within the next five years, is creating an exciting opportunity for computer architects to build a truly general-purpose microprocessor. Even when transistors are plentiful, finding the design point where a single architecture can efficiently support a diverse set of applications is a challenge.

From the perspective of parallelism, resources on a microprocessor can be divided into two types: basic computing resources such as functional units, registers, and memories; and parallelism-enabling resources that help an application utilize the compute resources concurrently. An important issue in designing a general-purpose microprocessor is how the transistor budget should be divided between these two types of resources.

This issue is complex because different applications have seemingly different requirements. The conflict is evident from the fact that historically, different applications prefer different architectures. Dense matrix applications and media applications prefer multiprocessors, vector processors, DSPs, and special-purpose processors. These processors provide ample computing resources but scanty parallelism-enabling resources. Integer applications, on the other hand, prefer superscalars, which contain modest computing resources but copious parallelism-enabling resources.

The Raw philosophy to building general-purpose microprocessors challenges the notion that one needs to build specialized hardware to manage various types of parallelism [54]. It dedicates all of the processor transistor budget to compute resources. Any specialized support for parallelization is implemented in software on top of excess compute resources. Through compiler optimization, the Raw philosophy aims to overcome this software overhead. Architectures that follow the Raw philosophy are called Raw architectures.

MIT Laboratory for Computer Science Technical Memo MIT-LCS-TM-619, July 20, 2001.

However, due to the lack of parallelism-enabling resources, a Raw architecture by itself does not provide any performance benefit for integer applications. In fact, only a very small portion of hardware resources, a single set of compute resources in a large fabric, can be used in executing an integer application. This paper introduces a software framework that enables the use of the entire fabric to increase performance of integer applications.

This paper applies the Raw philosophy to memory dependence speculation, a parallelization technique that speculatively executes memory dependences out of order to improve performance. The paper describes SUDS (Software Un-Do-System), a system for performing memory dependence speculation in loops on an Raw architecture. Briefly, SUDS works as follows. 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.

The cost of building a system in software is that, when a runtime subsystem *is* required, it consumes more time, power, energy and area than the equivalent functionality built from specialized hardware. On the other hand, we demonstrate in this paper that these costs can be contained by using compiler optimizations. In particular, for speculation, the key compiler optimizations are those that identify opportunities for renaming. *Data items that can be renamed between checkpoints don't require runtime management.*

The specific contributions of this paper include:

1. A framework to improve performance of integer applications on a Raw fabric without sacrificing the performance of scientific applications.
2. The design of an efficient all-software speculation system.
3. The description of a set of compile time optimizations that reduce the number of data elements requiring runtime management.

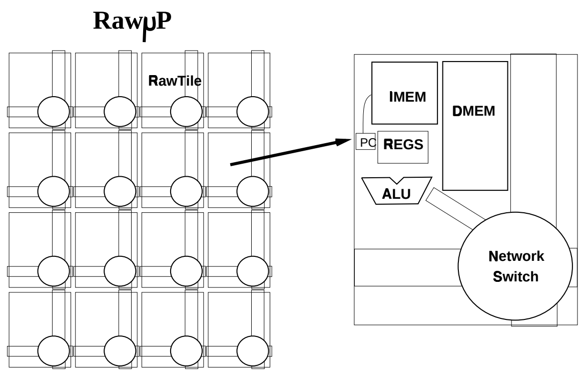

Figure 1: RawμP composition. The system is made up of multiple tiles. In addition to instruction and data memories, each tile contains a processor pipeline with a register-mapped network interface.

4. A case study that demonstrates the effectiveness of the SUDS system.

The rest of this paper is structured as follows. The next section gives an informal description of how SUDS works in the context of an example. Section 3 describes the design of the SUDS system. Section 4 presents a case study. Section 5 presents related work. Section 6 concludes.

2. BACKGROUND

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 (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 [34, 48, 54].

2.1 Chunk based work distribution

As shown in Figure 2, SUDS partitions Raw's tiles into two groups. Some portion of the tiles are designated as *compute* nodes. The rest are designated 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.

SUDS parallelizes loops by cyclically distributing the loop iterations across the compute nodes. 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.

In the current version of the 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 provided by SUDS work with any loop, even "do-across" loops, loops with true-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.

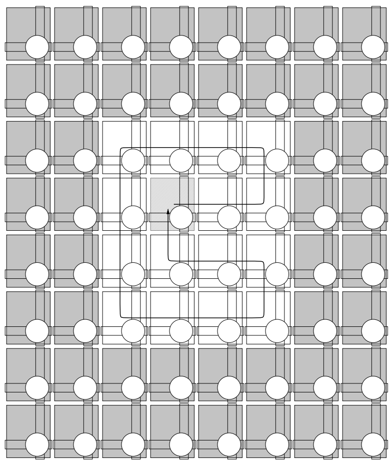

Figure 2: An example of how SUDS allocates resources on a 64 tile Raw machine. The gray tiles are memory nodes. The white tiles are worker nodes, the gray hatched tile near the center is the master node. Loop carried dependences are forwarded between compute nodes in the pattern shown with the arrow.

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

Figure 3: An example loop.

2.2 Example

Figure 3 shows an example of a simple loop with non-trivial dependences. Figure 4 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 5 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

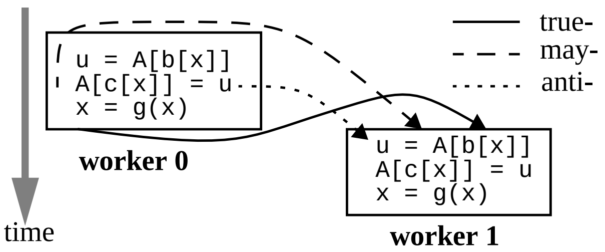

Figure 4: SUDS runs one iteration of the loop on each worker node. In this case the dependences between iterations limit the available parallelism.

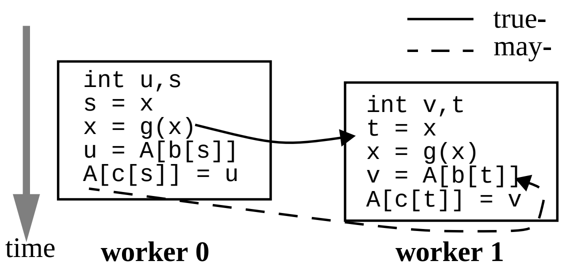

Figure 5: After renaming the anti-dependence is eliminated and the critical path length of the true-dependence is shortened.

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. 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 miss-speculation point. Because the system is speculating that the code contains no memory dependences, this technique is called *memory dependence speculation* [19].

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.

3. DESIGN

The previous section gave a basic overview of Raw processors and memory dependence speculation. This section describes the techniques used in the SUDS system. The challenge of a software based memory dependence speculation system is to make the runtime system efficient enough that its costs don't completely swamp the real work being done on behalf of the user's application.

The approach taken in the SUDS system is to move as much work as possible to compile time. In particular, SUDS takes the unique approach of using the compiler to identify opportunities for renaming. Since renaming no longer needs to be done at runtime, the runtime system is efficient enough to realize the desired application speedups. We next discuss the basic SUDS system and the optimizations that make the runtime system more efficient.

Object Category	Run Time Technique	Code Generation Technique
Private	Local Stack	Stack Splitting
Loop Carried	Checkpoint Repair	Communication Instruction Placement
Heap	Memory Dependence Speculation	Memory Abstraction

Figure 6: The SUDS system divides objects into three major categories. The system has a runtime system component and code generation component for each object category.

As discussed in Section 2.2, there are three types of dependences that are managed by SUDS. SUDS categorizes objects based on the structure of their dependences. Each object category is handled by a different runtime subsystem and a corresponding code generation technique. This breakdown is summarized in Figure 6.

Private variables are those that have a lifetime that is restricted to a single loop iteration. These are the major candidates for renaming and are handled most efficiently by SUDS. SUDS provides support for renaming by allocating a local stack on each worker, as well as a single global stack that can be accessed by any worker. At code generation time, the compiler allocates private objects to the tile registers and local stack using a technique called *stack splitting*. This technique separates objects on the stack between the local stack and global stack, depending on the compiler's ability to prove that the object has no aliases. This is similar to techniques that have been used to improve the performance of register spills on digital signal processors[12].

SUDS handles *loop carried dependent* objects at runtime by explicitly checkpointing them on the master node and then forwarding them from worker to worker through Raw's point-to-point interconnect, as shown in Figure 2. The code generator is responsible for placing the explicit communication instructions so as to minimize delays on the critical path while guaranteeing that each object is sent and received exactly once per iteration, no matter what arbitrary control flow might happen within the iterations.

The remaining objects, denoted *heap* objects, are those that the compiler is unable to analyze further at compiler time. They are handled at the memory nodes using a runtime memory dependence validation protocol that is based on Basic Timestamp Ordering [6]. This technique is described in more detail in the next section. The code generator converts all heap object load and store operations into instructions to communicate between the workers and memory nodes.

3.1 Managing heap objects

The memory dependence speculation system is in some ways the core of the system. It is the fallback dependence mechanism that works in all cases, even if the compiler cannot 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 substantial 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.

3.1.1 A conceptual view

The method we use to validate memory dependence correctness is based on Basic Timestamp Ordering [6], a traditional transaction

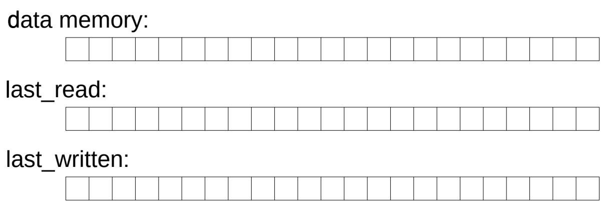

Figure 7: A conceptual view of Basic Timestamp Ordering. Associated with every memory location is a pair of timestamps that indicate the logical time at which the location was last read and written.

processing concurrency control mechanism. A conceptual view of the protocol is given in Figure 7. 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`). In addition, the memory is checkpointed at the beginning of each chunk so that modifications can be rolled back in the case of an abort.

The validation protocol works as follows. As each load request arrives, its timestamp (`read_time`) is compared to the `last_written` stamp for its memory location. If `read_time` $\geq$ `last_written` then the load is in-order and `last_read` is updated to `read_time`, otherwise the system flags a miss-speculation and aborts the current chunk.

On a store request, the 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 in-order, otherwise the system flags a miss-speculation and aborts the current chunk.

We have implemented an optimization on store requests that is known as the Thomas Write Rule [6]. 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 in-order and `last_written` is updated as `write_time`.

3.1.2 Implementation

We can't dedicate such a substantial amount of memory to the speculation system, so the system is actually implemented using a hash table. As shown in Figure 8, each processing element that is dedicated as a memory dependence node contains three data structures in its local memory. The first is an array that is dedicated to storing actual program values. The next is a small hash table that is used as a *timestamp cache* to validate the absence of memory conflicts. Finally, the *log* 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 beginning of the chunk.

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

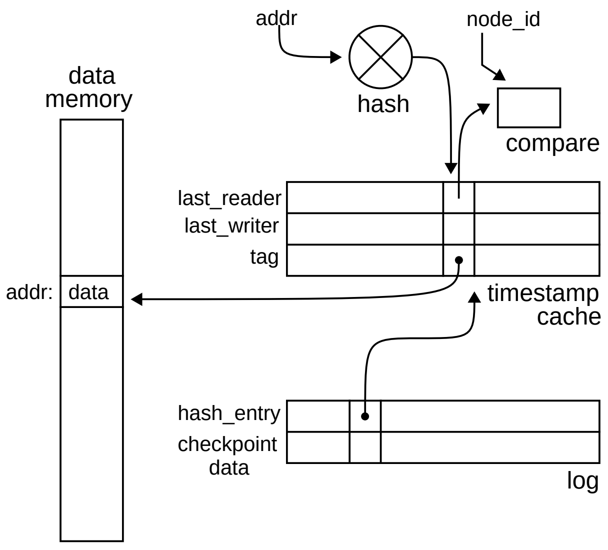

Figure 8: Data structures used by the memory dependence speculation subsystem.

Operation	Cost
Send from compute node	1
Network latency	4 + distance
Memory node	8
Network latency	4 + distance
Receive on compute node	2
Total	19 + 2 $\times$ distance

Figure 9: The round trip cost for a load operation is 19 cycles + 2 times the manhattan distance between the compute and memory node. The load operation also incurs additional occupancy of up to 40 cycles on the memory node after the data value is sent back to the compute node.

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 miss-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. Because we are storing the most current value in the memory itself, commits are cheaper, and we are able to implement a fast path for load operations. Before going through the validation process, a load request fetches the required data and returns it to the requester. The resulting latency at the memory node is only 8 cycles as shown in Figure 9. The validation process happens after the data has been returned, and occupies the memory node for an additional 14 to 40 cycles, depending on whether a log entry needs to be created.

In the common case the chunk completes without suffering a miss-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. This costs 5 cycles per memory location that was touched during the chunk.

If a miss-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. Rollback costs 11 cycles per memory location that was touched during the chunk.

3.1.3 Code generation

Since the computation and heap memory are handled on separate nodes in SUDS, every access to the heap involves communication through Raw's network. The code generation pass is responsible for identifying every load or store operation that needs to be sent from a compute node to a remote memory node. Each load operation to the heap is replaced by a pair of special instructions. The first is a message construction instruction. It calculates the correct destination memory node for a particular memory request, then constructs the corresponding message header and body, and launches the message. The second instruction is a register move instruction (with byte extract and/or sign extension if the system only wants an 8 or 16 bit result). This second instruction will stall until the corresponding data item is returned from the memory node. Heap stores are acknowledged asynchronously in SUDS, so each store instruction is replaced by a single message construction instruction, similar to that used for load instructions.

3.2 Handling 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. Any scalar variable modified within the loop nest needs to be either privatized by renaming (see Section 3.3) or forwarded to the next iteration. If the compiler finds that there is a true-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 [51] to identify the optimal placement of communication instructions. The compiler arranges communication instructions such that after the last modification of the variable it is sent to the next worker in the chunk, and before the first read of the variable it is received from the previous worker in the chunk.

At runtime the master node checkpoints all the compiler identified true-dependences so that if a miss-speculation occurs the computation can be rolled back to a consistent state. Since only the master node checkpoints, this cost can be amortized over a number of loop iterations. The drawback of this approach is that when a miss-speculation does occur we may need to rollback slightly further than necessary. So far we have not found this to be a problem. In the programs we have looked at, the rate of miss-speculations per chunk is low enough that it does not constrain parallelism.

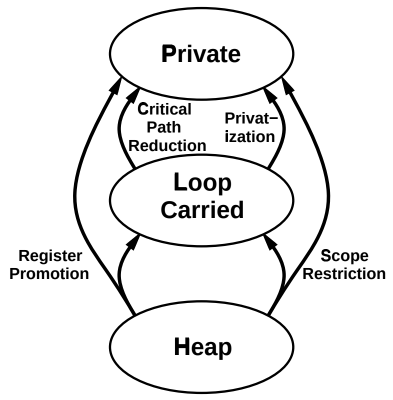

Figure 10: Compiler optimizations. Each of the optimizations (critical path reduction, privatization, register promotion and scope restriction) attempts to move objects from one category (heap, loop carried, or private), to a more efficient category.

3.3 Renaming

Renaming in SUDS is handled completely by the compiler. This simplifies the design of the memory dependence speculation system, because multiple versions of memory locations don't need to be maintained by the runtime system. The values that the compiler decides to privatize are kept in the registers and local memories of the compute nodes for the duration of a single iteration. In addition to the privatization optimizations described in Section 3.4, the code generator performs *stack splitting*.

Stack splitting simply identifies register allocatable scalars that never have their addresses taken. These values are kept on the normal C stack along with all register spills taken at procedure calls. For scalars that *do* have their addresses taken, the code generator creates and manages a second *global* stack as part of the heap. The values in the global stack are then managed at runtime as described in Section 3.1.

The advantage of stack splitting is that register spills don't need to be handled by the speculation system. Since we are parallelizing loops, the stack pointer is pointing at the same address on all the compute nodes. If we maintained only a single global stack, then every time the program took a procedure call, all the compute nodes would spill to the same memory locations, requiring the speculation system to manage multiple versions of the same memory location.

3.4 Compile time optimizations

Since SUDS has special runtime support that allows it to handle privates and true-dependences more efficiently at runtime, the goal of the SUDS optimizer is to move as many objects as possible into the more efficient categories. It does this using the four compiler optimizations identified in Figure 10.

Privatization is the central optimization technique. It depends on a dataflow analysis to identify objects whose live ranges do not extend outside the body of a loop. The SUDS privatization phase also identifies true-dependences and loop-invariants. It differentiates all of these from heap based objects that can not be handled as efficiently at runtime.

Critical path reduction is a technique for improving program parallelism in the face of loop carried dependences. An example of this optimization was shown in Figure 5. The idea is to introduce additional private variables that will hold the old value of the object while the new value is computed and forwarded to the other, waiting, workers.

Register promotion is similar to performing common subexpression or partial redundancy elimination on load and store instructions [13, 35, 8]. This reduces the number of requests that need to be sent from the worker nodes and processed by the memory nodes.

The final optimization is *scope restriction*, which allows privatization of structure objects that can not be fully analyzed by the privatization pass. Scope restriction takes advantage of scoping information provided by the programmer. It allows structure objects declared inside the body of the loop to be promoted to privates and handled efficiently by the runtime system.

4. CASE STUDY

In this section we demonstrate that moving speculation into software avoids the traditional area tradeoffs faced by computer architects. When speculation is not required the system can devote all of the chip area to useful computation. When speculation is required, the software system can be turned on and achieve IPC numbers similar to those achieved by a hardware based speculation system of similar area.

SUDS is designed to run on Raw microprocessors. As reported elsewhere [48], each Raw chip contains a 4 by 4 array of tiles; multiple chips can be composed to create systems as large as 32 by 32 tiles. Raw is currently (July 2001) running in RTL emulation at about 1 MHz on a 5 million gate IKOS VirtuaLogic emulator [4], and it will tape out at the end of the summer. It is implemented in IBM's .15 micron SA-27E ASIC process with a target frequency of 250 MHz.

The results in this paper were run on a (nearly) cycle accurate simulation of a Raw system with a few minor tweaks. In particular, the simulator provides access to a particular message header construction instruction that is not available in the actual implementation, and the simulator does not model network contention. The first tweak saves us several cycles during each remote memory operation, while the second is of little consequence since total message traffic in our system is sufficiently low.

Programs running with the SUDS system are parallelized by a SUIF based compiler that outputs SPMD style C code. The resulting code is compiled for the individual Raw tiles using gcc version 2.8.1 with the `-O3` flag. (Raw assembly code is similar to MIPS assembly code, so our version of the gcc code generator is a modified version of the standard gcc MIPS code generator).

Moldyn

Moldyn is a molecular dynamics simulation, originally written by Shamik Sharma [45], that is difficult to parallelize without speculation support. Rather than calculate all $O(N^2)$ pairwise force calculations every iteration, Moldyn only performs force calculations between particles that are within some cutoff distance of one another (Figure 11). The result is that only $O(N)$ force calculations need to be performed every iteration.

```
ComputeForces(vector<particle> molecules,
              real cutoffRadius) {
  foreach m in molecules {
    foreach m' in m.neighbors() {
      if (distance(m, m') <
          cutoffRadius) {
        force_t force = calc_force(m, m');
        m.force += force;
        m'.force -= force;
      }
    }
  }
}
```

Figure 11: Pseudocode for `ComputeForces`, the Moldyn routine for computing intermolecular forces. The neighbor sets are calculated every 20th iteration by calling the `BuildNeigh` routine (Figure 12).

The original version of Moldyn recalculated all $O(N^2)$ intermolecular distances every 20 iterations. For this paper, we rewrote the distance calculation routine so that it would also run in $O(N)$ time. This is accomplished by chopping the space up into boxes that are slightly larger than the cutoff distance, and only calculating distances between particles in adjacent boxes (Figure 12). This improved the speed of the application on a standard processor by several orders of magnitude.

Under SUDS we can parallelize each of the outer loops (those labeled "`foreach m in molecules`" in Figures 11 and 12). Each loop has different characteristics when run in parallel.

The first loop in the `BuildNeigh` routine moves through the array of molecules quickly. For each molecule it simply calculates which box the molecule belongs in, and then updates one element of the (relatively small) `boxes` array. This loop does not parallelize well because updates to the `boxes` array have a relatively high probability of conflicting when run in parallel.

The second loop in the `BuildNeigh` routine is the most expensive single loop in the program (although, luckily it only needs to be run about one twentieth as often as the `ComputeForces` loop). It is actually embarrassingly parallel, although potential pointer aliasing makes it difficult for a traditional parallelizing compiler to analyze this loop. SUDS, on the other hand, handles the pointer problem by speculatively sending the pointer references to the memory nodes for resolution. Since none of the pointer references actually conflict, the system never needs to roll back, and this loop achieves scalable speedups.

The `ComputeForces` routine consumes the majority of the runtime in the program, since it is run about twenty times more often than the `BuildNeigh` routine. For large problem sizes, the `molecules` array will be very large, while the number of updates per molecule stays constant, so the probability of two parallel iterations of the loop updating the same element of the `molecules` array is small. Unfortunately, while this loop parallelizes well up to about a dozen compute nodes, speedup falls off for larger numbers

```
BuildNeigh(vector<list<int>> adjLists,
           vector<particle> molecules,
           real cutoffRadius) {
  vector<list<particle>> boxes;

  foreach m in molecules {
    int mBox = box_of(m.position());
    boxes[mBox].push_back(m);
  }

  foreach m in molecules {
    int mBox = box_of(m.position());
    foreach box in adjLists[mBox] {
      foreach m′ in box {
        if (distance(m, m′) <
            (cutoffRadius * TOLERANCE)) {
          m.neighbors().push_back(m′);
        }
      }
    }
  }
}
```

Figure 12: Pseudocode for `BuildNeigh`, the Moldyn routine for recalculating the set of interacting particles. `adjLists` is a pre-calculated list of the boxes adjacent to each box.

MIPS R4000	0.40
SUDS	0.96
"perfect" superscalar	1.16

Figure 13: Comparison of IPC for Moldyn running on three different architectures.

of compute nodes because of the birthday paradox. This is the argument that one needs only 23 people in a room to have a probability of 50% that two of them will have the same birthday. Likewise, as we increase the number of iterations that we are computing in parallel, the probability that two of them update the same memory location increases worse than linearly. This is a fundamental limitation of data speculation systems, not one unique to the SUDS system.

Figure 13 shows the IPC of running Moldyn with an input dataset of 256000 particles on three different architectures. The first is a MIPS R4000 with a 4-way associative 64KByte combined I&D L1, 256MByte L2 with 12 cycle latency and 50 cycle miss cost. It achieves about .4 IPC. The second is SUDS running on a 40 tile Raw system. 8 tiles are dedicated as compute nodes and an additional 32 are dedicated as memory nodes. Each simulated Raw tile contains a pipeline similar to an R4000, and a 64KByte L1 cache. Cache misses to DRAM cost 50 cycles. SUDS is able to achieve .96 IPC.

The final architecture is a simulated superscalar architecture with a 32 Kbit gshare branch predictor, a perfect 8-way instruction fetch unit, a 64 Kbyte 4-way set associative combined I&D L1, and 256 MByte L2 with 12 cycle latency and 50 cycle miss cost. It has infinite functional units, infinite registers for renaming, a memory stunt-box to allow loads to issue to the cache out of order, and an infinite number of ports on all memories. Even though such an architecture is not feasible, we include it to show that SUDS performance is quite reasonable for this particular application. SUDS achieves 82% of the IPC achieved by this superscalar.

5. RELATED WORK

The main motivation for SUDS comes from previous work in *micro-optimization.* Micro-optimization has two components. The first, *interface decomposition* involves breaking up a monolithic interface into constituent primitives. Examples of this include Active Messages as a primitive for building more complex message passing protocols [52], and interfaces that allow user level programs to build their own customized shared memory cache coherence protocols [10, 33, 42]. Examples of the benefits of carefully chosen primitive interfaces are also common in operating systems research for purposes as diverse as communication protocols for distributed file systems [43], virtual memory management [24], and other kernel services [7, 16, 27].

The second component of micro-optimization involves using automatic compiler optimizations (*e.g.*, partial redundancy elimination) to leverage the decomposed interface, rather than forcing the application programmer to do the work. This technique has been used to improve the efficiency of floating-point operations [14], fault isolation [53], and shared memory coherence checks [44]. On Raw, micro-optimization across decomposed interfaces has been used to improve the efficiency of both branching and message demultiplexing [34], memory access serialization [5, 15], instruction caching [36], and data caching [37]. SUDS micro-optimizes by breaking the monolithic memory interface into separate primitives for accessing local and remote memory. The compiler then eliminates work by finding opportunities for renaming.

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 [6]. More sophisticated multiversion timestamp ordering techniques [41] provide some memory renaming, reducing the number of false dependences detected by the system at the cost of a more complex implementation. Optimistic concurrency control techniques [32], in contrast, attempt to reduce the cost of validation, by performing the validations in bulk at the end of each transaction.

Memory dependence speculation is even more similar to virtual time systems, such as the Time Warp mechanism [26] used extensively for distributed event driven simulation. This technique is very much like multiversion timestamp ordering, but in virtual time systems, as in data speculation systems, 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.

Knight's Liquid system [29, 30] used a method more like optimistic concurrency control [32] 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 [21]. SUDS uses a scalable protocol that is more similar to a directory based cache coherence protocol [9, 2, 1] with only a single pointer per entry, sometimes referred to as a Dir1B protocol.

The ParaTran system for parallelizing mostly functional code [49] 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 [26], with checkpointing performed at every speculative operation. A similar system, applied to an imperative, C like, language (but lacking pointers) was developed by Wen and Yelick [55]. While their compiler could identify some opportunities for privatizing temporary scalars, their memory dependence speculation system was still forced to do renaming and forward true-dependences at runtime, and was thus less efficient than SUDS.

SUDS is most directly influenced by the Multiscalar architecture [18, 46]. The Multiscalar architecture was the first to include a low-latency mechanism for explicitly forwarding dependences 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) [18, 19] that 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 true-dependences are similar to those used in the Multiscalar [51]. 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 compiler algorithm is guaranteed 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 [17, 22, 47, 31, 28, 23]. 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 [29, 49, 23] transmit all dependences through the speculative memory system. SUDS, like the Multiscalar, allows the compiler to statically identify true-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 [38, 50, 11] have proposed hardware prediction mechanisms, 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 [39] proposed inserting explicit address comparisons followed by branches to off-trace fixup code. Huang *et al* [25] 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 [20]. An MCB is included in the Hewlett Packard/Intel IA-64 EPIC architecture [3].

The LRPD test [40] 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 [56]. SUDS takes a finer grain approach that can cyclically parallelize loops with true-dependences and can parallelize most of a loop that has only a few dynamic dependences.

6. CONCLUSION

This paper presents SUDS, a software speculation system for Raw microprocessors that can effectively execute integer programs with complex control-flow and sophisticated pointer aliasing. SUDS can efficiently execute programs where compile-time parallelism extraction is difficult or even impossible, on an architecture with no hardware support for parallelism extraction. This ability of SUDS makes Raw architectures viable as a general purpose architecture, capable of supporting a very large class of applications. Furthermore, since SUDS does not require any additional hardware, applications that are compiler parallelizable do not have to sacrifice their performance in order to accommodate the integer programs.

SUDS uses compiler analysis to reduce the amount of work that needs to be performed at runtime. Unlike coarse-grained parallelizing compilers that either completely succeed or completely fail, application performance under SUDS degrades gracefully when the compiler analysis is only partially applicable.

SUDS relies on a sophisticated memory system to support memory dependence speculation. Since the system is implemented entirely in software, it can be extended or customized to suit individual applications. For example, the memory nodes could be augmented with read-modify-write requests for a set of simple operations such as add, subtract, and xor.

Using the Moldyn application as an example, we show that SUDS is capable of speculatively parallelizing applications with dependences not analyzable at compile-time. Currently we do not have all the compiler analyses implemented; thus minor hand modifications to programs are required. We are in the process of implementing the register promotion and scope restriction compiler passes. We hope to show results on more applications within the next few months, in time for the final presentation.

As a fine-grained, software-based speculation system, SUDS provides us with an opportunity to realize other concepts previously deemed impractical. For example, SUDS could be the basis of reverse execution in debugging. Error detection can be incorporated with the SUDS checkpoint and roll back mechanism. This feature could improve the reliability of applications in the face of hardware errors. We also plan to investigate other speculative program optimizations that use SUDS to undo speculation failures.

Acknowledgements

This work was supported in part by DARPA under contract #DABT63-96-C-0036.

7. REFERENCES

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

[2] J. Archibald and J.-L. 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] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W.-M. 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] J. Babb, R. Tessier, M. Dahl, S. Z. Hanono, D. M. Hoki, and A. Agarwal. Logic emulation with virtual wires. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(6):609–626, June 1997.

[5] R. Barua, W. Lee, S. P. Amarasinghe, and A. Agarwal. Maps: A Compiler-Managed Memory System for Raw Machines. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 4–15, Atlanta, GA, May 2–4 1999.

[6] P. A. Bernstein and N. 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, Oct. 1980.

[7] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. J. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain Resort, CO, Dec. 3-6 1995.

[8] R. Bodík, R. Gupta, and M. L. Soffa. Load-Reuse Analysis: Design and Evaluation. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 64–76, Atlanta, GA, May 1999.

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

[10] D. Chaiken and A. Agarwal. Software-Extended Coherent Shared Memory: Performance and Cost. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 314–324, Chicago, Illinois, April 18–21, 1994.

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

[12] K. D. Cooper and T. J. Harvey. Compiler-Controlled Memory. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, San Jose, CA, Oct. 3–7 1998.

[13] K. D. Cooper and J. Lu. Register Promotion in C Programs. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 308–319, Las Vegas, NV, June 1997.

[14] W. J. Dally. Micro-Optimization of Floating-Point Operations. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 283–289, Boston, MA, April 3–6, 1989.

[15] J. R. Ellis. *Bulldog: A Compiler for VLIW Architecture*. PhD thesis, Department of Computer Science, Yale University, Feb. 1985. Technical Report YALEU/DCS/RR-364.

[16] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, CO, Dec. 3-6 1995.

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

[18] M. Franklin and G. 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.

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

[20] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. 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, Oct. 1994.

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

[22] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. 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, Feb. 1998.

[23] L. Hammond, M. Willey, and K. 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, Oct. 1998.

[24] K. Harty and D. R. Cheriton. Application-Controlled Physical Memory using External Page-Cache Management. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 187–197, Boston, MA, October 12–15, 1992.

[25] A. S. Huang, G. Slavenburg, and J. P. 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, Apr. 1994.

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

[27] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Janotti, and K. Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 52–65, Saint-Malo, France, Oct. 5-8 1997.

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

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

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

[31] V. Krishnan and J. 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.

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

[33] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, Chicago, Illinois, April 18–21, 1994.

[34] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. 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, Oct. 1998.

[35] R. Lo, F. C. Chow, R. Kennedy, S.-M. Liu, and P. Tu. Register Promotion by Partial Redundancy Elimination of Loads and Stores. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 26–37, Montreal, Quebec, June 1998.

[36] J. E. Miller. Software Based Instruction Caching for the RAW Architecture. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1999.

[37] C. A. Moritz, M. Frank, and S. Amarasinghe. Flexcache: A framework for flexible compiler generated data caching. In *Proceedings of the 2nd Workshop on Intelligent Memory Systems*, Boston, MA, Nov. 12 2000. to appear Springer LNCS.

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

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

[40] L. Rauchwerger and D. 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.

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

[42] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, Chicago, Illinois, April 18–21, 1994.

[43] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov. 1984.

[44] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Cambridge, MA, October 1–5, 1996.

[45] S. D. Sharma, R. Ponnusamy, B. Moon, Y. shin Hwang, R. Das, and J. Saltz. Run-time and compile-time support for adaptive irregular problems. In *Proceedings of Supercomputing*, pages 97–106, Washington, DC, Nov. 1994.

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

[47] J. G. Steffan and T. 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, Feb. 1998.

[48] M. Taylor, J. Kim, J. Miller, F. Ghodrat, B. Greenwald, P. Johnson, W. Lee, A. Ma, N. Shnidman, D. Wentzlaff, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw processor: A composeable 32-bit fabric for embedded and general purpose computing. In *Proceedings of HotChips 13*, Palo Alto, CA, Aug. 2001.

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

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

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

[52] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 19–21, 1992.

[53] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, North Carolina, Dec. 5-8 1993.

[54] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, Sept. 1997.

[55] C.-P. Wen and K. Yelick. Compiling sequential programs for speculative parallelism. In *Proceedings of the International Conference on Parallel and Distributed Systems*, Taiwan, Dec. 1993.

[56] Y. Zhang, L. Rauchwerger, and J. 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, Feb. 1998.