Softspec: Software-based Speculative Parallelism

Derek Bruening, Srikrishna Devabhaktuni, Saman Amarasinghe
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
iye@mit.edu

Abstract

We present Softspec, a technique for parallelizing sequential applications using only simple software mechanisms, requiring no complex program analysis or hardware support. Softspec parallelizes loops whose memory references are stride-predictable. By detecting and speculatively executing potential parallelism at runtime Softspec succeeds in parallelizing loops whose memory access patterns are statically indeterminable. For example, Softspec can parallelize while loops with un-analyzable exit conditions, linked list traversals, and sparse matrix applications with predictable memory patterns. We show performance results using our prototype implementation.

1 Introduction

Parallel processing can provide scalable performance improvements, and multiprocessor hardware is becoming widely available. However, it is difficult to develop, debug, and maintain parallel code. Compilers can automatically parallelize some sequential applications but are limited in the type of code that they can parallelize. In order to identify parallel regions of code they must use complex interprocedural analyses to prove that the code has no data dependences for all possible inputs. Typical code targeted by these compilers consists of nested loops with affine array accesses written in a language such as FORTRAN that has limited aliasing. Large systems written in modern languages such as C, C++, or Java usually contain multiple modules and memory aliasing, which makes them not amenable to automatic parallelization. Furthermore, code whose memory access patterns are indeterminable at compile time due to dependence on program inputs can be impossible for these compilers to parallelize.

Hardware-based speculative parallelism has been proposed to address the problem of parallelizing code that is hard to analyze statically [HWO98, SM98, MBVS97]. In these schemes, code is speculatively executed in parallel and extra hardware is used to detect dependences and to undo the parallel execution in cases of misprediction. The additional hardware required is extensive due to the need for communication between all processors to determine if dependences exist. Softspec uses compiler and runtime techniques that take advantage of underlying properties of programs to parallelize difficult to analyze applications. Softspec succeeds in reducing the communication and computation overhead of speculation and is able to achieve performance improvements without requiring specialized hardware.

Our approach to parallelizing applications stems from two main observations. First, the performance improvements for only partially parallel code do not scale well due to synchronization costs. Second, memory access patterns in loops can often be predicted at runtime using simple value predictors. Softspec performs data dependence analysis at runtime using predicted access patterns. It speculatively parallelizes loops and detects speculation failures without inter-processor communication.

We describe the Softspec technique for parallelizing sequential applications using only simple software mechanisms which can improve performance of modern programs on existing hardware. We present a prototype implementation consisting of a compiler and accompanying runtime system and give experimental results on a symmetric shared-memory multiprocessor.

Since Softspec often targets fine-grain parallelism, we expect even better performance on architectures with lower inter-processor data sharing costs, such as single-chip multiprocessors [HNO97, SM98].

Softspec requires only local program information and does not rely on any global analysis. Its simplicity means it could execute entirely at runtime and target program binaries. Runtime translation [Kla00, CHH+98, ATCL+98] and runtime optimization [BDB00] techniques are becoming prevalent. Softspec can be readily incorporated into such frameworks.

This paper makes the following contributions:

- It is the first application of stride prediction to parallelizing loops.
- It presents a novel approach to speculative parallelism that requires minimal program analysis and no additional hardware.
- It shows how interprocedural parallelism can be exploited without any interprocedural analysis.
- It gives a general technique for parallelizing while loops and is the first technique that is applicable to while loops when the exit condition is not known until the last iteration.
- It gives a general technique for parallelizing sparse matrix algorithms, especially effective for matrices with dense clusters.

This research was supported in part by the Defense Advanced Research Projects Agency under Grant DBT63-96-C-0036 and by a Nippon Telegraph & Telephone graduate fellowship.

The paper is organized as follows. The next section gives an overview of our technique. Section 3 describes the core algorithm in detail, and Section 4 explains how the algorithm handles more complicated loops. Section 5 gives experimental results of our prototype implementation. Related work and conclusions finish the paper.

2 The Softspec Approach

This section gives an overview of the Softspec parallelization technique. Softspec parallelizes loops whose memory references are *stride-predictable.* A memory access is stride-predictable if the address it accesses is incremented by a constant stride for each successive dynamic instance (e.g., in a loop). Array accesses with affine index expressions within loops are always stride-predictable. It has been shown that many other memory accesses are also stride-predictable [SS97].

Rather than proving at compile time that a loop contains no inter-iteration dependences, we calculate the dependences at runtime. First we dynamically profile the addresses in the first three iterations of the loop. If the addresses are stride-predictable in these three iterations, we predict that the addresses have the same strides for the rest of the loop. Once the stride of each memory access has been identified, we determine whether or not there are inter-iteration dependences among the memory accesses. We do not parallelize loops that contain inter-iteration dependences since the synchronization costs can outweigh the benefits of parallelization — we only target loops whose iterations can all be executed in parallel (doall loops).

An inter-iteration dependence exists if a write in one iteration is to the same address as a read or a write in another iteration. In order to determine if any such dependences exist, we examine all memory accesses in the loop pairwise. Each memory access instruction covers a region of addresses throughout the iterations of the loop. For each pair we determine how many iterations may be executed before their regions overlap. If there are R memory reads and W memory writes in the loop, $W * (W + R)$ comparisons are performed. The minimum of all the results is then the number of parallelizable iterations in the loop.

If the number of parallelizable iterations in the loop is large enough, the loop is speculatively executed in parallel, its parallelizable iterations split evenly among the available processors. If there are very few parallelizable iterations and each iteration contains little work, speculation is not attempted and the loop is executed sequentially.

While speculating, each processor checks that the predicted addresses match the actual addresses. This is a local operation and requires no communication with other processors. In fact, the only global communication required is one bit of information at the end of speculation stating whether all predictions were correct or not. If there is a misprediction, all subsequent iterations must have their speculation undone and be re-executed sequentially. An undo buffer is allocated to store the original values at all addresses written in the loop. Since the predicted addresses are guaranteed to have no inter-iteration dependences, speculative memory accesses use the predicted addresses instead of the actual addresses (which may contain dependences). This enables each processor to undo the effects of speculation independently of the other processors. After undoing in parallel, the remaining iterations of the loop are executed sequentially. Alternatively, the speculative process can be restarted — for example, speculation could attempt to parallelize only a piece of a loop at a time and only give up if there are many mispredictions. In this way loops with a few gaps in their stride-predictability or with widely varying iteration counts can be fully parallelized.

```
/*** original code ***/
void foo(double *a, double *b, int j) {
  double *p;
  int i;
  p = &a[j];
  for (i=0; i<500; i++) {
    a[i] = i;
    b[i+j] = b[i] - *p;
  }
}
```

Figure 1: A sample procedure containing a parallelizable loop.

3 The Core Algorithm

This section describes in detail how Softspec parallelizes simple loops with known numbers of iterations whose bodies are straight-line code. This is the core of the Softspec algorithm. Section 4 explains how we extend the core algorithm to handle loop-carried dependences, branches in loop bodies, while loops, nested loops, and interprocedural parallelism.

To illustrate how Softspec works, consider the code in Figure 1. The loop in this procedure contains four memory accesses: two writes (`a[i]` and `b[i+j]`) and two reads (`b[i]` and `*p`). These accesses are stride-predictable with a stride of 0 for `*p` and strides equal to `sizeof(double)` for the others. If `a` and `b` are non-overlapping arrays with lengths at least 500 and `j >= 500`, there will be no inter-iteration dependences between the memory accesses and the entire loop will be parallelizable.

In order for a parallelizing compiler to parallelize this loop, it must prove statically that `a` and `b` are distinct arrays and that there will be no inter-iteration dependences between the memory accesses. Deducing this information at compile time requires sophisticated interprocedural analysis, or may be impossible if the memory addresses are dependent on program inputs. However, a compiler that makes use of runtime predicated parallelism can parallelize this loop by inserting a test that at runtime will deduce if there are memory dependences and only execute the parallel version of the loop if there are none. This example serves only to illustrate the core of the Softspec algorithm; the rest of the algorithm explained in Section 4 targets loops for which practical parallelism-detecting predicates do not exist.

Softspec parallelizes this loop by simply profiling the initial addresses and calculating the intersections of their strides. No complex compile-time analysis is required — all that needs to be done is instrument each memory access.

The Softspec algorithm replaces the loop with four loops: a profile loop, a detection loop, a speculation loop, and a recovery loop. The execution path through these loops is shown in Figure 2. The following sections describe this path in more detail. For further information on the core algorithm see [Dev99].

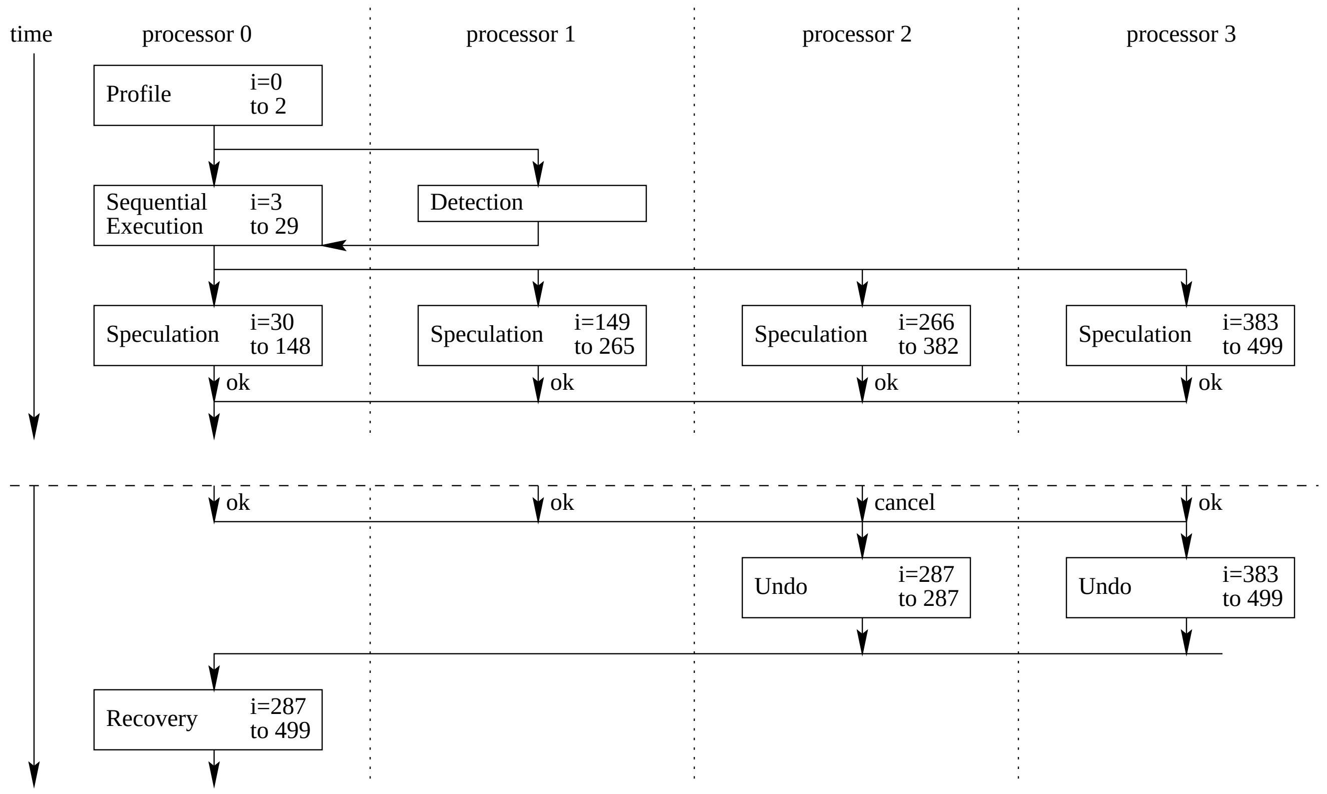

Figure 2: The execution path of Softspec's parallelization of the sample loop in Figure 1. The loop contains 500 iterations, `i=0` through `i=499`. The 27 iterations performed during detection is a typical empirical number for a small loop body. The top portion of the figure shows successful speculation. The bottom portion shows what would happen if a misprediction occurred when `i=287`.

```
/*** profile loop ***/
for (i=0; i<3; i++) {
  profile_address[i][0] = &a[i];
  a[i] = i;
  profile_address[i][1] = &b[i+j];
  profile_address[i][2] = &b[i];
  profile_address[i][3] = p;
  b[i+j] = b[i] - *p;
}
```

Figure 3: The profile loop created from the sequential loop in Figure 1.

3.1 Profiling and Parallelism Detection

The profile loop for the sample code of Figure 1 is shown in Figure 3. As can be seen, it runs the first three iterations of the original loop with instructions inserted to store the addresses of each memory access into data structures used by the runtime system. The outer index of the `profile_address` array corresponds to the iteration of the loop being profiled, and the inner index is used to number the memory accesses.

When the profile loop finishes the runtime system calculates the stride of each memory access by simply taking the difference of addresses in consecutive iterations. If each memory access has a consistent stride (i.e., the strides between the first and second and between the second and third profiled iterations are the same), the runtime system then determines how many iterations can be parallelized before an inter-iteration dependence is encountered. If the strides are not consistent then no speculation is performed and the loop is run sequentially. This catches many accesses that are not stride-predictable early on before any speculation needs to be undone. How Softspec handles accesses that are not executed in all three of the profiling iterations is discussed in Section 4.2.

A second thread performs the stride calculations and parallelism detection while the original thread continues sequential execution of the loop, as shown in Figure 2. This enables forward progress on the loop to be made while the detection calculations are performed (the calculations can take time equal to tens of iterations of loops with small bodies).

To identify whether parallelism exists, we examine the memory accesses pairwise to determine how many iterations may be executed before their addresses become equal. For a pair of addresses with values in the first iteration a_0 and a_1 and with strides s_0 and s_1, we need to find the minimum values of integers i and j such that $a_0 + i * s_0 = a_1 + j * s_1$. Rewriting the equation as $i*s_0 - j*s_1 = a_1 - a_0$, a solution exists if and only if the greatest common divisor (*gcd*) of s_0 and s_1 divides $a_1 - a_0$. The solution can be obtained as a singly parameterized set using Euclid's gcd algorithm [AHU74],

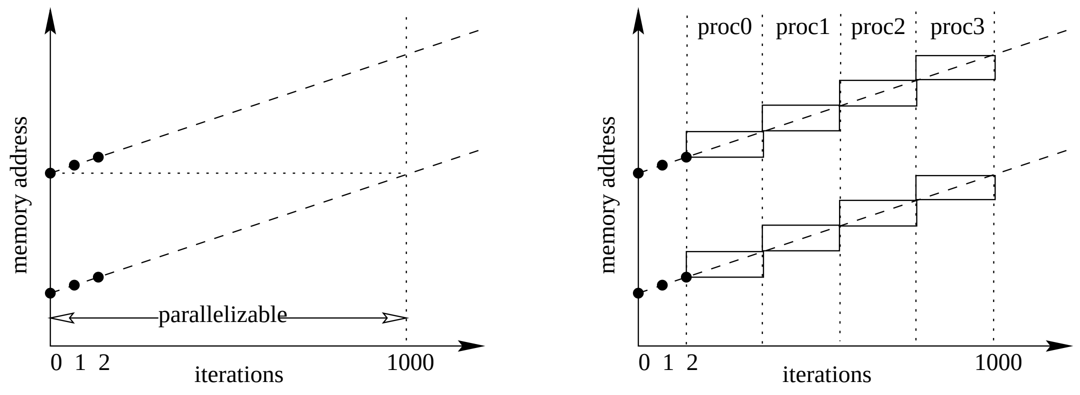

Figure 4: Two memory accesses are compared to determine the number of iterations before they have an inter-iteration dependence. The profiled addresses are shown as dots; their strides are extrapolated to obtain two lines. Inter-iteration dependences may exist starting at the point where one line enters the other's address space. In this case the first access is to a 1000-element array located in memory adjacent to the second access. Thus, the two overlap at 1000 iterations. The parallelizable iterations are divided among four processors as illustrated on the right.

from which the largest number of parallelizable iterations can be calculated.

We found that nearly 90% of the parallelism detection execution time was spent calculating gcd's. To avoid this calculation our prototype Softspec implementation uses an approximation algorithm that calculates a conservative solution but is much more efficient. In practice we have never found a case where the conservative algorithm reports less parallelism than the exact gcd algorithm.

This approximation algorithm views each address as a line defined by the initial address value a_i and the stride s_i: $y = s_i * x + a_i$. For a given pair of addresses, the algorithm extrapolates each profiled address into a line and determines the minimum x value in the first quadrant at which the two lines share y values. This x value is the first iteration at which an inter-iteration dependence can occur. Figure 4 gives an example of this process. The first inter-iteration dependence may actually occur later than the x value computed in this manner, since treating discrete address values as a continuous line considers addresses to overlap that may never actually line up due to equal strides but different offsets. For this reason we add a heuristic case to the approximation algorithm: if the pair of addresses have the same stride and either the same initial value or different initial values modulo the stride then they will never overlap.

The cost of the detection algorithm becomes negligible as the amount of computation in the target loop increases. An adaptive detection algorithm that runs the approximation algorithm for small loops and the full gcd calculations for large loops could be used.

3.2 Speculation and Recovery from Misprediction

If Softspec detects enough parallelism to warrant speculating, the speculative version of the loop is executed by each processor. The speculative version of the sample loop from Figure 1 is shown in Figure 5. It uses the predicted addresses instead of the actual addresses so that only one misprediction conditional at the end of the loop is needed, instead of a conditional before every memory access. The predicted addresses are initialized before the loop by simply adding the stride multiplied by the starting iteration number for the thread to the initial address.

```
/*** speculation loop ***/
for (i=start[thread]; i<stop[thread]; i++) {
  ncancel_flag &= (predict0 == &a[i]);
  *undo_buffer = *predict0;
  undo_buffer++;
  *predict0 = i;
  ncancel_flag &= (predict1 == &b[i+j]);
  ncancel_flag &= (predict2 == &b[i]);
  ncancel_flag &= (predict3 == p);
  *undo_buffer = *predict1;
  undo_buffer++;
  *predict1 = *predict2 - *predict3;
  if (!ncancel_flag) { /* fail */ }
  predict0 += delta0;
  predict1 += delta1;
  predict2 += delta2;
  predict3 += delta3;
}
```

Figure 5: The speculation loop created from the sequential loop in Figure 1.

Code inserted into the loop stores values to be overwritten in the undo buffer and increments each predicted address by its stride. Each processor has its own undo buffer that is allocated prior to speculation (it is made large enough to hold all writes in the loop). Additional code checks that the predicted address matches the actual address; if not, the speculation fails: all writes in iterations during and after the misprediction are undone and those iterations are re-executed sequentially by the recovery loop. The recovery loop is identical to the original sequential loop except that it begins execution on the iteration of the misprediction. This process is illustrated at the bottom of Figure 2. Note that all operations are local to each processor except for the success or failure of the speculation at the end of the loop; no other global communication is required.

Undoing writes involves restoring the values from the

Loop	Per Read	Per Write	Per Iteration
Profile	1-4	1-4	0
Detect	0	0	2-4
Speculate	6-11	10-13	2-5
Recovery	0	0	0

Table 1: Overhead in terms of machine instructions for the four loops that Softspec creates when parallelizing a target loop.

undo buffer to the predicted addresses, from the most recently executed iteration backward to the earliest executed iteration. Since the predicted addresses are stride-predictable, the undo buffer does not need to store any memory addresses, only values. The processors can undo in parallel since there are no overlaps in the predicted addresses.

If many execution instances of a loop experience early speculation failures, the runtime system disables speculation of that loop and executes it sequentially instead to avoid overhead costs.

3.3 Runtime Overhead

The Softspec algorithm transforms the original sequential loop into four loops. One of these, the recovery loop, is essentially identical to the original loop. The other three loops have extra instructions added that lead to runtime overhead when compared to the original sequential loop.

The profile loop stores the address of each load and store into a shared data structure. The detection loop, which sequentially makes forward progress on the loop while the runtime system detects the amount of parallelism in the loop, is equivalent to the original loop with a check every iteration to see if the parallelism detection has finished. Finally, the speculation loop contains an increment of the predicted address and a check that the predicted address equals the actual address for all memory reads and writes, a store to the write buffer and increment of the write buffer pointer for each write, and additionally a single speculation failure branch. Table 1 summarizes these costs in units of machine instructions for typical compilations.

Note that since speculation uses predicted memory addresses, nested array references (such as `A[B[i]]`) or multiple levels of pointer indirection (such as `**p`) do not need to wait for the first memory reference to resolve before accessing the second. Breaking this dependence allows for more instruction-level parallelism and reordering.

Additional overhead is required to synchronize the threads. A barrier is required at the end of the parallel execution to determine if any threads encountered mispredictions. This barrier may cost hundreds or even thousands of cycles on a shared-memory multiprocessor. Fortunately it is the only global synchronization needed by Softspec.

Very simple extensions to the underlying hardware, such as adding a speculative bit to caches that eliminates the need for an undo buffer, could reduce Softspec's overhead significantly.

4 Extending the Core Algorithm

The core of the Softspec algorithm described in Section 3 only handles loops with known numbers of iterations whose bodies are straight-line code, i.e., no procedure calls or branches inside of loops, and no while loops or nested loops. This section explains how Softspec extends the core algorithm to handle loop bodies containing loop-carried dependences and nonlinear control flow, while loops, nested loops, and interprocedural parallelism. We have incorporated all but the last of these into our prototype Softspec implementation. We plan to implement interprocedural parallelism in the near future; we describe here how to do so without using any interprocedural analysis. We then discuss using compiler information to reduce runtime overhead, and compare the runtime overhead of these extensions to that of the core algorithm.

4.1 Loops Containing Loop-Carried Dependences

Memory addresses are not the only values in a loop that are often stride-predictable. Scalar variables with loop-carried dependences exhibiting stride-predictability range from simple induction variables, which are usually analyzable at compile time, to pointers that are difficult to analyze at compile time. A pointer used to traverse a linked list, when the list is laid out contiguously in memory, is a good example of a stride-predictable loop-carried dependence.

Loop-carried dependences are treated in a similar manner to memory addresses. Their values are profiled in the first three iterations of the loop just like memory addresses, and a stride is calculated that is predicted to hold for the rest of the loop. If the stride does hold, the loop is parallelizable; no inter-iteration dependence analysis is needed since the value of the variable can be computed independently for any iteration. This is in contrast to memory addresses, for which merely being stride-predictable is not sufficient for parallelism to exist since the addresses and not the values at those addresses are being predicted, and the values may depend on each other.

During speculation the "actual" value of the loop-carried dependence is the predicted value for the current iteration. At the end of the iteration, after this actual value is modified in the loop body, its resulting value is compared to the predicted value for the next iteration and if a misprediction occurs the speculation aborts. This is different from a memory address whose actual value is computed independently of the predicted value.

The undo mechanism needs no extra information to be able to restore loop-carried dependences to the values they held prior to a misprediction. Each value can be computed using the profiling data for the iteration prior to the speculation failure.

If a loop-carried dependence is used as a pointer, care must be taken to avoid a misprediction causing a memory access outside of the program's address space. The predicted values of the pointer for the initial and final iterations of the loop need to be checked to ensure that they are within the application's address space. If they are, then so are all values at intermediate iterations. Within the address space, any misprediction will be detected and all erroneous writes will be restored to their original values from the undo buffer.

4.2 Loops Containing Nonlinear Control Flow

The core algorithm assumed that each memory access would be executed on each iteration. Branches inside the loop body often fail to meet this assumption. This can cause addresses to be encountered during speculation that were not executed during profiling. Speculation cannot proceed in such a case since no predicted stride is available and no inter-iteration dependence analysis has been performed on the new address. Speculation must also abort if a branch is taken that leads out of the loop.

If a particular access is not executed in any of the three profiling iterations, that access must be distinguishable from fully profiled accesses by the speculation loop. A sentinel in the value of the predicted address is used for this purpose. Unfortunately, the speculation loop must branch immediately based on its check for the sentinel because it cannot execute instructions using an unknown address. Such a speculation failure branch must be placed before every address use that could potentially be unprofiled. This branch causes two problems. First, it degrades performance. This can be avoided in some cases by generating a version of the speculation loop without the branches and using that version when all addresses are executed in the profile loop. Second, it requires the ability to undo a partially finished iteration. This is easily solved: when a misprediction occurs the address it occurred at is recorded, and the undo mechanism uses that information to undo just the portion of the iteration prior to the misprediction.

If an access is executed in only one of the three profiling iterations, a stride cannot be calculated. However, a stride can be guessed based on the strides of other addresses in the loop or possibly on stored strides from previous instances of this loop. Our prototype implementation does not currently guess strides and does not attempt to speculate a loop containing an address that was profiled only once.

If an access is executed in two of the three profiling iterations, a stride can be calculated and speculated just like a fully profiled address. Alternatively, a more dynamic approach could decide to execute further profiling iterations to verify the stride.

Detecting unprofiled loop-carried dependences is done in the same manner as for memory addresses, but since loop-carried dependences can take on any value a sentinel cannot be used. Instead, the profiling data structures are extended.

4.3 While Loops

While loops with un-analyzable exits are difficult to parallelize since the number of iterations in the loop is not known at compile time. In fact, the number of iterations is not known until the final iteration of the loop. The only way to parallelize such a loop is speculatively. The number of iterations must be guessed and overshooting (guessing more iterations than actually exist) must be handled.

The core algorithm is easily extended to handle while loops. Guessing the number of iterations is accomplished in one of several ways. It can be guessed outright. The loop can be run sequentially the first time it is encountered and the number of iterations from that execution can be used as the guess for the next execution. Or, repeated speculation can be used: a small number of iterations is repeatedly speculated until the loop exit condition is reached. Repeated speculation is a good strategy for while loops that may have a few stride-predictability gaps but for the most part are stride-predictable, and for while loops whose iteration count varies widely. In any case, later speculation of the loop is made adaptive by keeping a running average of the number of iterations in previous executions of the loop and using it to speculate future instances of the loop.

A while loop is converted into a for loop that executes the guessed number of iterations and breaks if the while loop exit condition is met before the for loop finishes. The profile, detection, and speculation loops are generated from this for loop. The recovery loop is the original while loop.

If the while condition is overshot, i.e., too many iterations are executed, the extra iterations are undone and the running average of the number of iterations is updated. If the speculation ends successfully and the while condition has not been reached, speculation can begin again on the remainder of the loop, or else the recovery loop finishes executing the loop. Address values in the speculation loop must be checked to ensure they are within the address space of the program (predicted values in iterations beyond the actual loop end could be outside of the address space). This can be done by checking the initial and final predicted values prior to the speculation loop.

4.4 Nested Loops

The core algorithm only handles innermost loops. Even when parallelism exists in inner loops, greater performance improvements can be achieved by parallelizing outer loops, especially when executing on machines with high data sharing costs between processors' caches where a higher computation to overhead ratio is required. However, targeting outer loops complicates parallelism detection because there are many more opportunities for inter-iteration dependences. Softspec extends the core algorithm to handle outer loops without unduly increasing its complexity.

In a nested loop, the inner loops can be treated in two ways. The inner loops' memory accesses can be required to be stride-predictable (the linear method), or they can be required to fall within a certain range (the range method). The two methods are handled similarly by Softspec. We implemented both methods and found that, as expected, the range method is less efficient than the linear method. Although the range method places fewer restrictions on which loops are parallelizable, we never encountered a loop for which the range method applied but the linear method did not. Our prototype implementation uses the linear method.

The profiling step for nested loops runs the outer loop for three iterations and the inner loops for their full number of iterations. The linear method only gathers profiling data on the first three iterations of the inner loops, while the range method keeps track of the maximum and minimum values of each address accessed in each inner loop. For the linear method, all memory accesses in a loop have a stride that holds within the outer loop; a memory access in an inner loop has a separate stride for that loop. Both strides are necessary to predict addresses that are accessed in both the inner and outer loop. To handle addresses that are accessed in more than two layers of loops, additional separate strides must be calculated.

Note that loop-carried dependences in nested loops can be treated just like in non-nested loops. Only a single stride is required for each loop-carried dependence since it does not matter what happens to its value inside inner loops; all that matters is that its value at the end of each iteration of the outer loop is stride-predictable. That is the only value that is profiled for both the linear and range methods.

In the detection step, the linear method calculates inner and outer strides and from them calculates a range of values for each memory access based on the number of iterations in the inner loops, while the range method calculates this range from the minimum and maximum profiled values. Each range has a stride, just like a single memory access in the outer iteration has a stride. The detection algorithm extrapolates and compares the ranges in the same manner as the core algorithm compares lines to calculate the number of parallelizable iterations.

Speculation and undoing for the linear method are similar to non-nested loops. For the range method, since its addresses are not assumed to be stride-predictable, either the undo buffer must store the address of each write whose old value it stores, or the entire range of values overwritten in an inner loop must be copied to the undo buffer at once. Also, the range method uses actual addresses instead of predicted addresses in its inner loops, and so it must check before each memory access that the actual address is within the predicted range. These two factors combine to make the range method less efficient than the linear method.

The greatest challenge for Softspec in parallelizing nested loops is determining which loop to parallelize in multiply-nested loops. Compiler analysis can sometimes find the outermost parallelizable loop; a dynamic solution attempts parallelization of the outermost loop and if it fails tries each nested inner loop in turn.

4.5 Interprocedural Parallelism

In a purely runtime implementation, procedure calls are not an obstacle to parallelization with Softspec since the program is viewed at the execution trace level and procedures are essentially inlined.

Extending a compiler-based Softspec implementation to handle procedure calls in loop bodies can be done without any interprocedural analysis. Separate versions of each procedure for profiling and speculating can be generated independently. The memory references are then dynamically numbered. Libraries must have their procedures made "Softspec-ready". Runtime analysis will view the loop from the level of its trace, treating the procedures as though they were inlined, and not need any interprocedural information. A flag must be used to prevent nested speculation from loops inside of procedure calls.

This is markedly simpler than any other treatment of procedure calls in loops. Parallelizing compilers must perform complex interprocedural analysis to determine if procedure calls may be parallelized. Softspec enables a very simple mechanism for interprocedural parallelism. We have not yet incorporated this into our prototype implementation of Softspec, but plan to do so in the near future.

4.6 Using Advanced Compiler Analysis

For many accesses in a loop, the compiler may be able to prove that the accesses have a given pattern. For example, the access pattern of a local induction variable whose address is not taken can be easily discerned. These variables do not require any profiling or runtime checking. Also, the loop's dependence analysis can be partially evaluated at compile time.

The overhead of the undo buffer can be reduced or eliminated using compile-time information. If a memory reference is read before it is written, the read value can be directly written to the undo buffer, eliminating a load instruction. Furthermore, if the compiler can deduce how to re-create the original value of a modified memory access, no undo information for that access needs to be stored.

Incorporating Softspec into a parallelizing compiler will make the compiler much more robust. A loop with a few unanalyzable accesses will still be parallelizable with a little extra runtime overhead.

4.7 Runtime Overhead of Extended Algorithm

The extensions to the core algorithm described in this section add additional runtime overhead to that analyzed in Section 3.3. Loop-carried dependences do not affect the overhead much, since even in a nested loop they are only checked at the end of the outer loop. Nonlinear control flow, however, adds failure branches to the middle of the speculation loop, which can be relatively expensive. The only extra overheads of while loops come from overshooting the number of iterations and the added overhead in repeated speculation. Finally, nested loops using the linear method do not have appreciably more runtime costs than non-nested loops, while the range method adds costs in the form of failure branches and copying large regions to the undo buffer at once.

5 Experimental Results

We have developed a prototype implementation of the Softspec algorithm consisting of a compiler and accompanying runtime system. The prototype incorporates all of the algorithm components described in Section 4 except for interprocedural parallelism. The compiler transforms target sequential code into speculatively parallel code which is then linked with the runtime system. The compiler was written in SUIF [AALT95] and the runtime system was written in C. The pthreads library is used to create a separate thread for each processor on the machine; these threads are created at program startup and are used for all speculation in the program.

This section presents the results of using our prototype to parallelize various types of applications. The target applications were run on a Digital AlphaServer 8400, which is a bus-based shared-memory multiprocessor containing eight Digital Alpha processors. We expect even better performance improvements on architectures with lower costs of data sharing between processors' caches, such as single-chip multiprocessors [HNO97, SM98].

Note that there is nothing about the Softspec technique that requires a source-code compiler. It could operate on program binaries, and could execute completely at runtime in a dynamic optimization framework. If the speedups presented here can be achieved in such environments, they will be even more impressive. The simplicity of the Softspec approach lends it versatility.

5.1 Dense Matrix Applications

We first examine how Softspec performs on dense matrix applications, which are typically highly parallelizable by current compilers. Obviously we do not expect Softspec to produce greater speedups than automatically parallelizing compilers because of our runtime overhead. This section gives an idea of how Softspec compares to automatically parallelizing compilers.

Consider the matrix multiplication code in Figure 6. The middle loop is parallelizable, and Softspec parallelizes it successfully with speedup shown in Figure 7.

```
for(i=0; i<L; i++) {
  for(j=0; j<M; j++) {
    for(k=0; k<N; k++) {
      c[i][k] += a[i][j] * b[j][k];
    }
  }
}
```

Figure 6: Dense matrix multiplication loop nest.

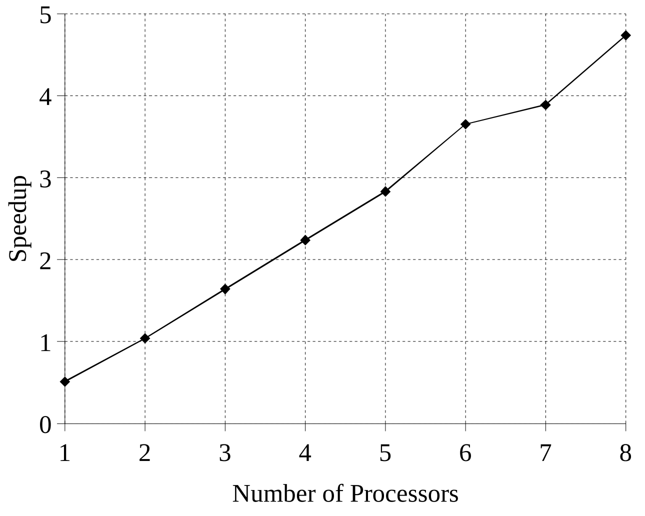

Figure 7: Speedup obtained by Softspec for multiplication of a dense 1000 by 1000 square matrix.

When executed on one processor, speculation has a nearly two times slowdown. The sequential program's loop body is very simple and more easily optimized by the compiler than the parallel version of the code generated by Softspec. The sparse matrix code in Section 5.4 is more difficult to optimize and thus exhibits higher speedups and much lower slowdown on one processor. On two processors the dense matrix multiplication breaks even, and its speedup increases linearly by a little more than 0.5 with each processor added.

Figure 8 gives Softspec's speedup when applied to the SPEC95FP benchmark swim. The results are similar to the matrix multiplication results.

5.2 While Loops with Un-analyzable Exits

Automatically parallelizing compilers naturally cannot parallelize loops whose termination conditions are un-analyzable until the last iteration at runtime. Softspec's dynamic techniques can parallelize such loops. Our prototype implementation of Softspec executes a while loop sequentially the first time it is encountered and uses the number of iterations in that first execution as the number of iterations to speculate on. The iteration count is updated as a running average to adapt to varying-length while loops.

Consider the example code in Figure 9, a procedure for performing convolution in which one array's end is marked with a sentinel. In a program that calls this procedure multiple times, Softspec obtains the speedup shown in Figure 10.

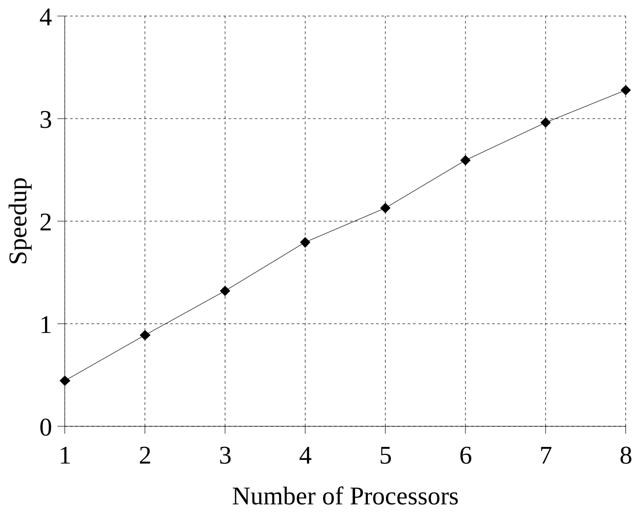

Figure 8: Speedup for the SPEC95FP benchmark swim.

```
void convolve(double *A, int lengthA, double *B,
  double sentinel, double *C) {
  int i,j;
  i = lengthA;
  while (B[i] != sentinel) {
    double tmp = 0;
    for (j=0; j<lengthA; j++) {
      tmp += A[j] * B[i-j];
    }
    C[i] = tmp;
    i++;
  }
}
```

Figure 9: Example code containing a loop whose exit condition is not analyzable at compile time. The code performs a convolution in which the end of array B is marked with a sentinel.

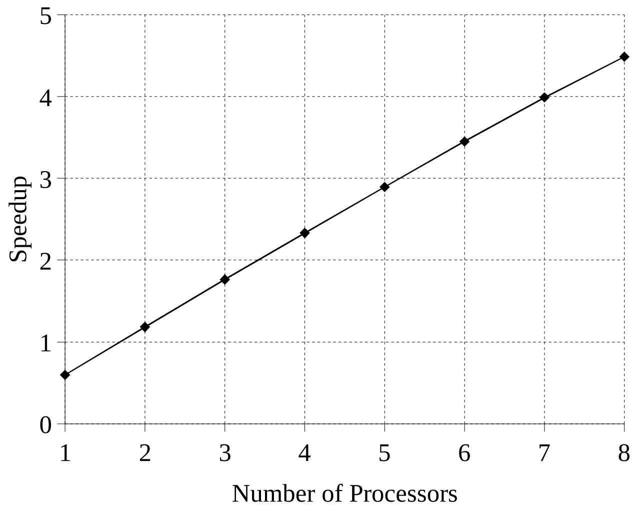

Figure 10: Speedup for a program that makes 100 calls to the convolve procedure of Figure 9, each with lengthA=1000 and length of B = 20000.

```
void compute(node *link) {
  while (link != NULL) {
    /* do work on link */
    ...
    /* move to next link */
    link = link->next;
 }
}
```

Figure 11: Code traversing a linked list and performing some computation at each node.

5.3 Linked List Traversal

Code that traverses a linked list can be difficult to impossible to parallelize. If the nodes of the list are all laid out contiguously then the traversal is amenable to parallelization by Softspec. In a system with a garbage collector, the memory layout of the list can be controlled. The garbage collector can cooperate with Softspec by keeping the list laid out contiguously as much as possible. A Java virtual machine, for example, could implement Softspec dynamically and use its control of memory layout to parallelize many loops otherwise not parallelizable.

Without control over the memory layout, a repeated speculation strategy can be used to obtain speedup on the regions of the list that happen to be contiguous. We investigated performance on lists with varying memory layouts. In practice lists often have clusters of noncontiguous nodes. To model this, we allocate a 20,000 node list such that each sequence of 50 consecutive nodes has a certain chance of either being contiguous or having frequent gaps between the nodes. We parallelized a program that traverses this list and performs some computation at each node. The code framework is shown in Figure 11.

Performance results for this program are given for different chances of each sequence of the list containing gaps and for different numbers of iterations to speculate (repeatedly). Results for a 0% chance of each sequence having gaps (a completely contiguous list) are shown in Figure 12, for a 1% chance in Figure 13, and for a 5% chance in Figure 14. Each figure shows the speedup for five different numbers of iterations speculated: "All" performs speculation of the entire loop and does not try again after reaching a misprediction while the other four, "100", "200", "400", and "800", repeatedly speculate that many iterations at a time until the loop finishes, regardless of how many speculations fail due to mispredictions.

5.4 Sparse Matrix Applications

In this section we give an example of a sparse matrix application that is for some matrices stride-predictable and contains parallelism, but is not parallelizable by current compilers. The FORTRAN code in Figure 15 is the core code for matrix multiplication of sparse matrices stored in compressed row storage format. The many arrays and multiple levels of indirection make analysis of this loop too difficult for current parallelizing compilers, but Softspec's parallelization scheme is applicable.

When the current row in each of the two matrices being multiplied contains contiguous regions of non-zero values, the memory references in each of the two branches of the

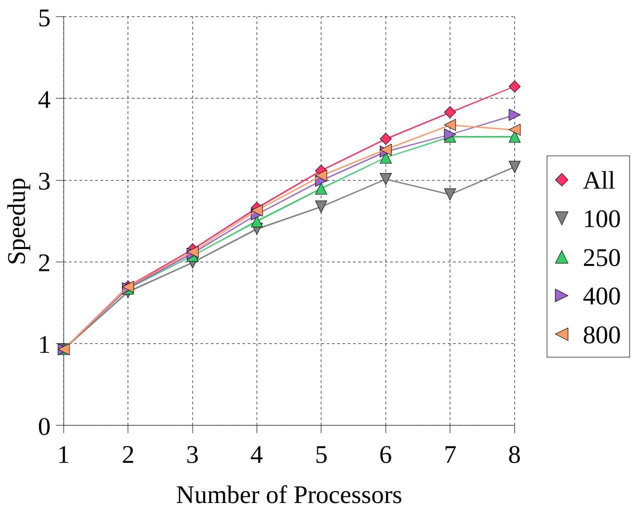

Figure 12: Speedup for the sample program in Figure 11 operating on a completely contiguous list. Softspec either used repeated speculation of a certain number of iterations (100, 250, 400, 800) or speculated the entire loop ("All").

`if` are stride-predictable. Thus when many consecutive iterations of the inner loop take the same branch, the inner loop is parallelizable by Softspec. The `then` branch initializes new non-zero elements in the product matrix while the `else` branch adds to an existing element. The row by row multiplication ends up often entering the same branch on consecutive iterations of the inner loop. The frequency and duration of parallelizable iteration sequences are dependent on the input matrices. Sparse matrix data sets such as the Non-Hermitian Eigenvalue Problem Collection [NHE] often contain matrices with non-zero values in a stripe down the diagonal or in blocks down the diagonal. Such patterns lead to parallelizable iteration sequences of lengths equal to the width of the stripes or blocks.

Figure 16 shows the speedup obtained by Softspec when multiplying block diagonal sparse matrices with different block sizes. Six different sparse matrices with non-zero elements in square blocks down the diagonal were multiplied by themselves. The block widths for the six matrices were 25, 50, 100, 200, 300, and 400, respectively. The number of blocks does not affect the speedup much, as it only changes the total run time and not the length of the parallelizable sequences.

No speedup (or slowdown) occurs for parallelizable sequences of 25 or 50 iterations. For 100 iterations moderate speedup is achieved, but there is not enough work for the speedup to scale beyond five or six processors. Iteration counts of 200, 300, and 400 all achieve scalable speedup.

The speedups obtained on a few hundred iterations of the inner loop of the sparse matrix multiplication are greater than those for the nested loop of the dense matrix multiplication, even though the latter has much more work in each parallelized iteration. The reason stems from the ability of the compiler to more aggressively optimize the dense matrix loop body versus the sparse matrix loop body. The multiple array indirections inhibit optimizations that can be applied to the simpler dense matrix code. This results in less of a disparity in speed between the sequential code and the

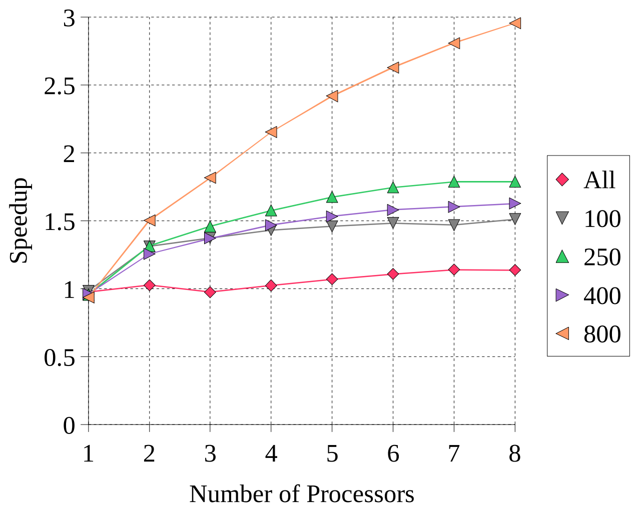

Figure 13: Speedup for the sample program in Figure 11 operating on a list with each sequence of 50 nodes having a 1% chance of containing many memory gaps (a 50% chance of a gap between each node). Softspec either used repeated speculation of a certain number of iterations (100, 250, 400, 800) or speculated the entire loop ("All").

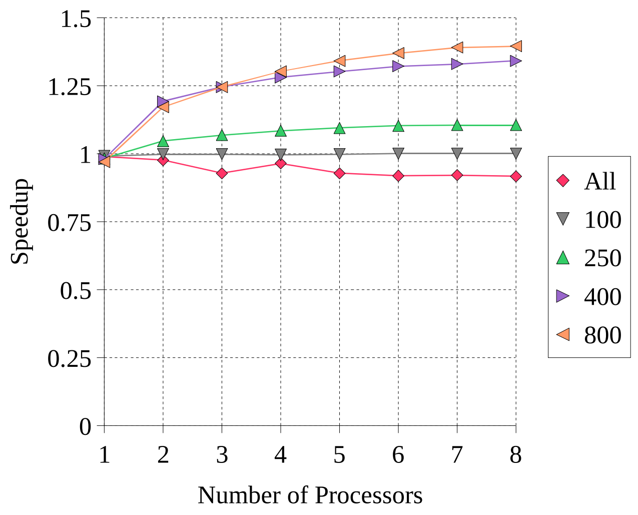

Figure 14: Speedup for the sample program in Figure 11 operating on a list with each sequence of 50 nodes having a 5% chance of containing many memory gaps (a 50% chance of a gap between each node). Softspec either used repeated speculation of a certain number of iterations (100, 250, 400, 800) or speculated the entire loop ("All").

```
do j = offa(i) , offa(i+1)-1
  do k = offb(ja(j)), offb(ja(j)+1)-1
    if ((ptr(jb(k)) .eq. 0)) then
      ptr(jb(k)) = i
      c(ptr(jb(k))) = a(j)*b(k)
      index(ptr(jb(k))) = jb(k)
      i = i + 1
    else
      c(ptr(jb(k))) =  c(ptr(jb(k)))+a(j)*b(k)
    endif
  enddo
enddo
```

Figure 15: The core FORTRAN loop for multiplying sparse matrices `a` and `b` and storing the result in `c`. The sparse matrices are stored in compressed row storage format. The arrays `offa`, `ja`, `offb`, and `jb` are used to locate the columns and rows of elements in `a` and `b`.

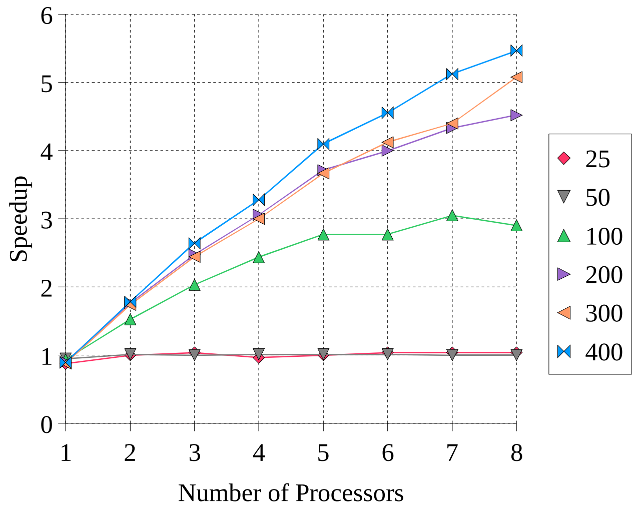

Figure 16: Speedup for the sparse matrix multiplication code of Figure 15 operating on six different block diagonal matrices. The six matrices have blocks of widths 25, 50, 100, 200, 300, and 400, respectively.

parallelized code generated by Softspec.

6 Related Work

Over the last two decades, compiler techniques for automatically parallelizing sequential applications have advanced remarkably. Modern parallelizing compilers are very successful when targeting certain types of code, namely nested loops containing limited memory aliasing. However, these compilers are large systems that use complex interprocedural analyses: the Polaris compiler [BEH$^+$94] contains over 170,000 lines of code, and the SUIF compiler [AALT95] contains over 150,000 lines of code.

Alias analysis algorithms [WL95, EGH94, RR98] can enable automatic parallelization in the presence of memory aliases. However, these alias analysis systems work only with small, self-contained programs and do not scale with program size.

Hardware-based speculative parallelization has been proposed to overcome the need for a compile-time proof that code is parallelizable. Candidate loops are speculatively executed in parallel while a complex hardware system observes all memory accesses in order to detect inter-iteration data dependences. When a dependence is detected, additional hardware mechanisms undo the speculative execution and the loop is re-executed sequentially. Some proposals extend existing multiprocessor hardware [HWO98, SM98] while others present completely new hardware structures [MBVS97]. Targeting procedural parallelism using hardware has also been proposed [OHL99]. Because speculative hardware schemes do not rely an program characteristics they require a lot of work and global communication. Softspec takes advantage of memory access patterns to reduce the amount of work and communication needed to detect dependences, making it viable in software.

Fundamental research into program behavior has shown that both data and address values can be predicted by stride prediction and last-value prediction [SS97]. Stride-predictability of memory accesses in scientific applications has been successfully exploited to improve the cache behavior of these codes through compiler-inserted prefetching in uniprocessor and multiprocessor machines [FP91]. The stride-predictability of memory addresses has been used to perform speculative prefetching in out-of-order superscalars [GG97].

Software-based speculative parallelization schemes have mostly focused on determining the inter-iteration dependence patterns of a partially parallel loop in order to construct parallelization schedules. These schemes focus on an inspector-executor model [LZ93], in which an extracted inspector loop analyzes the memory accesses at run time and constructs a schedule for the executor loop, which runs parts of the loop in parallel using synchronization.

Other software-based techniques speculatively run code in parallel and monitor all memory accesses using shadow arrays [RP95a]. When an inter-iteration data dependence is observed, the speculation is undone. This is similar to the hardware-based proposals, but with greater runtime overhead and only targeting for loops that operate on arrays. Mechanisms for parallelizing certain types of while loops have been developed [RP95b], but they require the help of compiler analysis and code reorganization that could not practically be done at runtime.

7 Conclusions

We have presented a novel approach to parallelization of sequential code that does not require complex program analysis or additional hardware mechanisms. We have demonstrated the benefits of this approach on a variety of programs using a prototype implementation.

The trend towards object-oriented programming and shared library usage is making it increasingly difficult for an automatically parallelizing compiler to perform the whole-program analysis it needs to identify parallelism. We have shown how to exploit interprocedural parallelism with no need for interprocedural analysis, leading to a highly scalable solution that supports separate compilation. We plan to incorporate this into our prototype implementation of Softspec.

Softspec can be combined with existing automatically parallelizing compilers to increase the range of loops they can parallelize. A Softspec-enhanced parallelizing compiler would provide more robust performance.

We plan to simulate the performance of our implementation on proposed single-chip multiprocessors. A factor limiting the speedups obtainable on shared-memory multiprocessors is the high cost of data sharing between processors' caches. These costs are much lower on proposed single-chip multiprocessors. Although we showed speedup on existing hardware, simple hardware extensions such as adding a speculative bit to caches that eliminates undo overhead can drastically improve Softspec performance.

Softspec can be implemented entirely in software and can target program binaries. In the future we hope to integrate Softspec into a virtual machine environment with control of the garbage collector to enhance data structure stride-predictability. We would also like to investigate incorporating Softspec into a binary optimizer. This would enable parallelization of legacy code or third-party software that is only available in binary form and cannot be recompiled.

Softspec introduces a framework for parallelization based on prediction rather than proof of parallelism that enables parallelization of a large class of important applications that are currently unable to use automatic parallelization techniques. We believe that the Softspec framework will lead to many other techniques involving different optimization and prediction schemes.

References

[AALT95] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. The SUIF compiler for scalable parallel machines. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, Reading, MA., 1974.

[ATCL$^+$98] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *Proceedings of the SIGPLAN'98 Conference on Programming Language Design and Implementation*, June 1998.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent runtime optimization system. In *Proceedings of the SIGPLAN'00 Conference on Programming Language Design and Implementation*, June 2000.

[BEH$^+$94] William Blume, Rudolf Eigenmann, Jay Hoeflinger, David Padua, Paul Petersen, Lawrence Rauchwerger, and Peng Tu. Automatic detection of parallelism: A grand challenge for high-performance computing. *IEEE Parallel and Distributed Technology*, 2(3):37–47, Fall 1994. Also `http://polaris.cs.uiuc.edu/polaris/`.

[CHH$^+$98] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye,

S. Bharadwaj Yadavalli, and John Yates. FX!32: A profile-directed binary translator. *IEEE Micro*, 18(2), March 1998.

[Dev99] Srikrishna Devabhaktuni. Softspec: Software-based speculative parallelism via stride prediction. Master's thesis, M.I.T., 1999. `http://www.cag.lcs.mit.edu/~saman/student_thesis/Sri-99.ps`.

[EGH94] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, June 1994.

[FP91] J. W. C. Fu and J. H. Patel. Data prefetching in multiprocessor vector cache memories. In *The 18th Annual International Symposium on Computer Architecture*, May 1991.

[GG97] J. Gonzalez and A. Gonzalez. Speculative execution via address prediction and data prefetching. In *11th International Conference on Supercomputing*, July 1997.

[HNO97] Lance Hammond, Basem Nayfeh, and Kunle Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9), September 1997.

[HWO98] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[Kla00] Alexander Klaiber. The technology behind Crusoe processors. Transmeta Corporation, January 2000. `http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf`.

[LZ93] Shun-Tak Leung and John Zahorjan. Improving the performance of runtime parallelization. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[MBVS97] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependencies. In *24th International Symposium on Computer Architecture*, June 1997.

[NHE] Non-Hermitian eigenvalue problem collection. `http://math.nist.gov/MatrixMarket/data/NEP`.

[OHL99] J. Oplinger, D. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, October 1999.

[RP95a] Lawrence Rauchwerger and David Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the SIGPLAN'95 Conference on Programming Language Design and Implementation*, June 1995.

[RP95b] Lawrence Rauchwerger and David Padua. Parallelizing while loops for multiprocessor systems. In *Proceedings of the 9th International Parallel Processing Symposium*, April 1995.

[RR98] R. Rugina and M. Rinard. Span: A shape and pointer analysis package. Technical Report LCS-TM-581, M.I.T., June 1998.

[SM98] 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*, February 1998.

[SS97] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997.

[WL95] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, June 1995.