Softspec: Software-based Speculative Parallelism

Derek Bruening, Srikrishna Devabhaktuni, Saman Amarasinghe
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{iye,chinnama,saman}@lcs.mit.edu
http://cag.lcs.mit.edu/commit

Abstract

We present Softspec, a technique for parallelizing sequential applications using a hybrid compile-time and run-time technique. Softspec parallelizes loops whose memory references are stride-predictable. By detecting and speculatively executing potential parallelism at runtime Softspec eliminates the need for complex program analysis required by parallelizing compilers. By using runtime information 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 software 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.

Our approach to parallelizing applications stems from the observation that 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 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.

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 [1, 8, 14] and runtime optimization [4] techniques are becoming prevalent. Softspec can be readily incorporated into such frameworks.

The paper is organized as follows. The next section gives an overview of our technique. Section 3 describes the core algorithm in detail. Section 5 gives experimental results of our prototype implementation. Related work and conclusions finish the paper. Further details, including extensions to the core algorithm, can be found in [6].

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 [26].

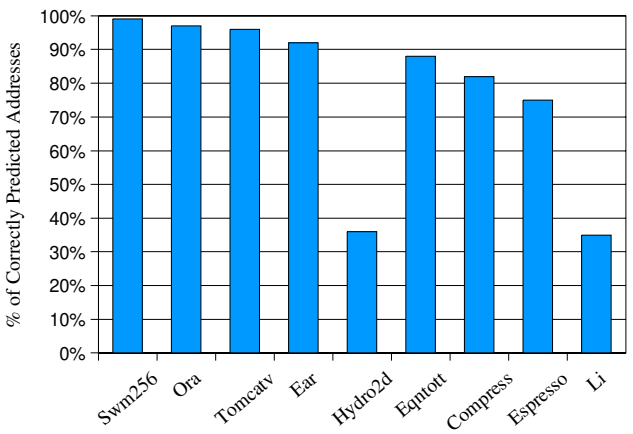

Figure 1: Stride predictability of SPEC92 benchmarks

Figure 1 shows the results of using the ATOM [27] profiling tool to measure stride-predictability in several SPEC92 benchmarks. The figure shows the percentage of dynamic memory references that could be predicted from the previous two dynamic instances of the same memory reference. Stride-predictability in the floating point benchmarks (the first five in the figure) are to be expected: such programs' loops often contain affine index expressions for loads and stores. The results for the integer applications (the final four in the figure) further attest to the prevalence of stride-predictability.

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.

3 The Algorithm

Softspec parallelizes a loop by simply profiling its 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.

To illustrate how Softspec works, consider the code in Figure 2. 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.

```
/*** 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 2: A sample procedure containing a parallelizable loop.

```
/*** 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 4: The profile loop created from the sequential loop in Figure 2.

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 extensions of the algorithm described in detail in [6] allow Softspec to target loops for which practical parallelism-detecting predicates do not exist.

The Softspec algorithm replaces the original 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 3. The following sections describe this path in more detail.

3.1 Profiling and Parallelism Detection

The profile loop for the sample code of Figure 2 is shown in Figure 4. 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.

A second thread performs the stride calculations and parallelism detection while the original thread continues sequential execution of the loop, as shown in Figure 3. This enables forward progress on the loop to be made while the detection calculations are performed.

To identify whether parallelism exists, we examine the memory accesses pairwise to determine how many iterations may be executed before their addresses in different iterations 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 [2], 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 ray 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 ray with a starting point on the y-axis. It then determines, out of all locations where a horizontal line intersects both rays, the minimum x value of

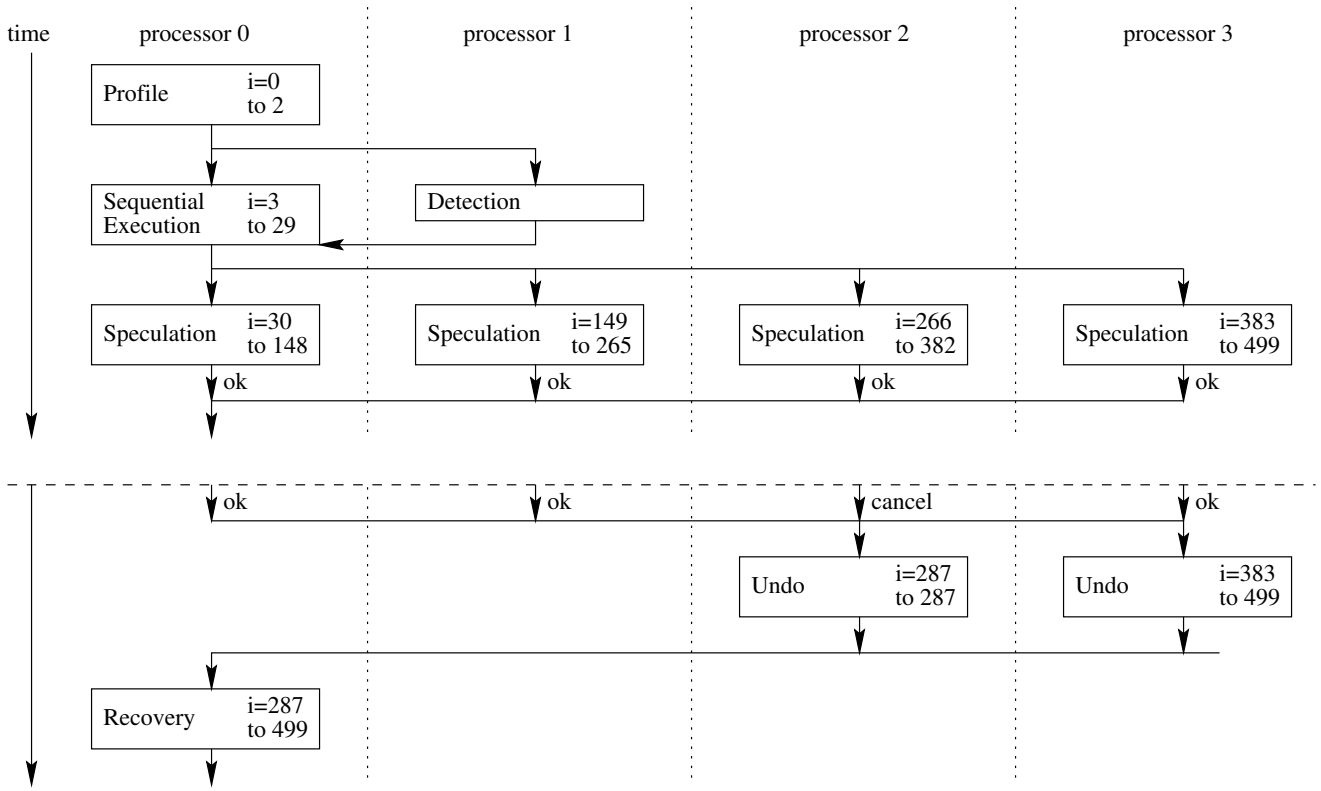

Figure 3: The execution path of Softspec's parallelization of the sample loop in Figure 2. The loop contains 500 iterations, `i=0` through `i=499`. The 27 iterations performed during detection is a typical empirical number for a small loop body. The top portion of the figure shows successful speculation. The bottom portion shows what would happen if a misprediction occurred when `i=287`.

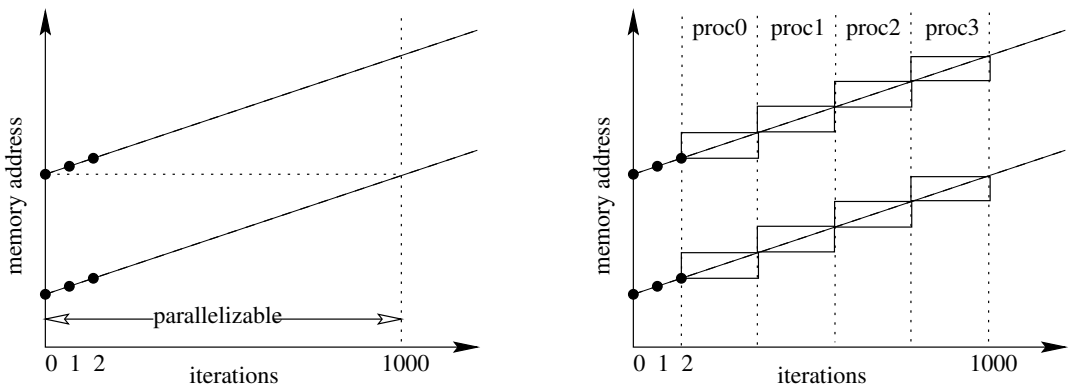

Figure 5: Two memory accesses are compared to determine the number of iterations before they have an inter-iteration dependence. The profiled addresses are shown as dots; their strides are extrapolated to obtain two rays. Inter-iteration dependences may exist starting at the point where one ray enters 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.

the rightmost intersection points. This x value is the first iteration at which an inter-iteration dependence can occur. Figure 5 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 (i.e., both reference the same address on the same iteration, which is fine) or different initial values modulo the stride then they will never overlap. For further information on using gcd algorithms versus our approximation algorithm see [9].

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 iterations of the loop are assigned to processors in a block-cyclic manner.

The speculative version of the sample loop from Figure 2 is shown in Figure 6. It uses the predicted addresses instead of the actual addresses so that the processors can undo their speculation in parallel in case of a misprediction. 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.

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. Additional code checks that the predicted address matches the actual address and stores the result in a flag. This flag can be checked at the end of each iteration (or even after the entire loop, but then there would be no information on which iteration failed) since the use of predicted addresses guarantees that no inter-iteration dependences actually occur. If the flag indicates a misprediction, 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

```
/*** 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 6: The speculation loop created from the sequential loop in Figure 2. All added variables are thread-private; only for speculation failure is a shared variable (not shown) needed.

begins execution on the iteration of the misprediction. This process is illustrated at the bottom of Figure 3. Note that all operations are local to each processor except for the success or failure of that processor's speculation; no other global communication is required.

Undoing writes involves restoring the values from the 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. As explained earlier, 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 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 paral-

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.

lelism 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.

Additional compiler analysis can be used to eliminate runtime overhead. For many accesses in a loop, the compiler may be able to prove that the accesses have a given pattern. Such accesses do not require any profiling or prediction checking and can allow the loop's dependence analysis to be partially evaluated at compile time. Compile-time information can also be used to reduce or even eliminate the overhead of the undo buffer. 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.

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 Extensions to the Algorithm

The Softspec algorithm as described in Section 3 only handles loops with known numbers of iterations whose bodies are straight-line code. We have developed extensions to the core algorithm to handle loop bodies containing loop-carried dependences and nonlinear control flow, while loops, and nested loops. We have incorporated all of these into our prototype Softspec implementation. These extensions are simple and do not unduly increase runtime overhead. We describe below how to extend Softspec to predict loop-carried dependencies. Details of the other extensions are available in [6].

4.1 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.

5 Experimental Results

We have developed a prototype implementation of the Softspec algorithm consisting of a compiler and accompanying runtime system. The prototype includes all of the extensions of Section 4. The compiler transforms target sequential code into speculatively parallel code which is then linked with the runtime system. The compiler was written in SUIF [3] and the runtime system was written in C.

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. Speedups reported are speedups over the original sequential program.

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.

Figure 7 shows the speedup obtained by Softspec when parallelizing a dense matrix multiplication and on the SPEC95FP benchmark swim. 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 5.3 is more difficult to optimize and thus exhibits higher speedups and much lower slowdown on one processor.

5.2 Linked List Traversal

Code that traverses a linked list can be difficult or impossible to parallelize. If the nodes of the list are all laid out contiguously then the traversal is amenable

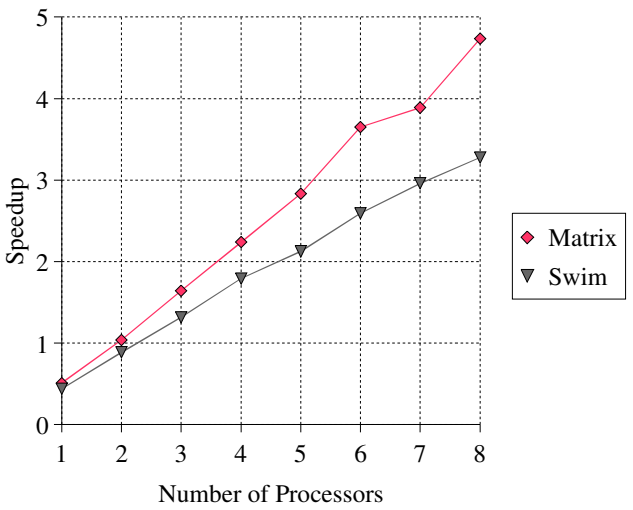

Figure 7: Speedup obtained by Softspec for multiplication of a dense 1000 by 1000 square matrix and on the SPEC95FP benchmark swim.

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. Performance results for this program are given in Figure 8.

5.3 Sparse Matrix Applications

Sparse matrix multiplication is for some matrices stride-predictable and contains parallelism, but is not parallelizable by current compilers. The multiple levels of indirection of sparse matrix storage formats make analysis of sparse matrix code too difficult for current parallelizing compilers, but Softspec's parallelization scheme is applicable.

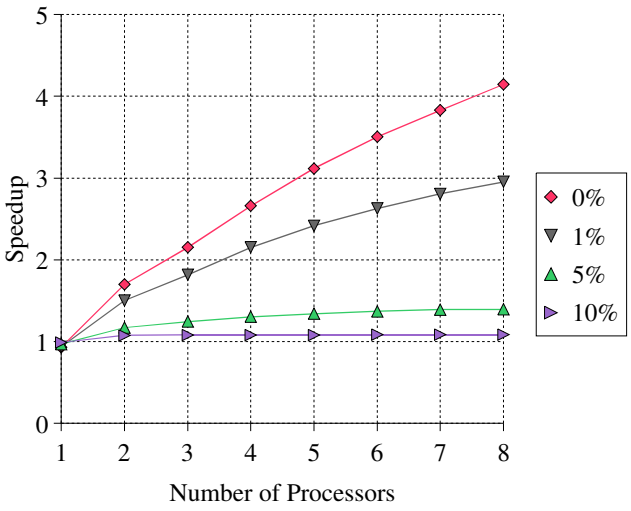

Figure 8: Speedup for a linked list traversal on a list with each sequence of 50 nodes having a given chance of either being contiguous or having frequent gaps between the nodes.

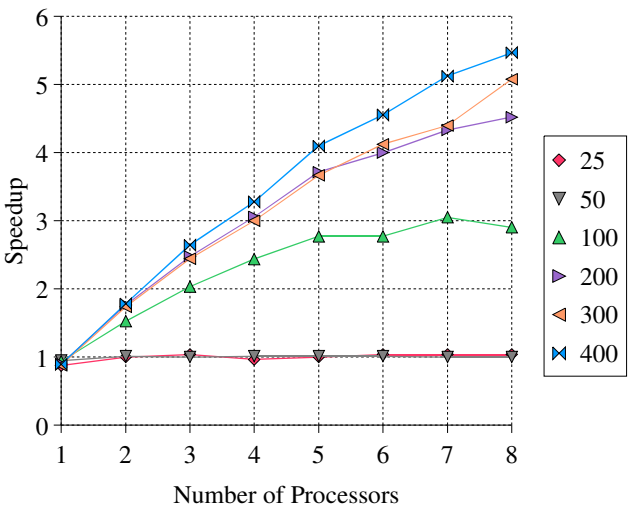

Figure 9: Speedup for sparse matrix multiplication of six block-diagonal matrices whose blocks have widths of 25, 50, 100, 200, 300, and 400, respectively.

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 [19] 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 9 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.

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. These compilers are large systems that use complex interprocedural analyses: the Polaris compiler [5] contains over 170,000 lines of code, and the SUIF compiler [3] contains over 150,000 lines of code. However, they are unable to parallelize loops with complicated or statically insufficiently defined memory access patterns. For these loops runtime parallelization is needed.

Modern parallelizing compilers employ a simple form of runtime parallelization by creating multiple-version loops [7] and delaying some of their analysis until runtime. This technique increases the range of loops they can parallelize but is not sufficient to parallelize all loops.

We can divide other efforts at runtime parallelization into two different approaches: the inspector/executor approach and the speculative approach. Purely software-based schemes have mostly focused on determining the inter-iteration dependence patterns of a partially parallel loop in order to construct parallelization schedules [22]. These schemes focus on an inspector/executor model [21, 25], 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. This approach relies on the separation of address calculations from the main work of the loop and on a quickly inferable memory access pattern. Also, partial loop parallelism often does not scale well (the critical path length often increases with data size so adding more processors may not yield more speedup).

The LRPD test [23] determines at runtime whether a loop is fully parallel (i.e., a doall loop). The test also validates privatizations and reductions. It can be used as an inspector in the inspector/executor paradigm or

it can be performed during speculation, in which case an undo mechanism is required to recover if the test fails. Although the Softspec technique targets only stride-predictable doall loops, this allows it to keep its overhead below that of the LRPD test, which requires shadow versions of every memory location accessed in the loop (except those that the compiler could analyze).

Fundamental research into program behavior has shown that both data and address values can be predicted by stride prediction and last-value prediction [26]. 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 [10]. The stride-predictability of memory addresses has been used to perform speculative prefetching in out-of-order superscalars [11].

Mechanisms for parallelizing certain types of while loops have been developed. Loops traversing lists in Lisp have been parallelized [13] by assuming that the loop has no cross-iteration dependences and that its nodes are allocated in contiguous regions. Loops in FORTRAN have been parallelized [30] by pipelining in doacross fashion, which involves large synchronization costs, or by first executing a sequential inspector that stores the values of the recurrences in the loop and then parallelizing the rest of the loop. The LRPD test can be applied to while loops whose termination condition is well-behaved [24].

The inspector/executor approach has been applied to parallelization of sparse matrix code [29]. The inspector overhead is high due to the multi-level indirections in the code. Sparse array rolling is used to allow a data-parallel compiler to treat multi-level indirections as single-level indirections.

Speculative runtime parallelization using extra hardware has been proposed. Some proposals extend existing multiprocessor hardware [12, 28] while others present completely new hardware structures [18]. 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. The candidate loops are typically identified via compiler, although some schemes instrument the program binary [15]. Value prediction, including stride prediction, has also been proposed for speculative parallelization in hardware [16, 17]. In addition to loop parallelism, procedural parallelism using hardware has been suggested [20]. 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.

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. Although we showed speedup on existing hardware, a simple hardware extension such as adding a speculative mode to the cache system can eliminate undo overhead and improve Softspec's performance.

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. Softspec's simple, local analysis using runtime information allows it to parallelize applications that previous approaches could not. 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.

Softspec is not limited to stride prediction. It can incorporate prediction techniques for different patterns such as tree walks, disjoint regions, and recurrences.

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.

We plan to investigate incorporating into Softspec important parallelism-enabling optimizations such as array privatization and reduction recognition.

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

[1] 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 *Proc. of the SIGPLAN'98 Conference on Programming Language Design and Implementation*, June 1998.

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

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

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

[5] 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/`.

[6] Derek Bruening, Srikrishna Devabhaktuni, and Saman Amarasinghe. Softspec: Software-based speculative parallelism. Technical Report LCS-TM-606, M.I.T., April 2000.

[7] Mark Byler, James Davies, Christopher Huson, Bruce Leasure, and Michael Wolfe. Multiple version loops. In *Proc. of the 1987 Int. Conference on Parallel Processing*, 1987.

[8] 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.

[9] 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`.

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

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

[12] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proc. of the 8th ACM Conference on Arch. Support for Programming Languages and Operating Systems*, October 1998.

[13] W. L. Harrison III. Compiling Lisp for evaluation on a tightly coupled multiprocessor. Technical Report 565, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, March 1986.

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

[15] Venkata Krishnan and Josep Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In *Proc. of the 12th Int. Conference on Supercomputing*, July 1998.

[16] P. Marcuello, A. Gonzalez, and J. Tubella. Speculative multithreaded processors. In *Proc. of the 12th Int. Conference on Supercomputing*, July 1998.

[17] P. Marcuello, A. Gonzalez, and J. Tubella. Value prediction for speculative multithreaded processors. In *Proc. of the 32nd Int. Symposium on Microarchitecture (MICRO-32)*, November 1999.

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

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

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

[21] L. Rauchwerger, N. Amato, and D. Padua. A scalable method for run-time loop parallelization. *Int. Journal of Parallel Programming*, 26(6):537–576, July 1995.

[22] Lawrence Rauchwerger. Run-time parallelization: Its time has come. *Journal of Parallel Computing, Special Issue on Languages & Compilers for Parallel Computers*, 24(3–4):527–556, 1998.

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

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

[25] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Runtime parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[26] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proc. of the 30th Int. Symposium on Microarchitecture (MICRO-30)*, December 1997.

[27] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proc. of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, June 1994.

[28] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proc. of the 4th Int. Symposium on High-Performance Computer Architecture*, February 1998.

[29] Manuel Ujaldon, Shamik Sharma, Joel Saltz, and Emilio Zapata. Runtime techniques for parallelizing sparse matrix applications. In *Proc. of the 1995 Workshop on Irregular Problems*, pages 78–85, September 1995.

[30] Y. Wu and T. Lewis. Parallelizing while loops. In *Proc. of the 1990 Int. Conference on Parallel Processing*, volume II, pages 1–8, 1990.