Spice: Speculative Parallel Iteration Chunk Execution

Easwaran Raman Neil Vachharajani Ram Rangan* David I. August

Department of Computer Science
Princeton University
Princeton, NJ 08540
{eraman,nvachhar,ram,august}@princeton.edu

ABSTRACT

The recent trend in the processor industry of packing multiple processor cores in a chip has increased the importance of automatic techniques for extracting thread level parallelism. A promising approach for extracting thread level parallelism in general purpose applications is to apply memory alias or value speculation to break dependences amongst threads and executes them concurrently.

In this work, we present a speculative parallelization technique called Speculative Parallel Iteration Chunk execution (Spice) which relies on a novel software-only value prediction mechanism. Our value prediction technique predicts the loop live-ins of only a few iterations of a given loop, enabling speculative threads to start from those iterations. It also increases the probability of successful speculation by only predicting that the values will be used as live-ins in **some** future iterations of the loop. These twin properties enable our value prediction scheme to have high prediction accuracies while exposing significant coarse-grained thread-level parallelism. Spice has been implemented as an automatic transformation in a research compiler. The technique results in up to 157% speedup (101% on average) with 4 threads.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation, Compilers, Optimization*; C.1.4 [**Processor Architectures**]: Parallel Architectures

General Terms

Algorithms, Performance

Keywords

Multicore architectures, automatic paralleization, speculative parallelization, thread level parallelism, value speculation

*Currently with the Performance and Tools Group, IBM Austin Research Laboratory, rrangan@us.ibm.com.

CGO'08, April 5–10, 2008, Boston, Massachusetts, USA.

1. INTRODUCTION

The emergence of multi-core architectures as the dominant computing platform is accompanied by a near stagnation in the performance of individual processor cores. Thus, single threaded applications can no longer rely on advances in process technology and microarchitectural innovations alone to improve their performance. On the other hand, writing multi-threaded applications is much more complex than writing single threaded applications since it requires programmers to reason about concurrent accesses to shared data, and to insert the right amount of synchronization that ensures correct behavior without limiting parallelism. Active research in automatic tools to identify deadlocks, livelocks, and race conditions [5, 7, 19] in multi-threaded programs is a testament to the difficulty of this task.

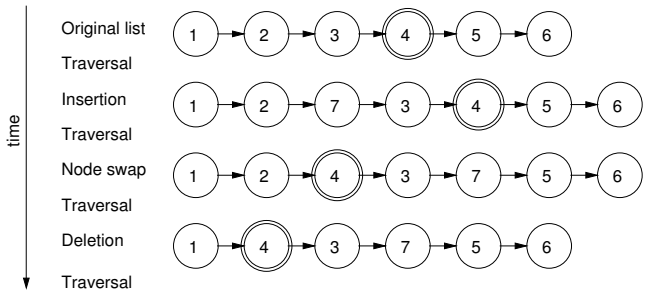
An alternative approach for producing multi-threaded codes is to use the compiler and the hardware to automatically convert single-threaded applications into multi-threaded applications. This approach is attractive as it takes the burden of writing multi-threaded code off the programmer, just as automatic instruction-level parallelism (ILP) optimizations take the burden of writing code optimized for complex architectures off the programmer. While significant progress has been made in parallelizing applications in the scientific and numeric computing domain, the same techniques do not seem to apply to general-purpose programs due to the complex control flow and memory access patterns in these programs.

A promising technique to parallelize general purpose applications is thread level speculation (TLS) [8, 13, 20, 21]. TLS *speculates* that a future iteration of a loop, typically the next iteration, is independent of the current iteration, and executes them in parallel. The most common form of speculation used in TLS is memory alias speculation where loads in later iterations are assumed not to conflict with stores in the earlier iterations. If the speculation turns out to be false, due to dependences between the iterations, the speculatively executed iterations are squashed and restarted. As long as the mis-speculation rate is low, TLS can improve the performance over single-threaded execution.

The alias speculation works as long as the conflict between loads and stores in different iterations are infrequent. But if the dependences between loop iterations manifest frequently, alias speculation suffers from high mis-speculation rates, causing a slowdown over single-threaded code. TLS systems typically overcome this problem by *synchronizing* those store-load pairs that conflict frequently. This limits the parallelism that can be exploited between the loop iterations. An alternative approach is to apply *value prediction* [10, 11] and speculatively execute the future iterations with the predicted values. Several TLS techniques [4, 12, 14, 22] have in fact proposed the use of value prediction to increase the amount of parallelism exploited. These TLS techniques with value pre-

```
1  c = cm->next_cl;
2  while(c  != NULL){
3    int w = c->pick_weight;
4    if (w < wm) {
5      wm = w;
6      cm = c;
7    }
8    c = c->next_cl;
9  }
```

(a) Traversal code



(b) Modifications to linked list

Figure 1: Value prediction example

diction typically use some value predictor originally proposed to improve ILP. While those predictors work in some cases, we believe that better value prediction techniques tailor-made to break inter-iteration dependences in a TLS system can be developed. We propose a new parallelization technique that uses a predictor based on the following two insights into value prediction for thread level speculation.

The first insight is that, *to extract thread level parallelism, it is sufficient to predict a small subset of the values produced by a static operation.* To see this, consider the loop in Figure 1(a) from the benchmark `otter` that traverses a linked list. For the sake of this discussion, assume that the `if` condition in line 4 is rarely taken. In that case, a TLS system can speculate that the read of `wm` in the next iteration does not conflict with the write of `wm` in the current iteration. It can then predict the value of `c` at the beginning of the next iteration and use the predicted value of `c` to run that iteration concurrently with the current iteration. But, it can also speculatively parallelize the loop by only predicting the value of `c` every tenth iteration, instead of every iteration. In that case, the TLS system can speculatively execute *chunks of 10 iterations* in parallel. If the predictions are highly accurate, then it is sufficient to predict only as many values as the number of speculative threads. Predicting more values does not increase the amount of TLP. This is in contrast to value prediction for ILP. ILP value prediction techniques predict values of a long latency operation **op** to speculatively execute the dependent operations. For every dynamic instance of **op** that is not predicted, the dependent operations would stall in the pipeline, thereby reducing the ILP.

The second key insight is that *the probability of predicting that an operation will produce a particular value* **some time** *in the future is higher than predicting that that value will be produced at a* **specific time** *in the future.* Predicting that a value will appear some time in the future is sufficient for the purposes of extracting TLP. To give an analogy from the stock market, the chances that the Dow Jones index will cross 15000 exactly a year from now is much lower than the chances that it will cross 15000 *some time* within the next 2 years. To give a more concrete example, consider Figure 1(b) which shows the linked list and how it gets modified and accessed over time. Consider a simple predictor that predicts that node 4 appears on the list. In this example, the prediction is always true, even though the relative position of node 4 from the head of the list changes over time. Thus, such a predictor will have a higher prediction rate than a predictor that predicts that node 4 appears as the fourth element from the head of the list.

This paper proposes a speculative parallelization technique called Spice. Spice uses a value prediction technique based on the above two insights. For every static value that needs to be predicted, our technique predicts only a small set of values produced by that operation in a given loop invocation. It does not predict the specific iteration in which those values will be produced. This prediction strategy is based on the observation that loop iteration live-in values tend to repeat across loop invocations.

The paper makes the following contributions:

1. It presents a viable software-only value prediction technique to break loop-carried dependences that can enable speculative parallelization based on key insights into value prediction for TLP.
2. It presents an automated speculative transformation called Spice (speculative parallel iteration chunk execution) that uses this value prediction methodology and a dynamic load balancing scheme to extract thread-level parallelism.
3. It presents an experimental evaluation with detailed performance simulations of Spice-parallelized codes of applications with linked list traversals, tree traversals, and complex control flow.
4. It describes a value profiler framework to gauge loop live-in predictability in order to automatically identify program loops amenable for Spice-parallelization. Profile results show that there is good loop live-in predictability across loop invocations for a wide variety of applications.

The rest of this paper is organized as follows. Section 2 uses an example to highlight the performance potential of Spice in comparison to other TLS techniques. Section 3 describes the architectural support that would be needed to support Spice TLS. The automatic Spice transformation is presented in Section 4, followed by a quantitative evaluation in Section 5. Section 6 describes our value profiling framework, provides a whole-application characterization of loop live-in predictability across several benchmarks, and discusses the issues and challenges in integrating such a profiler into an automatic transformation framework for Spice. We compare and contrast the contributions of this paper with related work in Section 7. Finally, Section 8 summarizes the key contributions of this work.

2. MOTIVATION

This section provides a qualitative comparison of code generated by existing TLS techniques, both with and without value speculation, and the code generated by the Spice transformation, for the loop in Figure 1(a).

2.1 TLS Without Value Speculation

Figure 2 shows how the loop from Figure 1(a) would be executed by existing TLS techniques that do not employ value speculation on two processor cores. The solid lines represent the execution of the

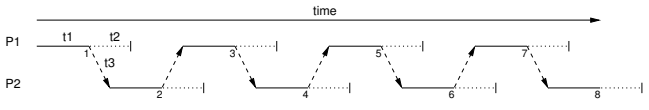


Figure 2: Execution Schedule for TLS

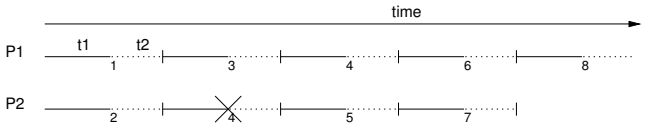


Figure 3: Execution Schedule for TLS with Value Prediction

code that performs the list traversal which is synchronized across the iterations. The dotted lines represent the code corresponding to the computation of the minimum element, and the dashed lines represent the forwarding of values from one thread to another thread. The numbers below the lines indicate the iteration numbers. Let t_1, t_2 and t_3 respectively denote the latency of each of the above three parts of the execution in an ideal execution model where there is no variance in the execution time of these three components due to microarchitectural effects such as cache misses and branch mispredictions. Let the total number of iterations of the loop be $2n$.

The total time taken to execute the loop by TLS depends on the relation between t_1, t_2 and t_3. If $t_2 > t_1 + 2 \times t_3$, then the computation of cm and wm lies on the critical path. In that case, the total execution time is roughly equal to $n \times (t_1 + t_2)$, resulting in a $2X$ speedup over single threaded code. On the other hand, if the minimum computation is not on the critical path, then the communication of values from one core to another, will also be a part of the critical path. In particular, if $t_2 \leq t_3$, then the total execution time becomes $2n \times (t_1 + t_3)$. This results in a speedup of $\frac{t_1+t_2}{t_1+t_3}$, which is always less than 2, over single-threaded execution. For the example in Fig 1, the list traversal is indeed on the critical path. This is because of the high cache miss rate of executing the pointer chasing load and also because of our assumption that the if statement mostly evaluates to false . Thus, the expected speedup of this loop in the ideal case is $\frac{t_1+t_2}{t_1+t_3}$. From this discussion, it is clear that the performance of TLS is susceptible to inter-core communication latencies.

2.2 TLS With Value Speculation

Let us now consider the case where the TLS technique employs value prediction to predict the value of c, eliminating the value forwarding of c between iterations. Figure 3 shows an execution schedule of the loop under TLS with value prediction. In the example, the prediction of iteration 4 is shown to be wrong, causing iteration 4 to be mis-speculated and re-executed. Let p denote the probability that a given value prediction is correct. Assuming that the probability of a prediction being successful is independent of other predictions, the expected speedup is $\frac{2n(t_1+t_2)}{(n+(1-p)n)(t_1+t_2)}$ or $\frac{2}{2-p}$. If *all* the values of c are successfully predicted, then TLS will give a $2X$ speedup. Successful prediction depends both on the nature of the code that produces the values and the predictor that is used. In this example, the values are produced by the nodes of a linked list. In otter , between successive invocations of the loop in Figure 1(a), the minimum element found by the loop is removed from the list and some other nodes are inserted into the list. Given this, let us now consider various value predictors and see how successful they would be in predicting the values of c.

- The simplest value predictor is the *last value predictor* that predicts that an instruction will produce the same value produced by the immediately preceding dynamic instance of that instruction. Obviously, such a predictor can not predict the address of the nodes of a linked list.

- Another common predictor is the stride predictor. While this is most suited for predicting array addresses, it can also predict linked list nodes as long as the nodes are allocated contiguously in the heap and the order of the nodes seen during the traversal matches the order in which the nodes were allocated. But, in this example, even if the nodes are allocated contiguously, a stride predictor can not successfully predict all the values of c since the insertions and deletions cause the traversal order to be different from the allocation order.

- Some TLS techniques use trace based predictors instead of instruction based predictors. Instead of predicting a value based on which instruction produces that value, these predictors try to exploit the correlation of values produced by different instructions in the same trace. Marcuello et al. [14] proposed the use of trace-based predictors for TLS. They proposed a predictor called *increment predictor* which is a trace based equivalent of a stride predictor. The traces they use are loop iteration traces, which are unique paths taken by the program within a loop iteration. There are two paths in our example loop and in both these paths, the value c is produced only once, by the same instruction. Hence, for this example, a trace based predictor would fare no better than an instruction based predictor.

Thus, even if value prediction is employed, it is unlikely that existing TLS techniques would significantly improve the performance of this loop. In general, for application loops with irregular memory accesses and complex control flow, conventional value predictors fail to do a good job. The next subsection describes how Spice predicts values by *memoizing* the values seen during the previous invocation of the loop to break previously hard-to-predict dependences with very low mis-speculation rates.

2.3 Spice Transformation With Selective Loop Live-in Value Speculation

Let us now see how Spice would transform the same loop. Figure 4 shows the parallel version of the loop in Figure 1 using Spice. The example shows parallelization with two threads, but it can be generalized to any number of threads. For the two threads case, only one value of c has to be predicted since there is only one speculative thread. Assume that the variable predicted_c contains that predicted value of c. Later, in our discussion on the compiler transformation, we describe how this value could be obtained. Both threads execute the original loop, but with some differences. In the main non-speculative thread, we add a check at the end of each loop iteration that checks if the current value of c equals the predicted value, in which case the main thread sets a flag indicating a successful speculation and exits the loop. Outside the loop, it checks if the flag indicating successful speculation is set. If it had exited the loop because of a successful speculation, the main thread receives the wm and cm values from the speculative thread and computes the minimum among the two wms and the corresponding cm. In case of mis-speculation, the speculative thread is squashed. In the speculated thread, the value of c is initialized predicted_c . When it exits the loop, the speculative thread sends the cm and wm values to the main thread.

```
1   c = cm->next_c1;
2   mispred  = 1;
3   while(c  != NULL){
4     int w = c->pick_weight;
5     if (w < wm) {
6       wm = w;
7       cm = c;
8     }
9     c = c->next_c1;
10    if(c == predicted_c)    {
11        mispred  = 0;
12        break;
13    }
14
15  }
16  if(!mispred){
17      receive(thread2,     wm2);
18      receive(thread2,     cm2);
19      if(wm2  < wm){
20          wm = wm2;
21          cm = cm2;
22      }
23  }
24  else{
25      squash_speculative_thread();
26  }
```

(a) Non-speculative thread (Thread 1)

```
1   c = predicted_c;
2   while(c  != NULL){
3     int w = c->pick_weight;
4     if (w < wm) {
5       wm = w;
6       cm = c;
7     }
8     c = c->next_c1;
9   }
10  send(thread1,    cm);
11  send(thread1,    wm);
```

(b) Speculative thread (Thread 2)

Figure 4: Parallelization using Spice

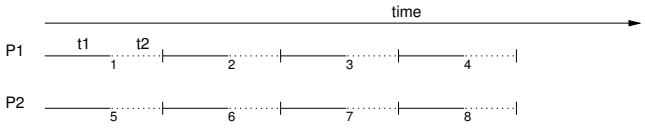


Figure 5: Execution Schedule for Spice

Figure 5 shows the execution schedule for this transformed code. Instead of alternating the iterations across the two processor cores, Spice splits the iteration space into two halves and executes both the halves concurrently in two different cores. Assuming again a probability p of successful prediction, and that the predicted value splits the list in the middle, applying Spice results in an expected speedup of $\frac{2}{2-p}$. Thus, if we have a predictor that can predict just a *few* values with higher accuracy, then the expected speedup of Spice would be higher than existing TLS schemes, since Spice requires fewer values to be speculated than the existing TLS schemes to produce a given number of threads. Consider a simple value prediction strategy where on every loop invocation, the value of `c` in the middle of the list is remembered and used as the predicted value in the following invocation. For the example in Figure 1(a), this simple strategy is likely to succeed since only one node is deleted from the list after each invocation and hence the probability of the remembered node being removed from the list is low.

In general, given n processor cores, the loop in Figure 1(a) can be parallelized into n parallel speculative threads by predicting only $n-1$ values of `c` across the entire iteration space[1]. Each of these threads executes one chunk of iterations and has a low mis-speculation rate. Each Spice thread is long-running compared to iteration-granular TLS. Consequently, Spice TLS does not incur frequent thread management overhead. In the following two sections, we describe the implementation details of Spice including the required architectural support and the compiler transformation.

3. ARCHITECTURAL SUPPORT

In this section, we describe the architectural support required to support Spice. Since Spice executes multiple threads concurrently, it naturally requires a multi-core architecture to execute those threads. Since Spice employs speculation, it requires the following additional support:

Speculative State When a speculative thread generated by Spice needs to be squashed due to mis-speculation, any changes to architectural state made by the speculative thread must be undone. Undoing the changes to register state involves saving and restoring some register values and discarding the rest and can be done purely in software. But to undo the changes to memory, special hardware support is required. Such support is provided by hardware transactional memory [9] and memory systems for thread level speculation [8, 21] which buffer speculative state and discard it on mis-speculation. When the program reaches a point where a speculative thread can not be squashed anymore, the buffered state is committed into the main memory and the state can no longer be rolled back. Our architectural model includes ISA support to enter into a speculative state, commit the speculative state and discard the buffered state.

Conflict Detection Like other TLS techniques, Spice can use memory alias speculation in addition to value speculation. This requires hardware support to detect if a store and load actually conflict during execution. Many existing TLS systems [8, 21] provide such hardware support.

Remote resteer When a thread is found to have mis-speculated, the thread that detects the mis-speculation forces it to execute the recovery code. To support this, we propose a *remote resteer* mechanism that allows one thread to transfer control in another thread. We currently support it using a special instruction called *resteer*.

4. COMPILER IMPLEMENTATION

In this section, we describe how Spice is implemented as an automatic compiler transformation. Algorithm 1 outlines the Spice transformation. The algorithm takes the loop to be parallelized and the number of threads to create as its inputs. The algorithm first computes the set of live-ins that require value prediction. This set is obtained by first computing the set of all inter-iteration live-ins. Those live-ins in this set that can be subjected to reduction transformations [1] such as sum reduction or MIN/MAX reduction do not require prediction. Value prediction is applied to the rest of the loop carried live-ins. After obtaining this set, the compiler then performs the following steps:

[1]A reduction transformation can remove the loop-carried dependence for `wm`.

Algorithm 1 Spice transformation

1: Input: Loop L, number of threads t
2: Compute inter-iteration live-ins *Liveins*
3: Compute reduction candidates *Reductions*
4: Live-ins to be speculated $S = Liveins - Reductions$
5: Create $t-1$ copies of the body of L to form $t-1$ procedures
6: Insert communication for non-speculative loop live-ins and live-outs
7: Generate code to initialize speculative live-ins S
8: Generate recovery code in speculative threads
9: Insert code for mis-speculation detection and recovery
10: Insert value predictor

Thread creation: The compiler replicates the loop $t-1$ times and places the loop copies in separate procedures. Each of these procedures is executed in a separate thread. To avoid spawning these threads before every invocation of the loop, threads are pre-allocated to the cores at the entry to the main thread. Code is inserted to the main thread loop's preheader to generate a `new_invocation` token to all the threads before each loop invocation. All other threads wait on this `new_invocation` token and start the loop when they receive this token.

Communication of live-ins and live-outs: The compiler identifies the set of register live-ins to the loop that needs to be communicated to the speculative threads. All live-ins except the speculative live-in set S and the set of accumulators are communicated. Variables used as accumulators are not communicated since they have to be initialized to 0 in the speculative threads.

Value speculation: The compiler creates a global data structure called the *speculated values array* of size $(t-1) \times m$, where m is the number of live-in values that need to be speculated. The compiler initializes the speculative live-ins of the loop in thread i with the values from the $(i-1)^{th}$ row of the speculated values array. Later we discuss how the contents of this array are obtained.

Recovery code generation: The compiler creates a recovery block for each speculative thread and generates code to perform the following actions:

1. Restore machine specific registers such as the stack pointer and the flag register. Registers used within the loop that is parallelized are simply discarded.
2. Rollback the memory state if the loop contains stores.
3. Inform the main thread that recovery is complete.
4. Exit the recovery block and jump to the program point where it waits for the main thread to send a token that denotes the beginning of the next invocation

Mis-speculation detection and recovery: Mis-speculation detection is done in a distributed fashion. We first look at mis-speculation detection when there is one speculative thread (a total of 2 threads) and then generalize it for t threads. Let S be the set of all the loop live-in registers that need speculation. At the beginning of each loop iteration, the non-speculative thread 1, which is also referred to as the *main thread*, compares its values of the registers in S with the values used to initialize those live-in registers in thread 2. If all the values match at the beginning of some iteration j, it implies that thread 2 started its execution from iteration $j+1$ of the loop with correctly speculated live-in values. In that case, thread 1 stops executing the loop at the end of iteration j and waits for thread 2 to communicate its live-outs at the end of the loop. On the other hand, if the values *never* match, thread 1 will eventually exit the loop by taking the original loop exit branch. Since it has executed all the iterations of the loop and exited the loop normally, it can conclude that thread 2 had mis-speculated. In that case it executes a `resteer` instruction to redirect thread 2 to its mis-speculation recovery code.

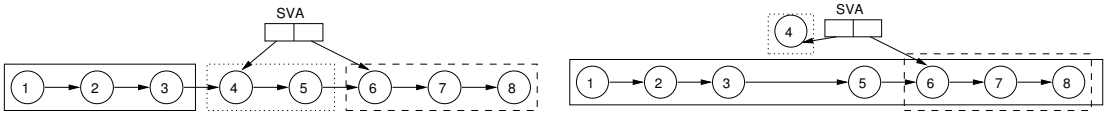
Mis-speculation detection and recovery can be generalized for t threads. Thread i is responsible for detecting whether thread $i+1$ has mis-speculated in a loop invocation. The compiler generates code in thread i at the beginning of each loop iteration to compare the values of all the registers in set S with the initial values of thread $(i+1)$'s live-ins. Thread $(i+1)$'s initial live-in values are loaded from the i^{th} row of the speculated values array. It then inserts code that sets a flag indicating successful speculation followed by a branch to exit the loop, if the values match. Outside the loop, code to check this flag and take the necessary recovery action is generated.

To recover from mis-speculation, the compiler generates code to perform the following actions. The thread detecting mis-speculation, if itself is not the main thread, communicates this information to the main thread. The main thread issues `resteer` instructions to all the mis-speculated threads that make these threads execute their recovery code. It then waits for an acknowledgment token from each of them that indicates that they have successfully rolled back the memory state. Finally, after all the tokens have been received, the main thread commits the current memory state.

Mis-speculation detection is illustrated in Figure 6. Figure 6(a) shows the traversal of a list that has 8 nodes. Three threads participate in the traversal of this list. The first thread traverses the nodes enclosed by the solid box, the second thread traverses the nodes enclosed by the dotted box and the nodes of the last thread are enclosed by the dashed box. SVA denotes the speculated values array whose elements contain the addresses of list nodes which are live-in to the list. Assume that after the invocation, node 4 is removed from the list as in Figure 6(b). The SVA entry still points to the removed node. When the loop is invoked again, the first thread traverses the entire list since it never finds the node whose address is in the SVA during its traversal as the node has been removed from the list. The second thread starts from the removed node and depending on the what its next pointer points to, will either stop the traversal, loop forever, or cause memory faults by accessing some invalid memory location. Thread 3 starts from node 6 and traverses till the end of the list, repeating the work done by thread 1. When thread 1 reaches the end of the list, it concludes that thread 2 has mis-speculated and squashes threads 2 and 3. Note that if thread 1 compares its live-in registers with the speculated live-ins of both threads 2 and 3, then it needs to squash only thread 2 and thread 3's computation would not have been wasted. But this increases the overhead in thread 1 due to the additional comparisons and hence in our current implementation, the compiler limits the comparisons to only one set of live-ins. If thread 2 goes into an infinite loop, the `resteer` issued by thread 1 will redirect it to the correct recovery code. The recovery code rolls back its memory state to undo any changes the thread has done to its memory state.

Value predictor insertion

The compiler inserts into the threads a value predictor that fills the contents of the speculated values array on every loop invocation. A trivial value prediction strategy is to *memoize* or remember the live-ins from $t-1$ different iterations during the first invocation. These $t-1$ set of live-ins can be used as the predicted values in all subsequent invocations. This approach does not adapt to change in program behavior and hence does not work in many cases. For



(a) Original value prediction

(b) After removal of node 4 from the list

Figure 6: List Traversal Example

instance if the values that are memoized are addresses of the nodes nodes of a linked list, and if a memoized node is removed from the list, all subsequent invocations will mis-speculate even if the list elements hardly change subsequently.

A better approach is to make the predictor memoize the values not just once, but on every invocation of the loop. Values memoized in one invocation are used as a prediction only in the next invocation. This approach adapts itself better to changes in the loop behavior. Moreover, this also allows dynamic load balancing since the work done by the speculative threads depends on the iterations whose live-in values are predicted.

The value predictor has two components. The first component of the value predictor that writes to the speculated values array is distributed across all the threads. To implement this component, the compiler first creates a set of data structures used by the predictor. A list called `svai` is created per thread. The entries of this array for thread *i* contain the indices to the speculated values array (`sva`) to which thread *i* should write to. Another per-thread list called `svat` contains thresholds that determine which values are memoized. The compiler also creates an array called `work` whose entries contain the amount of work done by each thread. This is used in dynamic load balancing.

Algorithm 2 Spice value prediction

```
my-work = 0
for each iteration of the loop do
  my-work = my-work + 1
  if my-work > svat[list-index] then
    sva[svai[list-index]] = loop carried live-ins in current iteration
    list-index = list-index+1
  end if
end for
work[my-thread-id] = my-work
```

The compiler emits the code corresponding to Algorithm 2 in each thread to memoize the values. Each thread maintains a counter `my-work` that is a measure of the amount of work done by that thread. In our current implementation, the threads increment the counter once per loop iteration. A more accurate measure of the work done could be obtained by reading the relevant hardware counters periodically. If the work done so far exceeds the threshold found in the head of the `svat` list for this thread, then the current loop live-in values are recorded in the `sva` array. The index to the `sva` array location to which the value has to be written is given by the value at the head of the `svai` list. After writing to the `sva` array, the head pointers of both the lists are incremented. After exiting the loop, the thread writes the total work done in that invocation to the global `work` array.

The other component of the value predictor is centrally implemented in a separate procedure. This component is executed at the end of each loop invocation. It collects the amount of work done by each thread in that invocation and decides on the iterations of the next loop invocation whose live-in values have to be memoized. Depending on which threads execute those iterations, it generates the entries of the `svai` list. But how the iterations will be partitioned among the threads in the next invocation can not be decided a priori at the end of current invocation since it depends on the program state. Hence, the predictor makes the following assumptions:

1. In future invocations, the total amount of work done will remain the same.
2. In the immediately following invocation, the three threads will execute the same amount of work.

These assumptions enable the predictor to orchestrate the memoization in a way that results in load balancing. This is best illustrated with an example. Consider the case where there are three threads where the work done by each thread in a particular invocation are 10, 1 and 1 units respectively. Based on the first assumption, the predictor wants to collect the live-ins when the total work done in a sequential execution are 4 and 8. It then uses the second assumption to determine which threads will have to write the values. In this example, the work done by the first thread in the next invocation is assumed to be 8 and so both the rows of the `sva` are written to by the first thread after iterations 4 and 8 and the other two threads do not write into the `sva`. Hence the `svat` list of the first thread is set to [4, 8] and its `svai` list is set to [0,1]. For the other two threads the head element of `svat` is set to ∞ so that they would never write to the `sva` array.

Core	Functional Units: 6 issue, 6 ALU, 4 memory, 2 FP, 3 branch L1I Cache: 1 cycle, 16 KB, 4-way, 64B lines L1D Cache: 1 cycle, 16 KB, 4-way, 64B lines, write-through L2 Cache: 5,7,9 cycles, 256KB, 8-way, 128B lines, write-back Maximum Outstanding Loads: 16
Shared L3 Cache	> 12 cycles, 1.5 MB, 12-way, 128B lines, write-back
Main Memory	Latency: 141 cycles
Coherence	Snoop-based, write-invalidate protocol
L3 Bus	16-byte, 1-cycle, 3-stage pipelined, split-transaction bus with round robin arbitration

Table 1: Machine details.

5. EXPERIMENTAL EVALUATION

We now describe our experimental evaluation of Spice. We use a cycle accurate multi-core simulator to evaluate the performance benefits of using Spice. Our base model is a 4 core Itanium 2 CMP model modeled using the Liberty simulation environment (LSE). The architectural details of each core in our model are given in Table 1. To support the remote resteer mechanism, we make the `resteer` instruction take a program address to which the control is transferred in a different core. This could also be implemented through an OS call that sends a signal to the remote thread. While

Benchmark	Description	Loop	Hotness
ks	Kernighan-Lin graph partitioning	FindMaxGpAndSwap (inner loop)	98%
otter	theorem prover for first-order logic	find_lightest_cl	20%
181.mcf	vehicle scheduling	refresh_potential	30%
458.sjeng	chess software	std_eval	26%

Table 2: Benchmark Details

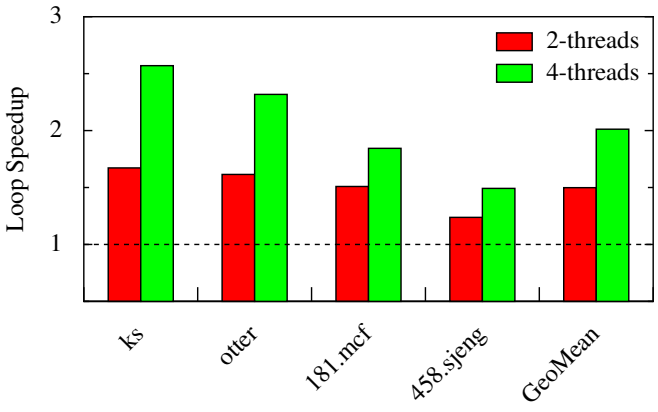


Figure 7: Spice Results

the cores have the ability to buffer memory state to enable rollback of the speculative state, our architecture simulator does not support detection of conflicts in hardware. This restricts the loops to which we can apply the transformation. The set of benchmarks we use to evaluate the technique is given in Table 2. These loops are chosen from a set of loops that are found, by manual inspection of the source code, to be good candidates for this technique but are not DOALL. From that set, we removed loops that do not execute for a significant fraction of the program's execution and those that require memory conflict detection. Section 6 describes a profiling framework that can be used to automate the above process of determining the loops to which Spice can be applied.

The program is first compiled to a low level intermediate representation. Classical optimizations are applied to this low level code before applying the Spice transformation. Translation to IA-64 assembly is followed by another round of classical optimizations, scheduling, register allocation and a final scheduling phase. The results reported are in terms of loop speedup over single threaded binary, which is subjected to the same sequence of optimization passes except Spice. Except for the loop being optimized, the rest of the application remains unchanged. Consequently, detailed performance simulation is done only for the loops being optimized. The caches and branch predictors are kept warm while the rest of the application is executed on a fast-forward mode.

Figure 7 shows the speedups obtained on these 4 loops. The speedup numbers are shown for both the 2 threads and the 4 threads cases. The speedups range from 24% in 458.sjeng with 2 threads to as high as 157% on the loop in ks with 4 threads. The whole program speedup can be obtained from the loop speedup and the fraction of time spent on the loop in the original program. There are several factors that prevent the actual speedup from being in line with the ideal linear speedup expected from threads executing in a DOALL fashion:

Mis-speculations 458.sjeng is the only benchmark that suffers heavily due to mis-speculation. Around 25% of the invocations mis-speculate in this benchmark. In the other three loops, the mis-speculation rate is less than 1%.

Load imbalance The number of iterations per invocation varies in otter due to insertions to the linked list. Load balancing plays an important role in speeding up the loop's execution. Since our distributed re-memoization strategy does not equally divide the work among the different cores, it results in some slowdown compared to an ideal case where the work among the threads is equally divided. In 458.sjeng, even though the variance in the number of iterations per invocation is not very high, the actual number of instructions executed per iteration varies across iterations. A better metric for load balancing than just iteration counts would improve the speedup. Load balancing is also an issue in 181.mcf due to the variability in the number of iterations of the inner loop per outer loop iteration.

Speculation overhead Even when there is no mis-speculation, there is an overhead in checking for mis-speculation every iteration. In otter, the time to execute the loop body per iteration is low and hence the overhead, as a fraction of useful execution time, is high. In 458.sjeng, the overhead is high because there are 8 distinct live-in values that are compared with the memoized values of the next thread and ANDed together to determine whether the speculation is successful.

Other overheads The main thread has to communicate live-in values that are defined outside the loop to all the other threads, collect all live-out values and perform operations such as reductions at the end of each invocation. This overhead is typically low as it is paid only once per invocation. However, certain early invocations of the otter loop have small trip count causing this overhead to become a significant performance factor.

The results show that the proposed technique is a viable parallelization technique for certain classes of loops. While the above parallelization opportunities were manually identified to the compiler, the next section presents a value profiling framework which can be used to profile applications for loop live-in predictability across loop invocations and enable the compiler to automatically identify opportunities.

6. VALUE PROFILER

This section describes a generic value profiling framework to characterize the predictability of loop live-in values across loop invocations. This characterization will enable us to understand the limits and the potential of Spice. Practical issues that must be overcome to enable the use of this profiling framework for automatic identification of loops for Spice-parallelization are also discussed.

The profiler takes a program in low level IR, and identifies the set of loops whose loop-carried live-ins are predictable. The profiler consists of two components: an *instrumenter* and an *analyzer*. We now describe these two components in detail.

6.1 Instrumenter

The first step of the instrumentation process is to identify the set of loops that are candidates for value profiling. Only loops that execute for more than 0.5% of the total execution time, as determined by the dynamic instruction count, and are not DOALL-able, are

considered for value profiling. For these loops, the instrumenter identifies the set of values that are live-in across loop iterations. The instrumenter then further trims this set as described below.

Reductions

It first identifies the set of operations that are candidates for reduction transformations [1]. For instance, if the loop has an accumulator variable `sum` which is incremented by the statement `sum = sum + A[i]`, then the variable `sum` is live-in at the loop header. But this does not prevent the loop from being parallelized if `sum` is not read or written anywhere else in the loop, since the parallel threads can independently compute the value of `sum`, and finally, after all the threads have completed their executions, add the values to get `sum`. Hence this live-in can be removed from the set. Similarly live-ins used to compute other associative operations such as multiplication, computation of minimum or maximum of an array, are removed from the set. Removing such loop-carried dependences, which can be handled in other ways, is important to obtain meaningful results from the profiler.

Low Frequency Memory Dependences

If the program on which the instrumenter is applied has memory profiling annotations then the instrumenter can use the memory conflict frequency information to exclude loop live-ins created by low frequency memory dependences from the corresponding loop's live-in set. Any resulting Spice parallelization will then need runtime memory conflict detection to handle any memory dependence violations caused by one of these excluded loop live-ins.

The instrumenter then instruments the program to record the set of loop carried live-ins for the loops under consideration. For each loop, an array is allocated whose size is set to the cardinality of the loop carried live-in set. At the entry to the loop, the set of register live-ins are stored at distinct locations of this array. For each memory live-in that loads a value into a register `r`, the instrumenter inserts the code to store the register `r` into the array after the load operation.

At the loop preheader, the instrumenter inserts a call to an analyzer method `new_invocation` that informs the analyzer the start of a new invocation of a loop which is uniquely identified by the argument to the method. At the end of the loop iteration, before the backward branch to the header of the loop, the instrumenter inserts a call to an analysis routine `record_values` passing the array containing the live-ins for that loop. Later we describe how the analyzer processes this array. Finally, the instrumenter inserts a call to an analyzer method `exit_program` to inform the analyzer that the program is going to exit making the analyzer output its analysis.

Loads that are contained within inner loops pose a problem during instrumentation. For a given iteration of the outer loop, a load in an inner loop can get executed more than once. However, only the value produced by the first dynamic instance of the load in the outer loop execution is considered loop-carried with respect to the outer loop and so only that value needs to be profiled. This can be achieved by associating a guard variable for each such load. A load is profiled only if the associated guard is true. The instrumenter ensures that the guard is set to true only for the first dynamic instance of that load in an outer loop iteration

6.2 Analyzer

The analyzer processes the set of live-in values in each loop iteration to determine if the live-ins of the loop exhibit predictability across invocations. For each loop L, the profiler associates a probability $P(L)$ and the invocations of L are sampled with that probability. Sampling is done to reduce the profiler overhead. For each sampled invocation, the analyzer gets the set of live-in values every iteration through the call to `record_values` method. It computes a signature s of the live-in values. All such signatures in an invocation are added to a set S. In the following invocation, the analyzer again computes the signature s of the live-in set and searches the signature set S to see if the signature s is found. It then computes the fraction of loop iterations f in which the signature is found in S. If f is above some threshold t, the analyzer considers the loop invocation to be predictable.

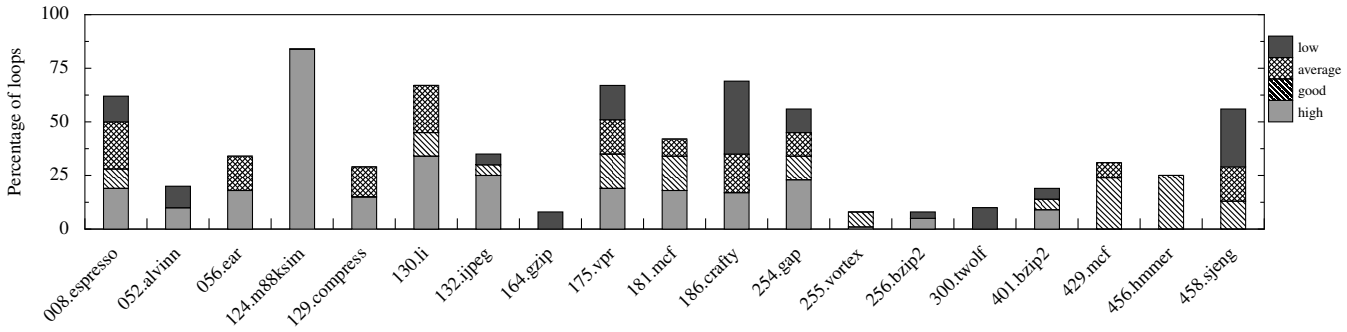
6.3 Predictability of Values

Figure 8 summarizes the results from the profiler on a set of benchmarks from the SPEC integer suites, the Mediabench suite and a few other applications. We use a threshold of 0.5 to determine if a loop invocation is predictable. In other words, a loop invocation is considered predictable if the live-ins of more than half its iterations match live-ins from the previous invocation. We then classify each loop into one of four predictability bins based on what percentage of its invocations are predictable: low (1-25%), average (26-50%), good (51-75%), and high (76-100%). The number of loops in each benchmark that fall under each of these bins is shown in Figure 8. The loop counts are normalized to 100. Missing bars for benchmarks indicate that none of the invocations in any of the loops show predictability. The above figure indicates that in many of the applications, a significant fraction of their dynamic loop invocations show good to high predictability. We expect that a good number of these loops will benefit from Spice TLS.

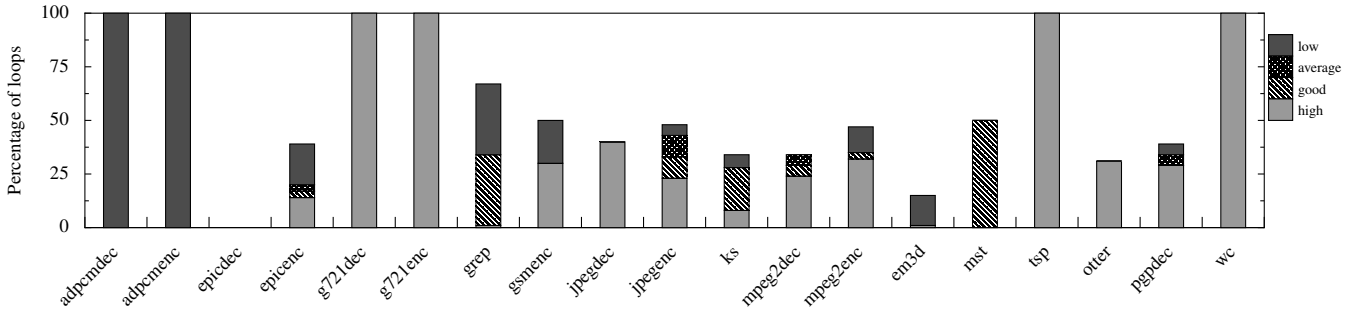
As mentioned in Section 5, while the Spice transformation has been automated in our research compiler, the specific parallelization opportunities themselves had to be manually identified to the compiler. The use of the above profiler output to automatically detect opportunities for Spice-parallelization was precluded due to a few missing pieces in the automation process. In particular, the predictability values returned by the value profiler will have to be taken into account in conjunction with other information such as the structure of the loop nest tree and the cumulative execution weight, the total number of invocations, and the number of iterations per invocation of each node in the loop nest tree. Furthermore, if both outer and inner loops of a loop nest have desirable execution and value profile characteristics, then the compiler has to use some heuristic to decide what the right granularity of Spice-parallelization is. Outer loops typically execute for many iterations and hence, when Spice-parallelized, can yield significant performance improvements. On the flip side, if the outer loop executes for very few invocations, the fraction of invocations that are successfully parallelized with good load balancing could be low. Finally, the absence of the hardware for memory conflict detection precludes the technique from being applied to loops with potential inter-iteration load store conflicts. We are currently working on formulating and incorporating these heuristics in our compiler to evaluate the speedup potential of Spice across entire applications.

7. RELATED WORK

Lipasti [10, 11] showed that data values produced by instructions during program execution show a lot of predictability and proposed the use of value predictors to exploit this predictability. In our work, we employ value prediction to only a few selected dynamic instances of a static operation. Calder et al. [2] also proposed a technique called *selective value prediction*. While they propose instruction filtering mechanisms to determine which instructions write into the value prediction table in a uniprocessor, ours is a purely software based value prediction with the goal of extracting



(a) SPEC integer benchmarks



(b) Mediabench and others

Figure 8: Value predictability of loops in applications. Each loop is placed into 4 predictability bins (low, average, good and high) and the percentage of loops in each bin is shown.

thread level parallelism. Marcuello et al. [14] proposed the use of value prediction for speculative multi-threaded architectures. They investigated both instruction based predictors and trace based predictors, in which the predictor uses the loop iteration trace as the prediction context. Steffan et al. [22] propose the use of value prediction as part of a suite of techniques to improve value communication in thread level speculation. They use a hybrid predictor that chooses between a stride predictor and a context based predictor. Cintra and Torellas [4] also propose value prediction as a mechanism to improve TLS. They use a simple *silent store predictor* that predicts the value of a load to be the last value written to that location in the main memory. Liu et al [12] predicts only the values of variables that look like induction variables. Oplinger [15] also incorporates value prediction in TLS design, with a particular focus on function return values. Our work focuses on a compiler based technique to predict a few values with high accuracy. Zilles [23] proposed Master/Slave speculative parallelization(MSSP), which is a speculative multi-threading technique that uses value speculation. The prediction in MSSP is made by a *distilled program*, which executes the speculative backward slice of the loop live-ins. Our software value predictor predicts based on values seen in the past and does not have to execute a separate thread for prediction.

Some TLS techniques [3, 4, 16, 17] have also proposed the use of iteration chunking. These chunks are iterations of fixed size, while in our case the number of chunks equals the number of processor cores and the chunk size is determined at runtime by the load balancing algorithm. The LRPD test [18] and the R-LRPD test [6] are also speculative parallelization techniques that chunk the iteration space, but they use memory dependence speculation and not value speculation.

8. CONCLUSION

This paper presented a new speculative parallelization technique called Spice that uses a novel value prediction mechanism for thread level parallelism. The value prediction mechanism is based on two new insights on the differences between value speculation for ILP and TLP. The highly accurate and targeted value prediction is accomplished by the compiler without any support from the hardware. The paper described in detail the algorithms, compiler implementation and the hardware support required to automatically apply Spice. We evaluated Spice on a set of 4 loops from general-purpose applications with complex control flow and memory access patterns. Spice shows up to 157% (101% on average) speedup on the set of loops to which it is applied. We also presented a value profiler which demonstrates that the kind of value predictability exploited by Spice is present in many benchmarks, indicating the wider applicability of Spice.

Acknowledgments

We thank the entire Liberty Research Group for their support and feedback during this work. Additionally, we thank the anonymous reviewers for their insightful comments. The authors acknowledge the support of the GSRC Focus Center, one of five research centers

funded under the Focus Center Research Program, a Semiconductor Research Corporation program. This work has been supported by Intel Corporation. Opinions, findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of our sponsors.

9. REFERENCES

[1] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[2] B. Calder, G. Reinman, and D. Tullsen. Selective value prediction. pages 64–74, 1999.

[3] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 13–24, New York, NY, USA, 2000. ACM Press.

[4] M. Cintra and J. Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 43. IEEE Computer Society, 2002.

[5] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, 1996.

[6] F. H. Dang, H. Yu, and L. Rauchwerger. The R-LRPD test: Speculative parallelization of partially parallel loops. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 318, 2002.

[7] P. A. Emrath and D. A. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 89–99, New York, NY, USA, 1988. ACM Press.

[8] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, 2000.

[9] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[10] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 226–237, December 1996.

[11] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, September 1996.

[12] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. Posh: a tls compiler that exploits program structure. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 158–167, 2006.

[13] P. Marcuello and A. González. Clustered speculative multithreaded processors. In *Proceedings of the 13th International Conference on Supercomputing*, pages 365–372, New York, NY, USA, 1999. ACM Press.

[14] P. Marcuello, J. Tubella, and A. Gonzalez. Value prediction for speculative multithreaded architectures. In *Proceedings of the 32nd annual ACM/IEEE International Symposium on Microarchitecture*. ACM Press, 1999.

[15] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, page 303, Washington, DC, USA, 1999. IEEE Computer Society.

[16] M. K. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, New York, NY, USA, 2003. ACM Press.

[17] M. K. Prabhu and K. Olukotun. Exposing speculative thread parallelism in SPEC2000. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 142–152, New York, NY, USA, 2005. ACM Press.

[18] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, 1999.

[19] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[20] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22th International Symposium on Computer Architecture*, June 1995.

[21] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, 2005.

[22] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *Proceedings of the 8th International Symposium on High Performance Computer Architecture*, pages 65–80, February 2002.

[23] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 85–96, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.