Softspec: Software-based Speculative Parallelism via Stride Prediction

Srikrishna Devabhaktuni
Advisor: Saman Amarasinghe
M.I.T. Laboratory for Computer Science
Cambridge, MA 02139, U.S.A.

{chinnama, saman} @lcs.mit.edu

Abstract: We introduce Softspec, an all-software, speculation based approach to automatic parallelization of sequential applications. Softspec parallelizes loops containing stride-predictable memory references, without resorting to complex compiler analyses, special hardware support. By detecting parallel regions at runtime and speculatively executing them in parallel, Softspec succeeds in parallelizing codes with memory access patterns that are indeterminable until runtime. We have implemented a prototype system and observed speedup on dense-matrix applications running on a symmetric shared-memory multiprocessor. We show how other classes of applications previously not amenable to automatic parallelization may be parallelized using the Softspec approach, including sparse-matrix applications.

1. Introduction

In the pursuit of higher and higher performance, architects have greatly increased the complexity of hardware and software systems. This is driving up the cost of design, development, implementation, and also decreasing the reliability of these systems. A primary motivation of our work is to discover ways to improve performance of modern computer systems without unduly increasing the complexity faced by either application developers, compiler writers or computer architects.

Symmetric shared-memory multiprocessors are able to provide scalable high performance with only simple hardware extensions to existing uniprocessors. However, these systems are not as widely accepted as uniprocessors due to the difficulty of programming them, making it difficult to develop, debug, and maintain applications. To overcome this difficulty, compiler techniques to automatically parallelize programs have been developed. However, these compilers require complex inter-procedural data-flow and alias analyses to identify parallel regions and prove that they do not have any data dependences between them for all possible program inputs and dynamic program behavior. Consequently, the codes amenable to compile-time parallelization are typically small, self-contained programs consisting of loop nests with affine array accesses. They are also written in languages such as FORTRAN, which permit limited memory aliasing. Current parallelizing compiler techniques fall short of being able to parallelize typical applications, which are large systems with multiple modules written in newer, more popular languages such as C, C++, and Java. Even the FORTRAN programs that were amenable to compile-time parallelization may not be parallelizable once re-written in the newer languages, since these languages permit aliasing of pointers and promote use of a variety of data structures. For example, the mere presence of a single unanalyzable pointer can render compile-time identification of parallelism impossible.

With the aim of parallelizing programs containing pointers and different kinds of data structures, new hardware-based speculative systems have been proposed and researched [1][2][3]. These techniques attempts to overcome the need for a compile-time proof of the existence of parallelism by speculatively executing candidate regions in parallel. The additional hardware support is required to identify violations of the sequential semantics of the program and annul the effect of speculative execution in case of a violation ranges from significant extensions of existing multiprocessing memory systems [1][2] to completely new hardware structures [3].

In this paper, we approach the problem of parallelizing modern sequential applications by relying on two observations: one about program performance and one about memory access patterns. These permit us to increase performance of applications without unduly increasing the complexity of the hardware and software required. First, it was observed in both parallelizing compilers and speculatively parallelizing schemes that good speedup was achieved when the parallel regions do not have any inter-dependences. These *doall* regions can be executed in parallel without any synchronization. By comparison, programs with memory dependencies between iterations that are executed in parallel must incur the overhead of frequent synchronization, to manage the memory dependencies. Secondly, we observe that the addresses appearing in loop iterations can be predicted based on runtime information. In Section 2, we show that applications with affine accesses as well as sparse matrix and integer applications have many predictable accesses.

This paper presents a parallelization scheme that takes advantage of these observations. Using runtime prediction of memory addresses of loop iterations, this scheme detects speculatively parallel regions. These regions are executed in parallel with a simple mechanism to detect speculation violations. When the prediction is accurate, then the speculative parallelism will succeed and scalable speedup can be observed. If the prediction is found to be inaccurate while executing the program in parallel, a (parallel) mechanism is provided to undo the effect of the speculation so that the loop may be re-executed sequentially. There is no need for synchronization or access to the same memory locations by different processors during parallel execution.

To provide a working example of this approach and evaluate its runtime performance, we implement Softspec: a compiler and an accompanying runtime-system to compile sequentially written C programs to run speculatively in parallel on a shared-memory multiprocessor. The compiler can be applied to programs containing loops with stride predictable load/store addresses, including loops containing affine array accesses and sparse matrix applications that are not parallelizable using compile-time analysis. Code generation involves only localized analysis of code regions and does not make use of whole program analyses. Our implementation is all-software and no hardware modifications to the multiprocessor are needed. We measure speedup when the compiled programs are run on a modern Digital Alpha multiprocessor, and we expect that these programs will achieve even greater speedup on tightly-integrated single-chip multiprocessors envisioned for the future [14]. Applicability of this approach hinges on the predictability of dynamic addresses to detect whether there will be inter-thread memory dependencies. It is not necessarily limited to programs with stride-predictable load/store addresses.

The rest of the paper is organized as follows. Section 2 describes the Softspec approach. Section 3 describes our prototype compiler and runtime system implementation. In Section 4, we examine the stride predictability of Spec92 benchmarks and provide three case studies of dense-matrix and sparse-matrix applications, to evaluate the Softspec approach. In this section, we also provide results for the runtime performance of dense-matrix codes that are parallelized by our compiler. Section 5 provides an overview of related work and section 6 concludes the paper.

2. The Softspec Approach to Parallelization

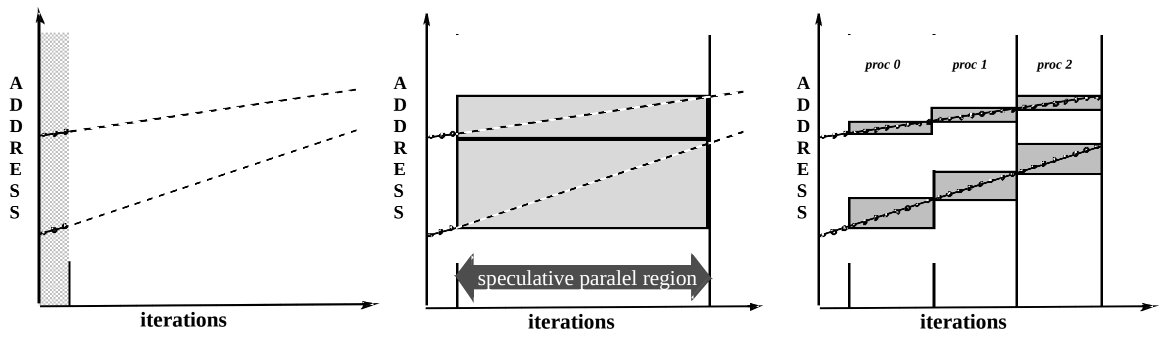
This section presents an overview of our novel approach to parallelization. In parallelizing a loop, the read and write accesses of the first three iterations are profiled. Using this profile information, the runtime system checks for stride predictability. If all accesses are stride predictable during the first three iterations, the runtime detection phase checks if the loop is parallelizable assuming that the stride predictions hold for the rest of the iterations. If parallelism is detected, the loop is speculatively executed in parallel. If the accesses behave as predicted during the parallel execution, the speculative parallelization of the loop will be a success. However, if there is a misprediction, part of the speculative parallel execution after the misprediction is aborted and the effect of that speculation is undone, i.e. Softspec has the ability to restore the state of the machine memory to what it was before speculative execution. The rest of the section describes these four steps, prediction, detection, speculation, restorability, and undo, in detail.

2.1 Stride Prediction

A load/store is said to be stride predictable, if the address it accesses is incremented by a constant stride for each successive dynamic instance (for example in a loop). Array accesses with affine index expressions within loops are always stride-predictable because of a basic property of affine functions.

In our approach to parallelization, the compiler does not perform analysis to prove that a loop has affine index expressions, or that its accesses are stride predictable. By profiling the addresses appearing in the first two iterations of a loop at runtime, it is possible to infer the stride with which addresses appear in subsequent iteration using stride-predictability. In practice, three iterations are profiled to additionally perform a check to verify that the addresses appearing in the third iteration follow the stride pattern deduced from the addresses in the first two iterations. This is illustrated in Figure 1 (a), where the addresses of two hypothetical memory references are graphed for the first three iterations and linearly extrapolated to predict the addresses appearing in subsequent iterations.

It has been observed that many memory references, beyond affine access functions, can be stride predictable [8][10]. In Section 2.1, we show that both floating-point and integer benchmarks of the SPEC92 suite have high degree of stride predictability of memory addresses.



(a) Prediction of two accesses after 3 iterations (b) Detection of speculative parallel region (c) Speculative parallel execution on 3 processors

Figure 1: Profile, prediction, detection and speculative execution of a loop with two memory accesses

2.2Detection of Parallelism

Once the addresses accessed by the loop iterations have been profiled and the strides have been calculated, it is necessary to determine whether the parallel execution of the loop will violate the sequential semantics. The requirement for inter-iteration parallelism is that a write in one iteration should not be to the same address as another read or write in another iteration. Otherwise, true, anti, or write dependences will result. Figure 1 (b) shows how the two address predictions from Figure 1 (a) are compared to find the maximum number of loop iterations where the two address ranges do not overlap.

We now show how to determine whether two stride predictable memory references will lead to inter-iteration data-dependencies. Suppose there are two memory instructions within the loop body, whose access addresses $\mathbf{p_0}$ and $\mathbf{q_0}$ during the first iteration, and continue to access memory with strides **s** and **t**, respectively. In general **s** and **t** may not be equal. Then on the **n**'th iteration, the accesses will be $\mathbf{p(n) = p_0 + s*n}$ and $\mathbf{q(n) = q_0 + t*n}$ respectively. If for some $\mathbf{m} \geq 0$ and $\mathbf{n} \geq 0$, both less than **N**, we have $\mathbf{p(m) = q(n)}$, then a memory dependence will result when **N** iterations are executed in parallel. This condition results in a diophantine equation in **m** and **n** with a solution set parameterized by one integer variable. Solutions to this equation indicate which iterations will have dependencies. From this solution the maximum number of iterations of the loop which can run in parallel can be calculated as follows.

We can re-write this equation as $\mathbf{s*m – t*n = q_0 - p_0}$. This equation will admit solutions if and only if the greatest common divisor of **s** and **t**, namely **(s,t)**, divides $\mathbf{q_0 - p_0}$. In this case, the solution set of this equation can be obtained as a singly-parameterized set, by using Euclid's gcd algorithm which also computes the **(s,t)**[16]. Once the parameterized solution is obtained, a further calculation determines the smallest non-negative values of m and n, which are solutions. From this, we can calculate the largest number of iterations that can be run without incurring inter-iteration memory dependencies.

By pairwise comparison of references, it is then possible to determine how many iterations of the loop can be parallelized. If there are **R** load instructions and **W** write instructions within the loop body, **W(W+R)** comparisons must be performed and the minimum of all the results is the number of parallelizable iterations within the loop.

2.3 Speculative Parallelization

Upon calculation of the number of parallel iterations within the loop, the program needs to make a runtime decision of whether or not to attempt parallel execution. If the number of iterations predicted to be parallel is too small, then the program will decide to execute the loop sequentially. Otherwise, the loop enters a speculative parallel execution where loop iterations are executed in parallel on the available processing nodes as illustrated in Figure 1 (c). If there are no mispredictions, then speculation is a success and the program has executed correctly in parallel.

2.4 Restorability

In case that any addresses were mis-predicted, the calculation performed by the speculative execution could violate the sequential semantics of the original program. Therefore, a mechanism is needed to ensure correct execution of the loop in this case.

Correctness of execution can be ensured only if the speculative execution has a property that makes it possible to restore the machine memory to its pre-speculation state in case of a mis-prediction, so that the loop may be re-executed sequentially. We call this property the *restorability* property of speculative execution. The *undo* mechanism we designed to implement restorability into the Softspec execution model is described in the next section. To enable an efficient mechanism that implements restorability, we design the Softspec program so that speculative execution does not cause any true, anti, or output dependencies between the iterations executed in parallel on each processor. Without such a design, different processors could be writing and reading to the same memory locations leading to hazards, making it impossible to reconcile the order of reads and writes between processors when the effect of speculation on the machine's (shared) memory needs to be undone. To prevent inter-iteration dependencies during speculation, we design the program so that

I. it performs speculative execution using the predicted addresses in place of the actual addresses for the loads/stores in the loop body, and

II. it enters speculative parallel execution only if the predicted addresses will not lead to true, anti, or output memory dependencies between the parallel iterations (as described in the previous section).

Since the actual addresses appearing in loads/stores within an iteration are unknown until the program executes that iteration, there is no way to guarantee that these addresses will not lead to dependencies in case of mis-predictions. Therefore I is necessary. II uses I to ensure that the loop is executed speculatively in parallel only if it is free of inter-iteration dependencies.

2.5 The Undo Mechanism

During parallel execution, the code running on each processor incorporates checks to determine if any of the predicted addresses differ from the actual addresses. When a misprediction is detected the effect of the remaining iterations, some of which may have already have been executed by other processors, needs to be undone. Once a misprediction is detected by a processor, the fact (one bit of information) is communicated to the other processors and each processor must restore the machine’s memory to the pre-speculation state.

Writes to a memory location destroy the value stored previously at that memory location. Therefore, as part of the Softspec approach, each processor saves the original values in a write buffer just before they are about to be overwritten. To recover the pre-speculative state of memory after parallel execution, each processor simply restores each value that it overwrote to the proper memory location. Additionally, since intra-iteration dependencies may exist when there are no inter-iteration dependencies, the processor restores the values in reverse order of which they were written; i.e. each processor restores the most recently written value and then proceeds to restore the next most recently written value, etc. The restoration is guaranteed to be work correctly without resulting in dependencies among the processors, since the writes performed by each processor during speculation use predicted addresses. These predicted addresses were dependence free by construction. Restoration of the values can proceed in parallel on all the processors for this same reason, requiring no global communication except for the broadcast of the existence of mispredictions between processors.

3 Softspec implementation

In this section, we describe a prototype implementation of a compiler and runtime system which is used to parallelize sequential programs using our approach. The implementation we describe is capable of parallelizing inner loops containing stride predictable memory accesses without resorting to inter-procedural analysis or whole-program analysis. It requires only simple localized analyses of code regions.

Our software implementation consists of a runtime system linked into code generated by a compiler, which takes a sequential C program as input and produces a parallelized C program as output. The runtime system was written in C using the POSIX Pthreads library [21]. The source-to-source compiler was written using the SUIF compiler infrastructure [19, 22]. Code is generated to run on a Digital AlphaServer 8400, which is a bus-based shared memory multiprocessor containing eight Digital Alpha processors.

We use the Pthreads library to create parallel threads once when the program starts and terminate them only when the program ends. A single set of threads is used for all loops within the program. Synchronization among threads is effected through the use of low-latency shared counters and symmetric memory barriers. We avoided using locks and other synchronization primitives provided in the Pthreads library, since they had unacceptably high latencies.

We illustrate the parallelization of a loop by our compiler through an example.

```
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);
      }

}
```

This loop has three memory accesses namely two writes {a[i], b[i+j]} and two reads {b[i], (*p)}. These accesses are affine with respect to the loop index and therefore will be stride predictable with strides 8, 8, 8, and 0 respectively. In order for the loop to be parallel, there should be no inter-iteration dependencies between the reads and the writes, as well as between the writes themselves. In this example, the loop will be parallel if a and b are non-overlapping arrays of 500 doubles and j >= 500. However a parallelizing compiler must make several inferences before coming to this conclusion. For example it must deduce that a and b are distinct arrays. Second, it must deduce that there will be no inter-iteration dependence between b[i+j], b[j], and (*p). Deducing this information at compile-time requires sophisticated whole-program alias analysis, or may be impossible since the dependencies may be input-data dependent. The problem becomes much simpler by simply measuring the initial values and strides of these memory accesses at the beginning of the loop and calculating whether the loop will be parallel, at runtime. Blind instrumentation of loads and stores within the loop body is all that is needed with Softspec, even if the code is located within procedures. Our compiler also avoids the overhead for privatizable local variables and reads to local variables within the loop body, through scalar privatization analysis, local to the loop body. Additionally, certain simple loop carried dependencies caused by local variables such as induction variables, can be eliminated through similar analysis on the loop body alone.

The compiler replaces the loop with three different loops: profile, detect, and speculate. The first two loops are run sequentially and the third one is the one during which the loop is executed speculatively in parallel. A fourth loop is needed to sequentially execute the remaining iterations in case the speculation fails. Figure 2 shows the execution path of each loop, with the vertical axis representing time, and the horizontal axis representing the processors in the SMP.

Each box represents the loop that is running on the respective processor, and the numbers within each box show the iteration numbers being run on each processor for a hypothetical example.

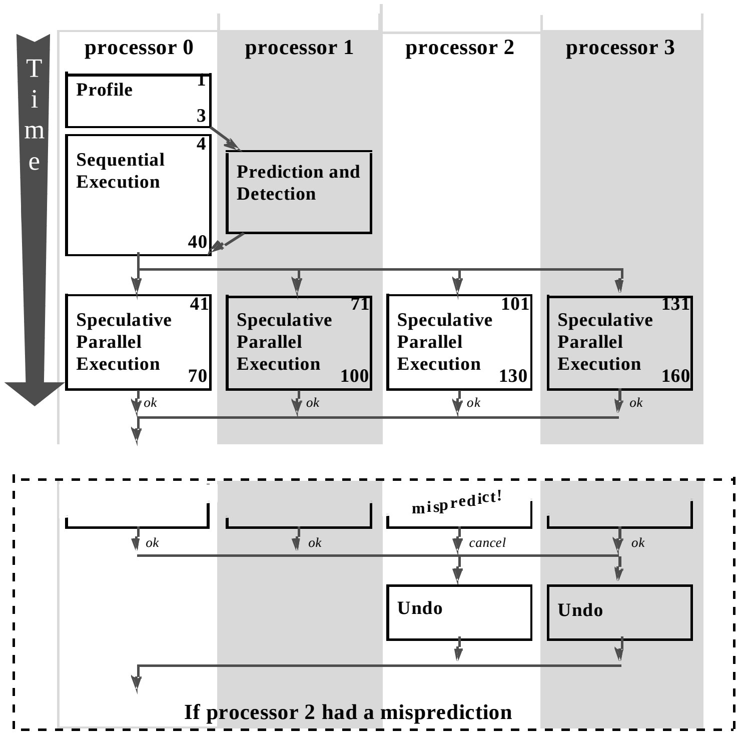


Figure 2: execution path of the processors in a Softspec parallelized loop.

3.1 Profile Loop

The address of the first write, a[i], will be &a[0], and the address of the write in the second iteration will be &a[1]. Using this information we conclude that the difference will be &a[1] - &a[0] = sizeof(double) = 8. The write in the third iteration is likely to be a[1] + 8 = a[2], which is indeed the case since the write is to a[i] on the i'th iteration. In the example, the remaining three memory accesses are also stride predictable.

The profile loop profiles the addresses appearing in three iterations and additional data structures in the runtime system are used to keep track of the addresses. The addresses are stored in these data structures after two iterations are executed. The strides are calculated by subtracting the addresses appearing in each load/store in the second iteration, with those appearing in the first iteration, respectively. The addresses obtained from the third iteration are used to recalculate the stride and verify that the stride remains constant across the three iterations. This check identifies many of the non-stride predictable accesses, and stops parallelization early in the process.

The code and data structures needed to implement the profile loop might look something like this, with code inserted within the loop body to store the addresses appearing in the first three iterations.

```
/* in runtime system */
void * profile_address[3][NUM_MEM_REFS];

/* in code generated for profile loop */
for(iter=0; iter<3; iter++)
{
     int i=iter;

     profile_address[iter][0] = &a[i];
     a[i] = i;
     profile_address[iter][1] = &b[i+j];
     profile_address[iter][2] = &b[i];
     profile_address[iter][3] = p;
     b[i+j] = b[i] -(*p);
}
```

Note that the `profile_address` data structure is shown as a pointer array. The outer index of this array is for the particular iteration that is being profiled. The inner index is used to consecutively number the memory accesses. In practice, the reads and write to different data types use separate `profile_address` arrays for reasons having to do with recovery from misspeculation.

3.2 Detection

By making use of the addresses and strides obtained by the prediction loop, the number of parallel iterations in the loop can be determined, assuming that the stride prediction holds true throughout the loop. In our implementation, we overlap execution of the calculation with execution of a copy of the loop so as to continue to make forward progress. This is important since the detection calculation may take as much time as 10s of iterations. It is always advantageous to make the detection faster since it means that the program can start parallel execution quicker, rather than running sequentially while the detect procedure is running in a parallel thread.

The intersect procedure

The intersect procedure takes the initial addresses and strides of two memory references and calculates the earliest future iteration at which a hazard can occur. One implementation of the intersect procedure solves the diophantine equation from section 2.3 exactly, by making use of Euclid's algorithm to calculate gcd's. Roughly 80-90% of the execution time for detecting parallelism was spent in the procedure that calculates the gcd. Therefore we have employed optimized versions of the gcd and obtained improved runtime performance. Secondly, we have implemented an approximate algorithm that is able to conservatively estimate the number of parallel iterations and at the same time decreasing the time spent in intersect. For a detailed discussion of optimized algorithms, see Appendix A. In figure 3, we show the (averaged) number of cycles taken by the optimized detect procedure for different numbers of reads and writes. The 3-d graph indicates that how the time taken by the detection algorithm scales with the number of reads and writes. From this figure we can deduce that the time increases much faster with an increase in the number of writes than an increase in the number of reads, which can be explained by the theoretical scaling of **W(W+R)** as derived in section 2.2.

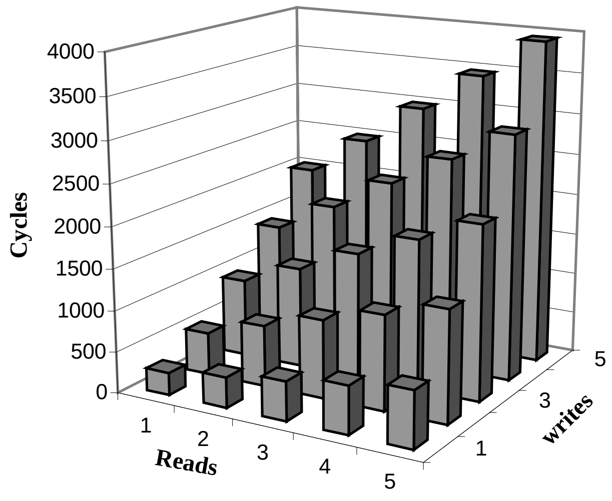


Figure 3: Execution time for detection algorithm

3.3 Speculation and Recovery from Misprediction (Undo)

Once the program has detected that parallelism exists within the loop, the program initiates the speculative parallel mode where loop iterations are executed in blocks on the processors. The speculative version of the loop involves several mechanisms to incorporate calculation of the predicted address, comparison with the actual address, and the undo mechanism. Throughout this section, we illustrate the mechanisms employed by providing pseudo-code which implement the mechanisms for loads/stores.

Since the parallelization is based on a prediction, the predicted addressed are calculated to be the first address appearing in the loop iteration offset by the number of loop iterations times the stride for that load/store.

```
predict_address = first_address + iteration * stride;
actual_address = [caculated from loop body]
```

We must be able to ensure correct execution in case of misprediction. To verify if a misprediction has taken place, we insert code to check whether the actual address is equal to the predicted address. During parallel execution, each processor maintains a flag that keeps track of whether the predicted addresses are equal the actual addresses. If there is no discrepancy between predicted and actual addresses, the speculation would have been correct. However if there is a discrepancy, mis-speculation has occurred and must be re-executed sequentially. All the iterations after the mispredicted iteration must be executed sequentially. This code appears as follows,

```
if(actual_address != predict_address) cancel_flag = 1;
```

At the end of speculation, a processor which encounters a misprediction informs all other processors, and all the processors can undo the effect of speculation as described below. In case of a load, no action need be taken since load operations do not change the state of memory. However in the case of writes, additional mechanisms are required. A write buffer, implemented as an array, is provided to save values about to be overwritten. In the pseudo-code below, each value about to be overwritten is stored at location pointed to by write_buffer, and the pointer into the write buffer is incremented.

```
overwritten_value = *(predict_address);
*(write_buffer++) = overwritten_value;
```

To undo the writes each processor has performed, the runtime system simply restores the values from the write-buffer into the predicted addresses for each iteration, starting from the most recent iteration and proceeding to the earliest executed iteration. Each processor can undo its writes in parallel, and without global communication, since the writes by different processors were written to different addresses. We note that this is another advantage of using prediction to eliminate the possibility of inter-iteration data dependencies before parallelization.

3.4 Softspec Compiler Architecture

The Softspec compiler takes a sequential C program as input and generates speculative parallel C code. The resulting code must be linked with the Softspec runtime library and compiled with the native C compiler to produce an executable that runs on a Digital Alpha symmetric shared memory multiprocessor.

The Softspec compiler (source code in Appendix B) was written using the Stanford University Intermediate Format (SUIF) basic library [22], which itself is written in C++. SUIF has many built-in capabilities that permit rapid prototyping of source-to-source compilers. C programs once read into SUIF, may be analyzed, transformed, and output back into C. The intermediate format has rich data structures that can represent low-level and high-level programming constructs, such as procedures, for-loops, as well as local and global variables along with the symbol tables they are defined in. The SUIF data structures are represented as objects with accompanying methods that are useful for analyzing and transforming them.

SUIF provides a translator (scc) which converts input C programs to SUIF's intermediate format. Compilers in SUIF are written as a series of passes which perform a certain well-defined transformation on the program, once the program is in the SUIF intermediate format. Each pass reads in the result of the previous pass that is stored in a file, then performs its task, and outputs the result to a file. Each pass is a C++ program (using the SUIF library) that manipulates the SUIF data structures and calls the SUIF library to accomplish its specified sub-task on the program. Any information that needs to be conveyed by one pass to the next pass, such as the results of code analyses, can be inserted in "annotations" attached to the SUIF data structures.

Numerous pre-written passes come with the SUIF standard library, and they perform a variety of code transformations and optimizations. While the use of intermediate file storage formats can become inefficient in terms of the time taken by the compiler, it provides flexibility in mixing and matching passes to architect new compilers.

Our current implementation of the Softspec compiler parallelizes inner loops of programs that do not contain any control flow or procedure calls within the loop body. The Softspec compiler analyses the program, to identify candidate loops to parallelize, then generates parallel code with calls to the accompanying runtime library. It is implemented as a SUIF pass, and the design bears resemblance to the Haltsuif profiling tool [20]. Prior to running the compiler pass, a scalar variable analysis pass (moo –Psce) must be run to get information (passed as annotations to the compiler) on the behavior of local variables appearing inside the loop body. It determines whether local variables can be privatized within iterations. If local variables lead to loop-carried dependencies, the compiler cannot privatize the local variables and decides not to parallelize the loop and more sophisticated dynamic compilation techniques are needed.

Once the compiler searches for and discovers a candidate inner loop to be parallelized, it proceeds to generate three loops each within its own procedure that correspond to the profile, detect, and (parallel) speculative loops. In the generated code, each of the procedures called by these loops is called in sequence at the location of the original loop, and the original loop is left intact to execute any remaining iterations if the speculative loop does not run to completion.

3.5 Runtime System Architecture

During parallel execution, the execution path of a Softspec program consists of multiple threads performing detection and speculation, apart from the thread of the program that executes the main procedure, as illustrated in Figure 2. We implemented our runtime system in C using the POSIX Pthreads library [21]. As the main procedure executes, it may contain several loops that are parallelized by the Softspec compiler, interspersed by several sequential regions. The execution time overhead of spawning new threads for each parallelized loop at runtime is too much, as determined by our experiments. Our solution then is to spawn the parallel threads when the main program begins, and signal these threads via shared memory to perform detection and speculation when the main thread encounters the parallelized regions. The architecture of the runtime system is largely determined by this design decision (source code in Appendix C).

During execution, once the main thread profiles addresses in the loop, it stores the address into data structures accessible by the runtime system. Next, the main thread sets a flag in shared memory that informs the detection thread to perform detection using the profiled data. The main thread then enters sequential execution of the loop to make forward progress while the detection thread is running. Once the detection thread completes detection, it in turn informs the main thread via shared memory so the main thread can use the results of detection to enter speculative parallel execution.

Similar to the detection thread, several speculative threads are spawned at the start of the main thread. However, several additional mechanisms are needed to transfer the context of the main thread to the speculative threads. Before informing the speculative threads to begin executing, the main thread must inform each speculative thread of the values of the local variables in the loop body. This is accomplished by having the main thread load these values into shared memory and having each speculative thread read these values into its own local variables. Since the speculative threads are active throughout the execution of the program, and multiple parallel loops may appear during the program, the speculative threads require a means of knowing which loop (i.e. code segment) they should be speculatively executing. This is accomplished by also passing the address of the procedure containing the speculative loop in shared memory to all the speculative threads, just like the values of local variables. Code for the undo mechanism need not be generated by the compiler, and it is included as part of the runtime system.

Synchronization among the speculative threads to determine the beginning and end of speculative execution requires low-overhead mechanisms to be effective. The synchronization primitives in the Pthreads library do not satisfy this requirement, so we implemented our own synchronization mechanisms based on shared memory flags. The speculative threads are numbered consecutively and each thread is responsible for informing the next thread when it began and commenced speculative execution, through a shared memory flag. By spacing the flags out in memory longer than a cache line, this approach can avoid cache conflicts between different processors.

3.6 Comparison of Softspec to sequential execution

In this section, we look at the overheads on the source loop which are required to compile it into a Softspec parallelizable loop. Each loop is replicated into three other modified loops: profile, detect, and speculate.

The profile loop differs from the original loop since addresses are written to a data structure in the runtime system, before they are used by loads and stores. The detect loop is designed to run in parallel with the calculation for detection of parallelism, so as to make forward progress. It is nearly identical to the original loop, except for a check to determine the whether the detection calculation has finished or not. Finally, the speculation loop incurs an overhead for each load/store to calculate the predicted address and check whether the predicted address equals the actual address. Each write additionally incurs the overhead of saving the values about to be overwritten. An estimate of the overheads in terms of machine instructions are given below in the table,

	Per iteration	Per read	Per write
Profile	0	4	4
Detect	2	0	0
Speculation	0	6	11

In addition, synchronization between the parallel threads adds additional overhead, which is not present during sequential execution. A barrier is required at the end of parallel execution to determine if any of the processors observed address mispredictions. This barrier may cost several thousand cycles on a shared-memory multiprocessor, and it increases linearly with the number of processors.

Symmetric multiprocessors on a chip, as envisioned for the future [14], are likely to reduce synchronization costs (in terms of execution time overhead). Application of Softspec techniques in such a realm becomes more attractive since the overheads due to synchronization appearing between the profile, detect, and speculative loops will be reduced.

2 Behavior of Programs and Runtime Performance

In section 4.1, we present evidence for stride-predictability of dynamic memory addresses in Spec92 benchmarks to indicate the potentially wide applicability of the Softspec approach. In section 4.2, we present three case studies of two dense-matrix and one sparse-matrix applications to evaluate the Softspec approach and study the runtime performance of our prototype Softspec implementation.

4.1 Stride Predictability of Spec and Sparse-matrix benchmarks

Stride Predictability of memory addresses is a common behavior of programs. We used the ATOM profiling tool to measure the extent of stride predictability in several Spec92

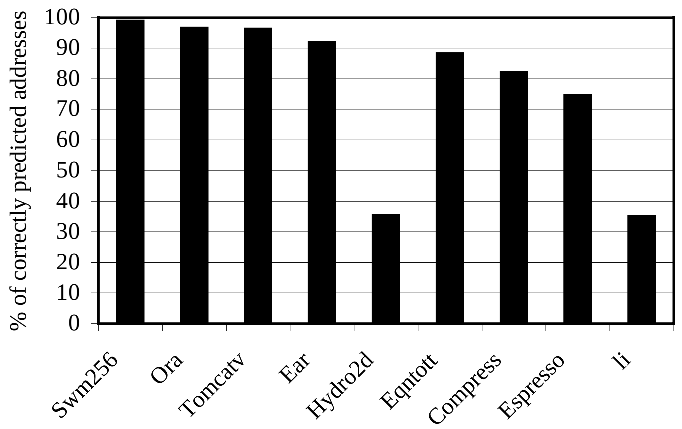


Figure 4: % stride predictability of Spec92 benchmarks

benchmarks. In the graph below, we show the percentage of dynamic memory references which could be predicted from the previous two dynamic instances of the same memory reference.

The high percentage of stride predictable references among floating point benchmarks (swm256, Ora, Tomcatv, Ear, Hydro2d) can be understood since most of the loops contain affine index expressions for loads and stores. The numbers are also encouraging in the case of the first three out of four integer applications (Eqntott, Compress, Espresso, Li) which exhibit stride predictability in more than half of the memory references. Experimental evidence for the stride-predictability of load addresses in several Spec benchmarks have also been reported in [8][10].

4.2 Case Studies of the Softspec Approach

To evaluate the runtime performance of this approach, we present an analysis of the execution times of familiar benchmark, namely (dense) matrix-multiplication in section 4.2.1 and a code which is not parallelizable by most parallel compilers, namely Serial Bidiagonal Cyclic

Reduction [15] in section 4.2.2. In section 4.2.3, we present evidence of the parallelizability of sparse matrix multiplication code, which cannot be parallelized by any compiler.

The dense matrix codes are compiled using our prototype Softspec from sequential C. The output of Softspec is compiled to with the native C compiler (cc) using the O2 and pthreads flags. The speedups over sequential execution (cc –O2) are shown in Figures 5 and 6.

4.2.1 Dense-Matrix Multiplication

The main loop nest in this (dense) matrix-multiplication application is:

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

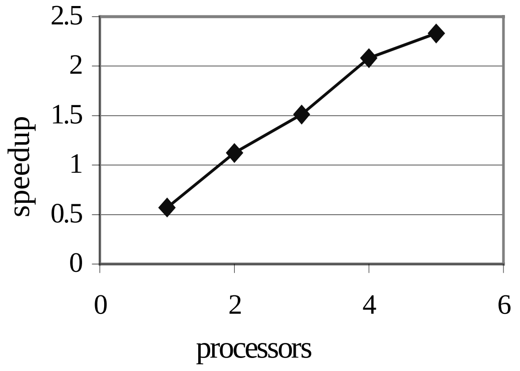


Figure 5: matrix multiplcation: speedup

The inner loop is parallel, and Softspec is able to parallelize this loop at runtime and achieve speedup shown in the graph in Figure 5.

We observe that the address check and write buffer overhead of the loop body, which has very little computation, is approximately 100% in this application. When two or more processors are used to carry out speculative parallel execution, we observe nearly linear speedup with the highest being 2.5 for 6 parallel threads.

To understand the overheads introduced by the profile, detect, and speculative parallel execution of the parallelized program, we show the number of iterations and processor cycle counts for each of these regions for the inner loop of the 10,000x10,000 matrix multiplication and compare them to the original sequential loop, in the table below.

Loop	Original	Profile	Detect	Speculate
Iterations	10,000	3	50	9947
Cycles	140269	543	2377	264436

The overhead introduced by the profile and detection loops is insignificant, in terms of both cycles and iterations. For loops with far fewer iterations, the overhead of the detection loops can be significant, and overlapping the detection phase with sequential execution of loop iterations allows the program to make forward progress, as our implementation permits.

4.2.2 Serial Bidiagonal Cyclic Reduction

Serial Bidiagonal Cyclic Reduction [15] operates on dense-matrices. However, the data access patterns are determined at runtime, and are hard to analyze by the compiler. For example, one of the loop nests present in this program is,

```
r = n/2;
x[r]  = b[r];
x[n] = b[n] - a[n-1] * x[r];
for(p=m-1; p>1;  p--) {
      r = r/2;
      x[r] = b[r];
      for(j=3; j<n/r; j+=2) {
            rj = r*j;
            x[rj] = b[rj] - a[rj-1] * x[rj-r];
      }
}
```

Most parallelizing compilers cannot analyze the data-accesses in this program, since the access depends on the variable r, whose value is not available at compile time. Depending on the value of r, the loop can be parallel or sequential, and the value of r is calculated in the outer loop. The advantage of Softspec is that is it is able to parallelize the inner loop based on runtime information of the value of r for each iteration. We present the speedup observed for this application in the graph in Figure 6.

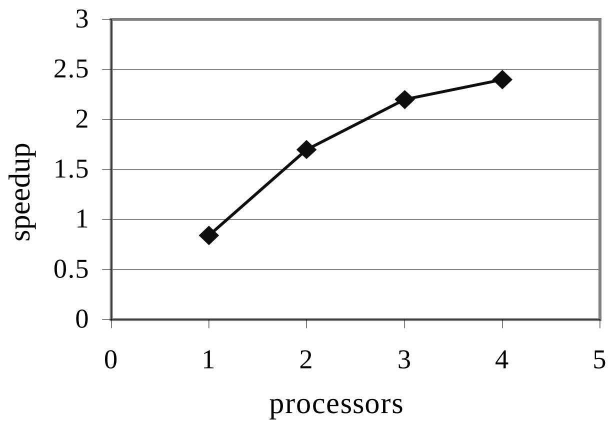


Figure 6: serial bidiagnal cyclic reduction speedup

We were able to achieve good speedup with low overheads for the two small benchmarks, matrix multiplication and serial bidiagonal cyclic reduction, that were fully automatically parallelized and executed on a modern multiprocessor using Softspec.

4.2.3 Sparse matrix multiplication

In this section we give an example of a sparse-matrix application which exhibits stride-predictability, and has parallelism, but is not amenable to compiler-analysis. Sparse-matrix algorithms differ from their dense-matrix counterparts largely because of the special storage formats used for sparse-matrices. The code for this application is an adaptation of the matrix multiplication algorithm given in section 4.2.1 for sparse matrices using compressed row storage (CRS) format. The loop nest for the code is provided below.

```
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
```

In this loop nest, the sparse matrices stored in a() and b() are being multiplied and stored (in CRS format) in c(). The arrays offa, ja, offb, and jb are used to locate the columns and rows of elements in the sparse matrices stored in a() and b(). The inner loop contains an *if-then-else* statement. The *then* statement initializes new sparse elements in the c() array, and therefore is far

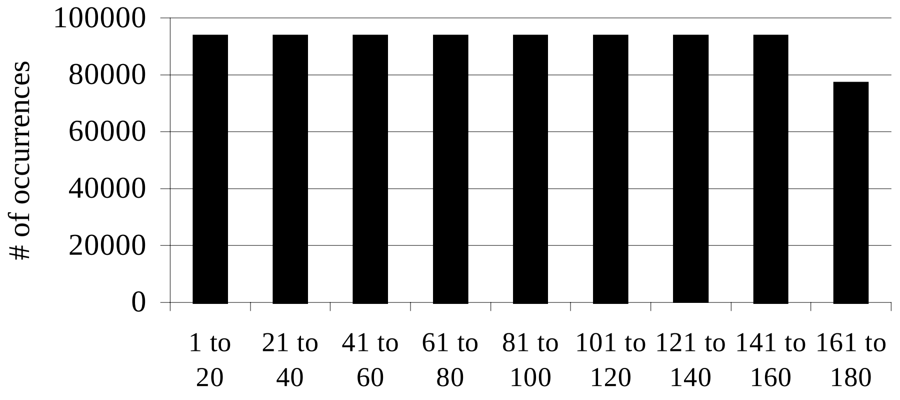


Figure 7: qc2534.cua: histogram of stride-predictable parallel iterations

more likely to take the *else* direction than the *then* direction. When the loop takes the *then* part, it has a loop carried dependence. However, when it takes the else part consecutively for many iterations, the loop is parallel, since, by construction, its inner loop directly corresponds to the inner loop of the (dense) matrix multiplication algorithm given in section 4.2.1.

Even though the loop may be parallel, its the memory accesses in the *else* part of the inner loop body are to five different arrays, namely jb, ptr, a, b, c and there are two levels of indirection in the access c(ptr(jb(k))). Therefore it is not plausible that any purely compile-time based approach to parallelization can deduce the existence of parallelism in the loop.

In order for this loop to be stride predictable, all the accesses within the loop must be stride predictable. We profiled the inner loop to determine how often and for how many iterations the memory accesses will be stride predictable and parallelizable. This is highly dependent on the sparse-matrix data set used, and we attribute the existence of stride predictability to the presence of smaller dense regions within the sparse-matrix. Two data-sets exhibiting good stride predictability and long stretches of consecutive parallel iterations are qc234 [17] and qc2534 [18] from the Non-Hermetian Eigenvalue Problem Collection. qc234 has 2,205,306 iterations of which 1,879,192 (85%) are parallelizable and stride-predictable. qc2534 similarly has 78,222,562 (92%) parallel iterations out of 84,728,996. For the qc2534 data set, we provide a histogram, in Figure 7, of the number of parallel regions found for a given length of the parallel region in terms of iterations. This data suggests that the linear sections may be parallelized at runtime using the Softspec approach. Our prototype runtime system and compiler is unable to parallelize this application, since we have yet to implement speculation invalidations due to memory references and loop-carried dependences within rarely taken conditionals. We plan to extend Softspec to take advantage of such programs and study more sparse algorithms and different data-sets.

5. Related Work

Advances in vectorizing and parallelizing compilers over the past two decades have led to remarkable successes in automatic generation of highly efficient parallel code for certain kinds of programs, such as dense matrix calculations. However these compiler systems are large complex systems. Examples include the Stanford SUIF interprocedural parallelizing compiler [1] which has over 150,000 lines of code and took more than 10 human years to develop, and the Illinois Polaris compiler [2(a)] consisting of over 170,000 lines of code [2(b)].

Alias analysis algorithms have been proposed to overcome the difficulty imposed by pointers on parallelizing compilers [3][12][13]. However these systems work with only with small, self-contained programs and do not scale with program size.

Speculative systems, based on hardware features not yet available on today's machines, have been recently proposed to overcome the need for compile-time proof of existence of parallelism in programs [4][5][6][7]. These systems speculatively execute candidate loop iterations in parallel. A sophisticated hardware system is put in place to observe all the memory accesses in order to determine if the sequential semantics of the program are violated by speculative parallel execution. When a violation is detected, the speculative execution is revoked and additional hardware mechanisms will undo the effects of speculative writes on the memory system. Additional hardware support required by some of these proposed speculative paradigms ranges from significant extensions of multiprocessor memory systems [4][5] to completely new hardware structures [6].

Fundamental research into program behavior has shown that both data and address values can be predicted by stride prediction and last-value prediction [8]. Stride predictability of load/store addresses in scientific applications has been successfully exploited to improve the cache behavior of these codes through compiler inserted prefetching in uniprocessor and multiprocessor machines [9]. In [10], the stride predictability memory addresses has been used to perform speculative prefetching in out-of-order superscalars.

The Inspector-Executor model proposed in [10] and its precursors is one of the first uses of the compiler to generate code which gathers runtime information about a program in order to parallelize it.

6. Conclusion

By exploiting inherent properties of applications, we have devised a novel parallelization scheme. Since this approach does not require extensive fine-grain communication and synchronization or complex hardware mechanisms, it can be fully implemented in software targeting shared memory multiprocessors. Our prototype system was able to obtain parallel speedups on few simple benchmarks. However, this scheme shows a huge potential to parallelize many large applications where compile time analysis is prohibitively expensive or impossible.

We plan to extend the capabilities of this system in many directions. Since it is sometimes difficult to obtain good performance with inner-loop (fine-grain) parallelism on multiprocessors with high communication latencies, outer-loop (coarse-grain) parallelism can be identified using a detector across inner loops. This involves a simple extension to the current detector. Secondly, it is also possible to eliminate loop-carried true dependences whose values are stride predictable. This technique may enable parallelization of algorithms that iterate over linked-lists. Thirdly, since this technique requires only simple local analyses, it could potentially be applied directly to executables. This will enable parallel execution of many sequential legacy applications. One limitation of this technique as it stands, is handling control-flow within the loop body. To expand the scope of the technique beyond loops containing straight-line code, we plan to employ a combination of profile-based prediction and invalidation techniques to handle many common cases of control-flow.

We believe that the Softspec techniques will be applicable to a large class of important applications that are currently unable to use automatic parallelization techniques. With the advent of single-chip multiprocessors with low latency communication, these applications will be able to obtain good parallel performance using Softspec.

7. Acknowledgements

I would like to express gratitude to individuals at MIT with whom I worked closely with on the Softspec project. My thesis advisor, Professor Saman Amarasinghe, was responsible for making my research experience at MIT an intense and exciting learning opportunity. Throughout my two years working with him, he has continually pushed me to improve my abilities in the field of computer systems and software, and do things that I never did before. Darin Petkov and George Peev, both undergraduate researchers at MIT played an important role in making this project a success, and worked hard with me before the PPOPP99 paper deadline.

I would like to thank Professor Anant Agarwal who invited me to join the Computer Architecture Group (CAG) at MIT when I didn't have any formal training in the field of computer systems, and all I had to show was my collage of research and hacking experience. My office mate, Matt Frank, was nothing short of an ideal mentor who patiently answered each of my novice questions, especially as I was dealing with the subtleties and annoyances of the SUIF library. I would like to thank my fellow CAG group members, particularly Matt Frank, John Babb, Rajeev Barua, and Walter Lee for the opportunities they gave me to partake in their research projects and learn several important computer systems techniques.

I'd also like to thank Phalgun Raju, John Hull, Sawan Deshpande, Zubin Irani, and Gangadhar Konduri of the underground India-Club for the fun times I had while going as I was going through the pain of writing the PPOPP99 paper and this thesis.

8. References

[1] S. P. Amarasinghe, J. M. Anderson, M. S. Lam and C. W. Tseng. The SUIF Compiler for Scalable Parallel Machines. *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, February, 1995.

[2] 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. (b) http://polaris.cs.uiuc.edu/polaris/polaris.html

[3] R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, June, 1995.

[4] L. Hammond, M. Willey, K. Olukotun. Data Speculation Support for a Chip Multiprocessor. *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, October 1998.*

[5] J. G. Steffan, T. C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture, February 2-4, 1998, Las Vegas, Nevada.*

[6] A. Moshovos, S. E. Breach, T. N. Vijaykumar, G. S. Sohi. Dynamic Speculation and Synchroniozation of Data Dependencies. *24th International Symposium on Computer Architecture (ISCA-24), June 1997.*

[7] M Frank, C. A. Moritz, B. Greenwald, S. Amarasinghe, A. Agarwal. SUDS: Primitive Mechanisms for Memory Dependence Speculation. Submitted to *26th International Symposium on Computer Architecture (ISCA-26), June 1999.*

[8] Y. Sazeides, J. E. Smith. The Predictability of Data Values. *Micro-30, December, 1997.*

[9] J. W. C. Fu, J. H. Patel. Data Prefetching in Multiprocessor Vector Cache Memories. *The 18th Annual International Symposium on Computer Architecture, pages 54-63, May 1991.*

[10] J. Gonzalez and A. Gonzalez. Speculative Execution Via Address Prediction and Data Prefetching. *11th International Conference on Supercomputing, July 1997*

[11] Shamik Sharma, Ravi Ponnusamy, Bongki Moon, Yuan-Shin Hwang, Raja Das, Joel Saltz. Run-time and Compile-time Support for Adaptive Irregular Problems. *Supercomputing 1994*

[12] M. Emami, R. Ghiya, L. Hendren. Context Sensitive interprocedural points-to analysis in the presence of function pointers. *SIGPLAN94, June 1994*

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

[14] L. Hamond, B. A. Nayfeh, and K. Olukotun. A Single-Chip Multiprocessor. *IEEE Computer, September 1997*

[15] Farid Khoury. Efficient Parallel Triangular Systems Solvers for Preconditioning Large Sparse Linear Systems. *Thesis, Applied Mathematics, University of New South Whales, November, 1994.* (http://server.srcpc.unsw.edu.au/ftp/papers/Khoury-thesis/thesis.html)

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

[17] http://math.nist.gov/MatrixMarket/data/NEP/h2plus/qc324.html

[18] http://math.nist.gov/MatrixMarket/data/NEP/h2plus/qc2534.html

[19] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng M. Hall, M. Lam, and J. Hennessy. SUIF: A Parallelizing and Optimizing Research Compiler. *ACM SIGPLAN Notices, 29(12):31-37, December 1994.*

[20] R. C. Young, Harvard HUBE Research Group, haltsuif-1.1.2-beta. http://www.eecs.harvard.edu/hube/#Software

[21] B. Nichols, D. Buttlar, J.P. Farrell. Pthreads Programming. *O'Rielly, Sebastopol, Ca, 1996*

[22] http://suif.stanford.edu/suif/suif1/

Appendix A

In addition to the gcd of two integers a and b, the calculation for detection of parallelism requires a solution to the diophantine equation **x*a + y*b = 1**. We have found that the largest percentage of execution time in the detection loop (see section 2.2) is in the procedure that calculates the gcd, **(a,b)**, and solution to the equation. Therefore we have investigated alternative algorithms that can speed up the calculation of the gcd, as well as an approximate algorithm for detection that avoids the gcd calculation altogether.

The first method we studied used a computational version of Euclid's gcd algorithm as presented in [16, page 301]. Another related algorithm, called the Continued Fraction GCD algorithm, involves converting the fraction **a/b** into a continued fraction. Psudeo-code is for this algorithm is shown below the continued fraction coefficients are obtained in an array of integers **c(n)**:

```
1. let $n = 1$
2. set $c(n) = 0$
3. while $a > b$, increase $c(n)$ by $b$ and subtract $b$ from $a$.
4. if $a \neq b$ then increase $n$ by $1$, set $c(n) = 0$
5. while $b > a$ increase $c(n)$ and subtract a from $b$
6. if $a \neq b$ then increase $n$ by $1$
7. goto step 2
```

Observe, that in steps 2 and 3 we could replace the subtractions with one division. But, since the Alpha processor does not have an integer division instruction, and, being a RISC processor, has a very fast subtract, it turned out that this particular implementation is faster. After this algorithm finishes, we have the coefficients of the continued fraction in **c(n)** and **(a,b).** Then we ignore the last term of the obtained continued fraction and calculate the fraction that corresponds to it, namely **p/q.** Then **x = -q** and **y = p.**

The second method we studied to speed up detection avoids lengthy integer calculations involving the gcd. However it is an approximate solution and always provides a conservative estimate for the number of parallel iterations in the loop. Therefore, in some cases it may turn out that this approximate algorithm will detect less parallelism than there really is. However in our experience, it suffices in most cases.

To restate the detection problem, we are given ordered pairs of initial addresses and strides, **(a,b)** and **(c,d)**,and we are to find the minimum of over all pairs **(m,n)** that satisfy **a*m + b = c*n + d**. We can visualize two lines in **(x,y)** space, namely **a*x + b = y** and **c*x + d = y**. Without loss of generality, let **b**>**d**, that is, the line **a*x + b = y** intersects the y-axis higher than **c*x + d = y**. Then we distinguish between the following cases:

1. **a** ≥ **0, c > 0** : first line is horizontal or going up, second line is going up. Then **t=(b-d)/c** gives us the number of the iteration when the second line will reach the y-axis intercept of the first line (that is, the second write operation will have the same address as the first address of the first write operation). We can parallelize up to t iterations.
2. **a = c = 0 -** both lines are horizontal.
3. **a** ≥ **0, c < 0 -** first line is horizontal or going up, second line is going down. Then they don't intersect at all. We can parallelize all iterations.
4. **a < 0, c** ≤ **0 -** first line is going down, second line is horizontal or going down. Then **t=(b-d)/(-a)** is the number of the iteration when the first line will reach the y-intercept of the second line (that is, the first write operation will have the same

address as the first address of the second write operation). We can parallelize up to t iterations.

5. **a < 0, c > 0 -** first line is going down, second line is going up. Then **t=(b-d)/(c-a)** is the number of the iteration when both lines will intersect. We can parallelize up to t iterations.

We came up with some minor improvements for certain special cases. In case 1. if the value of **t**, computed is equal to 0, which is the case when b=d, then we can actually parallelize up to **max{a, c}/(a,c)** iterations. Analogously, if t=0 in step 4, then we can parallelize up to **max{-a, -c}/(a,c)**. Those two cases occur when the two write operations start from the same address. If their deltas (the linear differences of two consequent addresses) are different, then they really occur when writing memory address **(a,c)**, the first operation writes there in iteration **a/(a,c)** and the second - in iteration **c/(a,c)**. The first write to that address is not problematic, but the second is, so we take the bigger of those two numbers for the maximal number of parallelizable iterations.

We have provided three different algorithms for the detect stage. The first uses *Euclid's* gcd algorithm, the second uses the *continued fraction* gcd algorithm. And third, we presented an algorithm that does not involve gcd, which we refer to as "*extrapolation*.." To provide a comparison of the three algorithms, we counted the cycles taken on our Alpha machine for two problems of differing sizes. The first problem is for a detection involving three reads for six writes, taken from a loop in the Spec92 tomcatv benchmark. The second problem is for 300 writes and 300 reads, all randomly generated. The (averaged) cycles counts for these cases are given in the table below.

	Euclid’s	Continued Fraction	Extrapolation
3 reads x 6 writes	4277	3820 (11%)	1932 (55%)
300 reads x 300 writes	68194	54201 (21%)	24322 (64%)

The percentages in brackets shown for the continued fraction and extrapolation methods are the percent improvement over Euclid’s algorithm. From these results, we conclude expect that the continued fraction gcd algorithm can provide a modest improvement in the range 10-20%. The extrapolation algorithm provides over 50% improvement over Euclid’s algorithm, but it offers only a conservative estimate of the number of parallel iterations.

Appendix B

This section contains source code in SUIF/C++ for the compiler. The header file is duds.h, and the source files are: main.cc, duds_init.cc, insert_code.cc, generate_arg_code.cc, generate_profile_code.cc, generate_detect_code.cc, generate_slave_code.cc.

duds.h

```
/** Softspec Parallelizing Compiler */
/** Author: Devabhaktuni Srikrishna, 1998 */
/** MIT Lab for Computer Science */

/** This program takes a sequential C program as input and
 *  generates (parallel) C code to be run on a shared memory
 *  multiprocessor. The resulting code must be linked with
 *  the Softspec runtime library.
 *  Please refer to the Softspec Paper for further details.
 *  The program uses the basic SUIF library.
 *  Note that the program only parallelizes inner loops
 *  appearing within the main procedure, which are not
 *  embedde within sub-procedures. However inner loops
 *  within sub-procedures may also be parallelized
 *  with a little more work.
 */

/** The following passes must be run in order on the input foo.c before
    running this program

    (1) scc -V -Target alpha-dec-osf -.spd foo.c
        (produces foo.spd)
        this converts the file to SUIF forma. The -Target
        flag is needed so that the program may be compiled
        to a Digital Alpha machine. Otherwise you will have
        problems compiling later on.
    (2) porky -Dmemcpys foo.spd foo.no_memcpy
        this pass dismantles memcpy instructions into
        lod and str instructions. Only lod and str instructions are
        alowed by the compiler, only for convenience in implementation.
        Alternatively, if there are no memcpy instructions in the
        input progra, then there is no need to run thsi pass.
        The -Dmemcpys option is not a stanrad porky option,
        and Walter added it as part of the rawcc version of porky.
     (3) moo -Psce foo.no_memcpy foo.moo
        By running this pass, local variables appearing in
        loop bodies are analysed for whether they are
        read, written, and can be privatized. This information
        is neede by the compiler to detect whether these local
        variables lead to loop carried dependencies. If they do,
        then the loop cannot be parallelized by the current
        implementation of softspec. moo -Psce writes three annotations
        to each loop: read_vars, write_vars, and privatizable.
        These annotations are needed by the compiler to decide whether
        the loop may be parallelized or not.

  Therefor the compiler is run on the file foo.moo.

 */


// Standard C include files
#include <stdlib.h>
#include <iostream.h>
#include <stdio.h>
// for SUIF to work
#include <suif.h>
// for SUIF builder
#include <builder.h>
// for SUIF useful library
#include <useful.h>
// for stuff in RAW useful library
#include <raw_useful.h>

#define DEBUG 1

#define NEED_MEMORY_BARRIER
#define DOUBLE_TO_CHAR 8

#define NUM_PAR_ITERS 1000
#define MAX_NUM_MEM_REFS 50
#define MAX_ARGS 50
#define NUM_PROF_ITERS 3

struct instruction_count {

        int local_count;
        int read_count;
        int write_double_count;

};

struct investigate {

        boolean Parallelizable;
        file_set_entry *fse;


};


// annotations

extern char *k_read_vars;
extern char *k_write_vars;
extern char *k_privatizable;
extern char *k_args_list;


// initialized in duds_init.h
extern ptr_type        *ptGenericFunc;
extern proc_sym        *ps_spec_init;
extern proc_sym        *ps_spec_quit;
extern proc_sym        *ps_memory_barrier;
extern func_type       *ftGenerated;
extern ptr_type        *ptGenerated;
extern proc_sym        *ps_profile_loop;
extern proc_sym        *ps_detect_loop;
extern proc_sym        *ps_remainder_loop;
extern proc_sym        *ps_spec_try;
extern func_type       *ftSlave;
extern ptr_type        *ptSlave;
extern proc_sym        *ps_slave;
extern ptr_type        *ptSpecifics;
extern proc_sym        *ps_pass_loop_specifics;

// variable corresponding to those in the runtime library
extern var_sym         *detect_flag_vs;
extern var_sym         *read_addr_prof_vs;
extern var_sym         *write_double_addr_prof_vs;
extern var_sym         *read_deltas_vs;
extern var_sym         *write_double_deltas_vs;
extern var_sym         *num_par_iters_vs;
extern var_sym         *int_args_vs;
extern var_sym         *double_args_vs;
extern var_sym         *pointer_args_vs;
extern var_sym        *inner_start_vs;

/* duds_init.cc */
void init_syms(void);

/* insert_code.cc */
void insert_spec_init_and_quit(tree_node_list *main_tnl);
void find_inner_loops(tree_node *tn, void *x);

/* generate_arg_code.cc */
void get_args(tree_for *tf, int master_slave);
immed_list *set_args(tree_for *tf);

/* generate_profile_code.cc */
tree_proc *create_profile_loop(file_set_entry *fse, tree_for *tf);

/* generate_detect_code.cc */
tree_proc *create_detect_loop(file_set_entry *fse, tree_for *tf);

/* generate_slave_code.cc */
tree_proc *create_slave_loop(
file_set_entry *fse,
tree_for *tf,
instruction_count *s);
```

main.cc:

```
#include <duds.h>

/* this procedure calls find_inner_loops on the body of the tree_proc tp on to locate
 * inner loops and parallelize them (if possible)
 */
void parallelize_main(tree_proc *tp)
{
       // set builder to work with the current procedure
        block::set_proc(tp);

       // there is only one file set in this program
       // and it needs to be accessed by code generation
       // passes in other files
        fileset->reset_iter();
        file_set_entry *fse = fileset->next_file();

       // the "investigate" data structure is used to pass certain
       // values between procedure calls
       // one of these values is fse
        investigate *Inv = (investigate *) malloc(sizeof(investigate));
        Inv->fse = fse;

       // this line applies find_inner_loops to all nodes
       // in the body of the procedure
       // it is depth first
        tp->body()->map(find_inner_loops, (void *) Inv, FALSE, FALSE);

       // this procedure inserts calls to initialization and
       // finalization procedures from the runtime library
       // at the very beginning and end of the procedure
       // body (tp->body()).
        insert_spec_init_and_quit(tp->body());

       // this is necessary to ask SUIF to re-number the
       // instructions so they have unique ids.
        tp->number_instrs();

}

/* When applied to a procedure (tp), it applies
 * "parallelize_main" if the name of the procedure is
 * "main"
 */
void do_proc(tree_proc * tp)
{
       block::set_proc(tp);

       proc_sym *cur_psym = tp->proc();

       /* name of main procedure */
       char *mainname = "main";
       if (strcmp(cur_psym->name(), mainname) == 0)
               parallelize_main(tp);

       return;

}


/* Writes the procedures generated by the program into
 * the output file. Rajeev Barua showed me how to do
 * this. Don't ask me how it works.
 */

static void write_back() {
       fileset->reset_iter();
       file_set_entry *fse;

       while ((fse = fileset->next_file())) {

               fse->reset_proc_iter();
               proc_sym *ps;

               while ((ps = fse->next_proc())) {
                       if (ps->is_in_memory())
                        {
                                ps->write_proc(fse);
                        }
                }
        }
}


/* This is the main procedure of this program.
 * It parses the input file, generates parallel code,
 * and writes the generated code back to suif files.
 */

main(int argc, char *argv[])
{
        // routine from SUIF library
        start_suif(argc, argv);

        // routine from raw_useful.cc
        // it reads the files into memory
        // the routine provided in the SUIF
        // library is buggy
        my_suif_read_files(argc, argv, TRUE);

        // creates the symbols that will be used
        // in the generated code including procedures
        // and global variables
        init_syms();

        // routine from raw_useful.cc
        // iterates over all procedures in the
        // input program and applies "do_proc" to
        // the main procedure.
        // do_proc looks for parallelizable inner
        // loops and generates parallel code in an new procedure
        my_suif_proc_iter(do_proc, FALSE, FALSE, TRUE);

        // writes back the generated code from memory to a file
        write_back();

        // routine from standard library
        // does not exit this program
        exit_suif();

        // exits from the program
        exit(0);

} // ** main ** //
```

duds_init.cc:

```
#include <duds.h>

/* This file contains initializations of various SUIF variables
 * which are used to generate code. They should be consistent with
 * the runtime library.
 */

/* these are annotations output by a previously
 * run porky pass
 */
char *k_read_vars;
char *k_write_vars;
char *k_privatizable;
/* this is an annotation created and used by this pass */
```

```
char *k_args_list;

/* these are the pointers to procedures and types generated for use by
 * this program
 */
ptr_type        *ptGenericFunc;              // void (*foo)(void)
proc_sym       *ps_spec_init;                // void spec_init(void)
proc_sym       *ps_spec_quit;                // void spec_quit(void)
#ifdef NEED_MEMORY_BARRIER
proc_sym       *ps_memory_barrier;           // void memory_barrier(void);
#endif


func_type      *ftGenerated;                 // int (*foo)(int, int, int)
ptr_type       *ptGenerated;                 // int (*foo)(int, int, int)
proc_sym       *ps_profile_loop;             // int profile_loop(int, int, int)
proc_sym       *ps_detect_loop;              // int detect_loop(int, int, int)
proc_sym       *ps_remainder_loop;           // int remainder_loop(int, int, int)
proc_sym       *ps_spec_try;                 // int spec_try(int, int, int)

func_type      *ftSlave;                     // int (*foo)(int, int, int, double *)
ptr_type       *ptSlave;                     // int (*foo)(int, int, int, double *)
proc_sym       *ps_slave;                    // int (-slave-)(int, int, int, double *)

func_type      *ftSpecifics;         // void (*foo)(ptGenerated, int, int, int)
ptr_type       *ptSpecifics;
proc_sym       *ps_pass_loop_specifics;

/* From lib.h must be consistent */

/* lib.detect.c variables */
var_sym        *detect_flag_vs;              // int detect_flag
var_sym        *read_addr_prof_vs;           // unsigned long**read_addr_prof
var_sym        *write_double_addr_prof_vs;   // unsigned long **write_double_addr_prof
var_sym        *read_deltas_vs;                      // long*read_deltas
var_sym        *write_double_deltas_vs;              // long *write_double_deltas
var_sym        *num_par_iters_vs;                    // int num_par_iters

/* lib.master.c variables */
var_sym        *int_args_vs;                 // int *int_args
var_sym        *double_args_vs;              // double *double_args
var_sym        *pointer_args_vs;             // void **pointer_args
var_sym        *inner_start_vs;              // int inner_start

/* this procedure initialize annotations */
void annote_init() {

       ANNOTE(k_read_vars, "read vars", TRUE);
       ANNOTE(k_write_vars, "write vars", TRUE);
       ANNOTE(k_privatizable, "privatizable", TRUE);
       ANNOTE(k_args_list, "args list", TRUE);

}

/* the following procedures beginning with "init" initialize the
 * various global variables and function pointers
 */
void init_global_variables(void) {

       /* from lib.detect.c */

       detect_flag_vs = new var_sym(type_signed, "detect_flag");
       fileset->globals()->add_sym(detect_flag_vs);

       type_node *aptq = new array_type(type_unsigned_long, 0, MAX_NUM_MEM_REFS-1);
       fileset->globals()->add_type(aptq);
       type_node *apt = new array_type(aptq, 0, NUM_PROF_ITERS-1);
       fileset->globals()->add_type(apt);
```

```
        read_addr_prof_vs = new var_sym(
                apt,
                "read_addr_prof");
        fileset->globals()->add_sym(read_addr_prof_vs);

        write_double_addr_prof_vs = new var_sym(
                apt,
                "write_double_addr_prof");
        fileset->globals()->add_sym(write_double_addr_prof_vs);

        type_node *delat = new array_type(type_signed_long, 0, MAX_NUM_MEM_REFS-1);
        fileset->globals()->add_type(delat);
        read_deltas_vs = new var_sym(
                delat,
                "read_deltas");
        fileset->globals()->add_sym(read_deltas_vs);

        write_double_deltas_vs = new var_sym(
                delat,
                "write_double_deltas");
        fileset->globals()->add_sym(write_double_deltas_vs);

        /* from lib.master.c */

        type_node *iat = new array_type(type_signed, 0, MAX_ARGS-1);
        fileset->globals()->add_type(iat);
        int_args_vs = new var_sym(
                iat,
                "int_args");
        fileset->globals()->add_sym(int_args_vs);

        type_node *dat = new array_type(type_double, 0, MAX_ARGS-1);
        fileset->globals()->add_type(dat);

        double_args_vs = new var_sym(
                dat,
                "double_args");
        fileset->globals()->add_sym(double_args_vs);

        type_node *pat = new array_type(type_void->ptr_to(), 0, MAX_ARGS-1);
        fileset->globals()->add_type(pat);

        pointer_args_vs = new var_sym(
                pat,
                "pointer_args");
        fileset->globals()->add_sym(pointer_args_vs);

        inner_start_vs = new var_sym(
                type_signed,
                "inner_start");
        fileset->globals()->add_sym(inner_start_vs);

        return;
}

void init_generic_func_syms(void)
{

        // void (*foo)(void)
        func_type *ft = new func_type(type_v0);
        ft = (func_type *)fileset->globals()->install_type(ft);
        ptGenericFunc = new ptr_type(ft);
        ptGenericFunc = (ptr_type *) fileset->globals()->install_type(ptGenericFunc);
        ps_spec_init = fileset->globals()->new_proc(ft, src_c, "spec_init");
        ps_spec_quit = fileset->globals()->new_proc(ft, src_c, "spec_quit");
#ifdef NEED_MEMORY_BARRIER
        ps_memory_barrier = fileset->globals()->new_proc(ft, src_c, "memory_barrier");
#endif
```

```
}


void init_generated_func_types()
{
        ftGenerated = new func_type(
                      type_signed, type_signed, type_signed, type_signed);
        ftGenerated = (func_type *)fileset->globals()->install_type(ftGenerated);
        ptGenerated = new ptr_type(ftGenerated);
        ptGenerated = (ptr_type *) fileset->globals()->install_type(ptGenerated);
        ps_spec_try = fileset->globals()->new_proc(ftGenerated, src_c,
                                                          "spec_try");

        ftSlave = new func_type(
                      type_signed,  type_signed,  type_signed,  type_signed,  type_double-
>ptr_to());
        ftSlave = (func_type *)fileset->globals()->install_type(ftSlave);
        ptSlave = new ptr_type(ftSlave);
        ptSlave = (ptr_type *) fileset->globals()->install_type(ptSlave);

}

void init_param_func_syms(void)
{


        {
          func_type *ftSpecifics = new func_type(
                      type_v0, ptSlave, type_signed,
                      type_signed, type_signed);
          ftSpecifics = (func_type *)fileset->globals()->install_type(ftSpecifics);
          ptSpecifics = new ptr_type(ftSpecifics);
          ptSpecifics = (ptr_type *) fileset->globals()->install_type(ptSpecifics);
          ps_pass_loop_specifics = fileset->globals()->new_proc(
                                       ftSpecifics, src_c, "pass_loop_specifics");
        }

        return;

}

/* This procedure calls the other init procedures.
 * It is called from main
 */
void init_syms(void)
{

        annote_init();

        init_global_variables();

        init_generic_func_syms();

        init_generated_func_types();

        init_param_func_syms();

        return;

}
```

insert_code.cc

```
#include <duds.h>
```

```
/* This procedure inserts calls to the generated procedures into the
 * body of the main procedure. The instruction_count structure
 */
void insert_into_main(
tree_for *tf,
tree_proc *tp_profile,
tree_proc *tp_detect,
tree_proc *tp_slave,
instruction_count *s)
{

        // the original for loop is replaced by the profile, detect,
        // slave procedures and the remainder loop.

        // First, we have to generate some preliminary code

        // record lower bound of the loop in "start"
        // code generated: start = lower bound of for loop
        var_sym *start_vs = tf->scope()->new_var(type_signed, "start");
        instruction *start_set = new in_rrr(io_cpy,
                              type_signed,
                              operand(start_vs),
                              tf->lb_op().clone(tf->scope()));
        tree_instr *start_ti = new tree_instr(start_set);
        tf->parent()->insert_before(start_ti, tf->list_e());

        // set the index variable to the lower bound as well
        // code generated: index = lower bound of for loop
        instruction *lb_set = new in_rrr(io_cpy,
                              tf->index()->type(),
                              operand(tf->index()),
                              tf->lb_op().clone(tf->scope()));
        tree_instr *lb_ti = new tree_instr(lb_set);
        tf->parent()->insert_before(lb_ti, tf->list_e());

        // record step of the loop in "step" variable
        // code generated: step = step of loop
        var_sym *step_vs = tf->scope()->new_var(type_signed, "step");
        instruction *step_set = new in_rrr(io_cpy,
                              type_signed,
                              operand(step_vs),
                              tf->step_op().clone(tf->scope()));
        tree_instr *step_ti = new tree_instr(step_set);
        tf->parent()->insert_before(step_ti, tf->list_e());

        // record upper bound of loop in "ub" variable
        // code generated: ub =
        var_sym *ub_vs = tf->scope()->new_var(type_signed, "ub");
        instruction *ub_set = new in_rrr(io_sub,
                              type_signed,
                              operand(ub_vs),
                              tf->ub_op().clone(tf->scope()),
                              operand(step_vs));
        tree_instr *ub_ti = new tree_instr(ub_set);
        tf->parent()->insert_before(ub_ti, tf->list_e());

        // pass_loop_specifics communicates the several values
        // via shared memory to the runtime system.
        // these values include the loop bounds, the step,
        // the number of reads and writes in the loop, etc.
        // therefore it is inserted into the code before calls to
        // other procedures

        instruction *pass_specifics = new in_ldc(ptSpecifics,
               operand(), immed(ps_pass_loop_specifics));
        instruction *slave_ptr = new in_ldc(ptSlave,
                              operand(), immed(tp_slave->proc()));
        instruction *num_reads = new in_ldc(type_signed,
                              operand(), immed(s->read_count));
        instruction *num_write_doubles = new in_ldc(type_signed,
                              operand(), immed(s->write_double_count));

        instruction *call_specifics = new in_cal(type_v0, operand(),
                              operand(pass_specifics), operand(slave_ptr),
                              operand(num_reads),
                              operand(num_write_doubles),
                              operand(start_vs));
        tree_instr *call_specifics_ti = new tree_instr(call_specifics);
        tf->parent()->insert_before(call_specifics_ti, tf->list_e());

        // memory barrier is needed to make sure all values written
        // to shared memory are propogated visible to the parallel
        // threads running on different processors.

        instruction *mb = new in_ldc(ptGenericFunc, operand(),
                              immed(ps_memory_barrier));
        instruction *mb_in = new in_cal(type_v0, operand(),
                              operand(mb));
        tree_instr *mb_ti = new tree_instr(mb_in);
        tf->parent()->insert_before(mb_ti, tf->list_e());

        // code generated: j = profile_loop
        instruction *profile_loop = new in_ldc(ptGenerated,
               operand(), immed(tp_profile->proc()));
        instruction *profile_loop_in = new in_cal(type_signed,
                      operand(tf->index()), operand(profile_loop),
                      operand(tf->index()), operand(step_vs),
                      operand(ub_vs));
        tree_instr *profile_loop_ti = new tree_instr(profile_loop_in);
        tf->parent()->insert_before(profile_loop_ti, tf->list_e());

        // code generated: j = detect_loop(j)
        instruction *detect_loop = new in_ldc(ptGenerated,
               operand(), immed(tp_detect->proc()));
        instruction *detect_loop_in = new in_cal(type_signed,
                      operand(tf->index()), operand(detect_loop),
                      operand(tf->index()), operand(step_vs),
                      operand(ub_vs));
        tree_instr *detect_loop_ti = new tree_instr(detect_loop_in);
        tf->parent()->insert_before(detect_loop_ti, tf->list_e());

        // code generated: j = spec_try()
        instruction *spec_loop = new in_ldc(ptGenerated,
               operand(), immed(ps_spec_try));
        instruction *spec_loop_in = new in_cal(type_signed,
                      operand(tf->index()), operand(spec_loop),
                      operand(tf->index()), operand(step_vs),
                       operand(ub_vs));
        tree_instr *spec_loop_ti = new tree_instr(spec_loop_in);
        tf->parent()->insert_before(spec_loop_ti, tf->list_e());

        // keep the original loop and make it the remiander loop
        // by replacing the lower bound with the latest value of
        // the index.
        tf->lb_op().remove();
        tf->set_lb_op(operand(tf->index()));

        return;
}

/* This procedure determines wheter a loop may be paralleized by
 * the sofspec compiler.
 * assumes:    (1) that tf is an inner loop
 *             (2) the SUIF pass "moo -Psce" has been run on the input program
 *                 and therefore the read_vars, write_vars, and provatizable
 *                 annotations have been written on this for loop
 * returns: true or false depending on the result of analysis.
 */
boolean is_parallelizable(tree_for *tf)
{

        // get the annotations
        immed_list *read_vars_iml
                = (immed_list *) tf->annotes()->peek_annote(k_read_vars)->immeds();
        immed_list *write_vars_iml
                = (immed_list *) tf->annotes()->peek_annote(k_write_vars)->immeds();
        immed_list *privatizable_iml
                = (immed_list *) tf->annotes()->peek_annote(k_privatizable)->immeds();

        // if any of the annotations are not present,
        // we cannot performa analysis, so return FALSE
        if(     read_vars_iml == NULL
                || write_vars_iml == NULL
                || privatizable_iml == NULL)
                                        return FALSE;

        // if there are no variables written to,
        // then something is wrong. At least the
        // loop index variable should be written to
        if ( !(write_vars_iml->count() > 0) ) {
                return FALSE;
        }

        // if the first variable written to is not
        // a symbol, then return FALSE
        if( !(((*write_vars_iml)[0]).is_symbol()) ) {
                return FALSE;
        }

        // if the first variable written to is not the
        // index variable, then return FALSE
        // the index variable usually appears as the
        // first variable written to
        if(  (((*write_vars_iml)[0]).symbol())->sym_id()
                != tf->index()->sym_id()  )
                return FALSE;


        // condition for no loop carried dependencies:
        // the write vars list is identical to the
        // privitizable vars list except for the
        // presence of the loop index in ths write
        // vars list.
        // the following code checks for this
        // condition.
        int write_count = write_vars_iml->count();
        int privatizable_count = privatizable_iml->count();

        for(int i=1; i < write_count; i++) {

                if(i-1 >= privatizable_count)
                        return FALSE;

                if( (*privatizable_iml)[i-1]
                        != (*write_vars_iml)[i] )
                                return FALSE;

        }

        // if all the above tests pass, return TRUE

        return TRUE;
}
```

```
/* Given a tree_node tn, this procedure finds all for loops which
 * contain straight line loop bodies, and then replaces them with parallel
 * versions. The search is depth-first.
 */

void find_inner_loops(tree_node *tn, void *x)
{

        // the number of parallellizable loops found so far
        static int num_par_loops=0;

        // a data structure to passed between procedures
        // see parallelize_main in main.cc
        investigate *Inv = (investigate *) x;

        // find_inner_loops is recursively applied to all nodes
        // in the body of the tree_node (tn) to identify inner
        // loops.

        if( tn->is_for() ) {

                // if the node is a for loop, it could be an
                // inner loop, or its body could contain a loop.
                // therefore it is necessary to determine which
                // is the case, and parallelize only inner loops.

                tree_for *tf = (tree_for *) tn;

                // the data structure Inv is also used to
                // determine which loop in a loop nest is
                // the inner loop, and hence parallelizable.
                // the Parallelizable flag is set initially to
                // true and then it is subsequently set to
                // false by recursive calls to find_inner_loops,
                // if the for loop contains control flow or loop themselves.

                Inv->Parallelizable = TRUE;

                // recursive call to fine_inner_loops to determine
                // whether the current for loop (tf) is an inner loop

                tf->body()->map(find_inner_loops, x, FALSE, FALSE);

                // if tf is not an inner loop, then do not try to parallelize
                if(Inv->Parallelizable == FALSE) {
                        return;
                }

                // check to see if there are any loop carried
                // dependencies of local variables, precluding
                // parallelization with the current softspec
                // implementation.
                if( !is_parallelizable(tf) ) {
                        Inv->Parallelizable == FALSE;
                        return;
                }

                // now generate parallel code

                // first get the list of local variables whose
                // current values need to be passed in to
                // the context of the profile, detect, and
                // slave procedures.
                immed_list *args_iml = set_args(tf);
                tf->prepend_annote(k_args_list, args_iml);

                // the following generate procedures automatically
                // install the procedures they generate into the
        // code. Calls to these procedures are inserted
        // into main by insert_into_main below.

        //generate profile code
        tree_proc *tp_profile = create_profile_loop(Inv->fse, tf);

        // generate detect loopby insert_into_main
        tree_proc *tp_detect
               = create_detect_loop(Inv->fse, tf);

        // this is a structure which keeps track of the
        // number of reads and writes in the body of the
        // loop, and is used to pass these numbers to the
        // runtime system. The number of reads and writes
        // are determined in the code which generates
        // the slave code.

        instruction_count *s = new instruction_count;

        // generate the dlave code
        tree_proc *tp_slave
               = create_slave_loop(
                      Inv->fse,
                      tf,
                      s);

        // insert the generated procedures into
        // the main procedure
        insert_into_main(
                      tf,
                      tp_profile,
                      tp_detect,
                      tp_slave,
                      s);

        // now set the Parallelizable flag to FALSE to indicate
        // that any outer loops are not parallelizable
        Inv->Parallelizable = FALSE;

 } else if( tn->is_instr() )  {

        // if there are branches or procedures within the loop body,
        // we do not parallelize the loop. This is not a necessary
        // limitation. We could put some code here to insert
        // duds code within procedures.

        inst_format inf = ((tree_instr *) tn)->instr()->format();

        if(inf == inf_bj || inf == inf_cal || inf == inf_mbr)
        {
               Inv->Parallelizable = FALSE;
               return;
        }


 } else if( tn->is_block() ) {
        // if you encounter a block, then look inside the
        // block.
         tn->map(find_inner_loops, x, FALSE, FALSE);
 } else {
        // in all other cases, we say that the loop is not
        // parallelizable
        Inv->Parallelizable = FALSE;
 }

 return;

}
```

```
/* This procedure inserts procedures which initialize (spec_init)
 * and wrap up (spec_quit) the runtime system. The init procedure
 * starts threads for the detection and slave , and the quit
 * destroys these threads. Therefore the init procedure is
 * inserted at the beginning of the main procedure body, and
 * quit is inserted at the end of main
 */
void insert_spec_init_and_quit(tree_node_list *main_tnl)
{

        /* spec_init */
        instruction *init = new in_ldc(ptGenericFunc, operand(),
                                  immed(ps_spec_init));
        instruction *init_in = new in_cal(type_v0, operand(),
                                  operand(init));
        tree_instr *init_ti = new tree_instr(init_in);
        main_tnl->insert_before(init_ti, main_tnl->head());


        /* spec_quit */
        instruction *quit = new in_ldc(ptGenericFunc, operand(),
                                  immed(ps_spec_quit));
        instruction *quit_in = new in_cal(type_v0, operand(),
                                  operand(quit));
        tree_instr *quit_ti = new tree_instr(quit_in);
        main_tnl->insert_before(quit_ti, main_tnl->tail());


        return;
}
```

generate_arg_code.cc:

```
#include <duds.h>

void get_args(tree_for *tf, int master_slave)
{

        immed_list *args_iml = tf->annotes()->peek_annote(k_args_list)->immeds();

        int args_count = args_iml->count();

        {
          int int_count=0;
          int double_count=0;
          int pointer_count=0;

          for(int i=0; i<args_count; i++) {

            sym_node *sn = ((*args_iml)[i]).symbol();

            if( sn->is_var() ) {

                var_sym *vs = (var_sym *) sn;

                if( vs != tf->index() ) {

                  type_node *vst = vs->type();

                  if( vst == type_signed ) {

                       //instruction *init = new in_ldc(ptSetInt, operand(),
                         //       immed(ps_set_master_int_args));
```

```
               //instruction *arg_num = new in_ldc(type_signed,
               //             operand(), immed(int_count));
               //instruction *init_in = new in_cal(type_v0, operand(),
                 //        operand(init), operand(arg_num), operand(vs));
               // tree_instr *init_ti = new tree_instr(init_in);

               block int_args_bl(int_args_vs);
               block arg_num_bl(int_count);
               block arg_var_bl(vs);
               block assign(arg_var_bl = int_args_bl[arg_num_bl]);
               tree_node *init_tn = assign.make_tree_node();
               tf->parent()->insert_before(init_tn, tf->list_e());

               int_count++;

          } else if (vst == type_double) {

               //instruction *init = new in_ldc(ptSetFloat, operand(),
                 //        immed(ps_set_master_double_args));
               //instruction *arg_num = new in_ldc(type_signed,
               //             operand(), immed(double_count));
               //instruction *init_in = new in_cal(type_v0, operand(),
                 //        operand(init), operand(arg_num), operand(vs));
               //tree_instr *init_ti = new tree_instr(init_in);
               //tf->parent()->insert_before(init_ti, tf->list_e());

               block double_args_bl(double_args_vs);
               block arg_num_bl(double_count);
               block arg_var_bl(vs);
               block assign(arg_var_bl = double_args_bl[arg_num_bl]);
               tree_node *init_tn = assign.make_tree_node();
               tf->parent()->insert_before(init_tn, tf->list_e());

               double_count++;

          } else if( vst == type_double->ptr_to()
               || vst == type_signed->ptr_to() )  {

               //instruction *init = new in_ldc(
               //      ptSetPointer, operand(),
                 //        immed(ps_set_master_pointer_args));
               //instruction *arg_num = new in_ldc(type_signed,
               //      operand(), immed(pointer_count));
               //instruction *init_in = new in_cal(
               //      type_v0, operand(),
               //      operand(init), operand(arg_num),
               //      operand(vs));
               //tree_instr *init_ti = new tree_instr(init_in);

               block pointer_args_bl(pointer_args_vs);
               block arg_num_bl(pointer_count);
               block arg_var_bl(vs);
               block assign(arg_var_bl = pointer_args_bl[arg_num_bl]);
               tree_node *init_tn = assign.make_tree_node();

               tf->parent()->insert_before(init_tn, tf->list_e());

               pointer_count++;

          }
        }

    }

  }
}

return;
```

```
}

boolean immed_is_in_list(sym_node *s, immed_list *iml)
{

        if(iml == NULL)
                return FALSE;

        int iml_count = iml->count();

        for(int i=0; i < iml_count; i++) {

                assert( ((*iml)[i]).is_symbol() );
                if(s == ((*iml)[i]).symbol() )
                        return TRUE;

        }

        return FALSE;

}


immed_list *set_args(tree_for *tf)
{

       immed_list *read_vars_iml
              = (immed_list *) tf->annotes()->peek_annote(k_read_vars)->immeds() ;
       immed_list *privatizable_iml
              = (immed_list *) tf->annotes()->peek_annote(k_privatizable)->immeds() ;

       if(read_vars_iml == NULL)
                     return NULL;

       int read_count = read_vars_iml->count();

       immed_list *args_iml = new immed_list;

       for(int i=0; i < read_count; i++) {

              assert( ((*read_vars_iml)[i]).is_symbol() );

              if( !immed_is_in_list(
                     ((*read_vars_iml)[i]).symbol(),
                     privatizable_iml)
                     ) {

                     sym_node *sn = ((*read_vars_iml)[i]).symbol();
                     args_iml->append(immed(sn));

              }
       }


       {
         int int_count=0;
         int double_count=0;
         int pointer_count=0;
         int args_count = args_iml->count();

         for(int i=0; i<args_count; i++) {

           sym_node *sn = ((*args_iml)[i]).symbol();

           if( sn->is_var() ) {

               var_sym *vs = (var_sym *) sn;
```

```
if( vs != tf->index() ) {

  type_node *vst = vs->type();

  if( vst == type_signed ) {

        //instruction *init = new in_ldc(ptSetInt, operand(),
          //       immed(ps_set_master_int_args));
        //instruction *arg_num = new in_ldc(type_signed,
        //            operand(), immed(int_count));
        //instruction *init_in = new in_cal(type_v0, operand(),
          //        operand(init), operand(arg_num), operand(vs));
        // tree_instr *init_ti = new tree_instr(init_in);

        block int_args_bl(int_args_vs);
        block arg_num_bl(int_count);
        block arg_var_bl(vs);
        block assign(int_args_bl[arg_num_bl] = arg_var_bl);
        tree_node *init_tn = assign.make_tree_node();
        tf->parent()->insert_before(init_tn, tf->list_e());

        int_count++;

  } else if (vst == type_double) {

        //instruction *init = new in_ldc(ptSetFloat, operand(),
          //        immed(ps_set_master_double_args));
        //instruction *arg_num = new in_ldc(type_signed,
        //            operand(), immed(double_count));
        //instruction *init_in = new in_cal(type_v0, operand(),
          //        operand(init), operand(arg_num), operand(vs));
        //tree_instr *init_ti = new tree_instr(init_in);
        //tf->parent()->insert_before(init_ti, tf->list_e());

        block double_args_bl(double_args_vs);
        block arg_num_bl(double_count);
        block arg_var_bl(vs);
        block assign(double_args_bl[arg_num_bl] = arg_var_bl);
        tree_node *init_tn = assign.make_tree_node();
        tf->parent()->insert_before(init_tn, tf->list_e());

        double_count++;

  } else if( vst == type_double->ptr_to()
        || vst == type_signed->ptr_to() )  {

        //instruction *init = new in_ldc(
        //      ptSetPointer, operand(),
          //        immed(ps_set_master_pointer_args));
        //instruction *arg_num = new in_ldc(type_signed,
        //      operand(), immed(pointer_count));
        //instruction *init_in = new in_cal(
        //      type_v0, operand(),
        //      operand(init), operand(arg_num),
        //      operand(vs));
        //tree_instr *init_ti = new tree_instr(init_in);

        block pointer_args_bl(pointer_args_vs);
        block arg_num_bl(pointer_count);
        block arg_var_bl(vs);
        block assign(pointer_args_bl[arg_num_bl] = arg_var_bl);
        tree_node *init_tn = assign.make_tree_node();

        tf->parent()->insert_before(init_tn, tf->list_e());

        pointer_count++;
```

```
                    }
                  }

              }

            }

            assert(int_count <= MAX_ARGS);
            assert(double_count <= MAX_ARGS);
            assert(pointer_count <= MAX_ARGS);
        }

        return args_iml;
}
```

generate_profile_code.cc

```
#include <duds.h>

/* This procedure inserts the address profiling code for an
 * instruction, lod or str.
 */
void instr_insert_profile(
instruction *in,
instruction_count *s,
var_sym *iter_count,
proc_symtab *pst_profile)
{

  // we do not handle memcpy instructions as of yet.
  // however they could be disassembled into lod and str
  // instructions
  assert( in->opcode() != io_memcpy );

  if(in->opcode() == io_lod
        || in->opcode() == io_str) {

        in_rrr *inls = (in_rrr *) in;

        // after running the following code,
        // vs should refer to the variable
        // that contains the address of the
        // lod/str
        var_sym *vs;

        if( inls->src1_op().is_instr() ) {

                // now detach the address expression and store
                // it in a "promote" variable
                instruction *tmp_ins = inls->src1_op().instr();
                type_node *tmp_typ = inls->src1_op().type();
                vs = new var_sym(tmp_typ, "promote");
                pst_profile->add_sym(vs);
                tmp_ins->remove();
                inls->src1_op().remove();
                inls->set_src1(vs);
                tmp_ins->set_dst(vs);

        } else if( inls->src1_op().is_symbol() ) {
                vs = (var_sym *) inls->src1_op().symbol();
        } else {
                assert(FALSE);
        }
        // next, insert code that stores the profiled address
        // into the shared array from the runtime system (read_addr_prof_vs)
        // this array is different depending on whether the dealing with
        // loads or stores
        block addr_array_bl;
        block ref_count_bl; // this is the index into the array
        if(in->opcode() == io_lod) {
                addr_array_bl.set(read_addr_prof_vs);
                ref_count_bl.set(s->read_count);
                s->read_count += 1;
                s->local_count++;
        } else {
                addr_array_bl.set(write_double_addr_prof_vs);
                ref_count_bl.set(s->write_double_count);
                s->write_double_count += 1;
                s->local_count++;
        }


        // now make the assignment of the address value to the
        // shared array
        instruction *cast = new in_rrr(io_cvt, type_unsigned_long,
                operand(),  operand(vs));
        block cast_bl(cast);
        block iter_count_bl(iter_count);
        block init_bl;
        init_bl.set(addr_array_bl[iter_count_bl][ref_count_bl] = cast_bl);
        tree_node *init_tn = init_bl.make_tree_node();
         in->parent()->parent()->insert_before(init_tn, in->parent()->list_e());

  }

  return;

}

/* This procedure inserts the instrumentation needed to
 * profile addresses appearing in loads/stores within the
 * loop body, for the profile loop.
 */

void insert_profile(
tree_node_list *tnl,
var_sym *iter_count,
proc_symtab *pst_profile)
{

        int length = tnl->count();

        // this data structure is needed to count the number of
        // reads and writes within the loop body
        struct instruction_count *s = new instruction_count;
        s->read_count = 0;
        s->write_double_count = 0;

        // apply instr_insert_profile to all instructions
        // within the loop body
        for(int i=0; i<tnl->count(); i++) {

                tree_node *tn = (*tnl)[i];
                if( tn->is_instr() ) {
                        s->local_count = 0;
                        tree_instr *ti = (tree_instr *) tn;
                        instr_insert_profile(
                                ti->instr(),
                                s,
                                iter_count,
                                pst_profile);
                        i+= s->local_count;
                }

        }

}

/* As the name of the procedure suggests ..
 */

tree_proc *create_profile_loop(file_set_entry *fse, tree_for *tf)
{


         // a variable that keeps track of the number of
         // detect procedures successfully generated, so
         // it can give them different names.
        static int count=0;
        char name[100];
        sprintf(name, "profile_loop%d", count);

         // create the procedure and install it into the
         // file symbol table
        proc_sym *ps_profile
                = fse->symtab()->new_proc(ftGenerated, src_c,
                        name);
        ps_profile->set_fse(fse);

         // create a procedure symbol table for the procedure
        proc_symtab *pst_profile = new proc_symtab(name);
        fse->symtab()->add_child(pst_profile);

         // create the arguments of the procedure, namely
         // starting index, the step, and the upper bound

        char input_name[100];

        // add first argument
        sprintf(input_name, "%s_input", tf->index()->name());
        var_sym* input_var = pst_profile->new_var(type_signed, input_name);
        input_var->set_param();
        pst_profile->params()->append(input_var);

        // add second argument
        sprintf(input_name, "%s_step", tf->index()->name());
        var_sym* step_var = pst_profile->new_var(type_signed, input_name);
        step_var->set_param();
        pst_profile->params()->append(step_var);

        // add third argument
        sprintf(input_name, "%s_stop", tf->index()->name());
        var_sym* stop_var = pst_profile->new_var(type_signed, input_name);
        stop_var->set_param();
        pst_profile->params()->append(stop_var);

        // make the body of the profile procedure
        tree_node_list *profile_body = new tree_node_list;

        // first insert a copy of the original loop itself
        tree_for *tfc = tf->clone(pst_profile);
        profile_body->push(tfc);

        // create tree_proc for  profile procedure, and install
         // the body into the tree_proc
        tree_proc *tp_profile = new tree_proc(profile_body, pst_profile);
        ps_profile->set_block(tp_profile);

        // set the current procedure for the builder library
         // using the following command
        block::set_proc(tp_profile);

         // set the lower bound, step, and upper bound of the
         // for loop to argument values
        (tfc->lb_op()).remove();
        tfc->set_lb_op(operand(input_var));
        (tfc->step_op()).remove();
        tfc->set_step_op(operand(step_var));
        (tfc->ub_op()).remove();
        tfc->set_ub_op(operand(stop_var));

         // the following call to get_args inserts
         // calls to data structures within the runtime
         // library to initialize local variables so they
         // have the same values as in the main procedure

        get_args(tfc, 1);

        // a counter is needed to keep track of the number of iterations
        var_sym *iter_count = pst_profile->new_var(type_signed, "iter_count");

         // this block of code genertes all the necessary
         // additional code needed in the body of the loop
         // to perform detection

        {


               // code generated: iter_count = 0
               instruction *iter_count_init = new in_ldc(type_signed,
                                         operand(iter_count), immed(0));
               tree_instr *iter_count_init_ti = new tree_instr(iter_count_init);
               tfc->parent()->insert_before(iter_count_init_ti, tfc->list_e());


               // install exit_loop label symbol
               label_sym *exit_loop_ls = pst_profile->new_label("exit_loop");
               in_lab *exit_loop_il = new in_lab(exit_loop_ls);
               tree_instr *exit_loop_ti =  new tree_instr(exit_loop_il);
               tfc->parent()->insert_after(exit_loop_ti, tfc->list_e());

               // make break condition at beginning of loop body
               // code generated:
               //      if(!(iter_count < NUM_PROF_ITERS)) jump exit_loop;

               instruction *three = new in_ldc(type_signed,
                              operand(), immed(NUM_PROF_ITERS));
               tree_instr *three_ti = new tree_instr(three);
               in_rrr *test = new in_rrr(io_sl, type_signed,
                       operand(), operand(iter_count), operand(three));
               tree_instr *test_ti = new tree_instr(test);
               in_bj *break_loop = new in_bj(io_bfalse,
                       exit_loop_ls, operand(test));
               tree_instr *break_loop_ti = new tree_instr(break_loop);


               // code generated: iter_count+=1
               instruction *one = new in_ldc(type_signed, operand(), immed(1));
               instruction *iter_count_incr = new in_rrr(io_add, type_signed,
                       operand(iter_count), operand(iter_count), operand(one));
               tree_instr *iter_count_incr_ti
                             = new tree_instr(iter_count_incr);


               // insert the generated into the loop body
               tfc->body()->insert_after(iter_count_incr_ti,
                                                     tfc->body()->tail());
```

```
                tfc->body()->insert_before(break_loop_ti, tfc->body()->head());
                tfc->body()->insert_before(test_ti, tfc->body()->head());
                tfc->body()->insert_before(three_ti, tfc->body()->head());

        }


        // this instruments all the loads and stores in the loop body
        // so that the addresses are made accessible to the runtime library
        insert_profile(tfc->body(), iter_count, pst_profile);

         // memory barrier, needed to make sure that the profiled addresses
        // are seen by the threads running on other processors.
         instruction *mb = new in_ldc(ptGenericFunc, operand(),
                                 immed(ps_memory_barrier));
         instruction *mb_in = new in_cal(type_v0, operand(),
                                 operand(mb));
         tree_instr *mb_ti = new tree_instr(mb_in);
         profile_body->append(mb_ti);

        // create return instruction
        in_rrr *in_ret = new in_rrr(
                io_ret, type_void, operand(), operand(tfc->index()));
        tree_instr *ti_ret = new tree_instr(in_ret);
        profile_body->append(ti_ret);

        // increment count
        count++;

        // number the instructions in this procedure, since new ones have been
        // created
        tp_profile->number_instrs();

        return tp_profile;

}
```

generate_detect_code.cc

```
#include <duds.h>

/* As the name of the procedure suggests ..
 */

tree_proc *create_detect_loop(file_set_entry *fse, tree_for *tf)
{

        // a variable that keeps track of the number of
        // profile loops successfully generated, so
        // it can give them different names.

        static int count=0;
        char name[100];
        sprintf(name, "detect_loop%d", count);

        // create the detect procedure and install it into the
        // file symbol table
        proc_sym *ps_detect
                = fse->symtab()->new_proc(ftGenerated, src_c, name);
        ps_detect->set_fse(fse);

        // create a procedure symbol table for the procedure
        proc_symtab *pst_detect = new proc_symtab(name);
        fse->symtab()->add_child(pst_detect);
```

```
// create the arguments of the detect procedure, namely
// starting index, the step, and the upper bound

char input_name[100];

// add first argument
sprintf(input_name, "%s_input", tf->index()->name());
var_sym* input_var = pst_detect->new_var(type_signed, input_name);
input_var->set_param();
pst_detect->params()->append(input_var);

// add second argument
sprintf(input_name, "%s_step", tf->index()->name());
var_sym* step_var = pst_detect->new_var(type_signed, input_name);
step_var->set_param();
pst_detect->params()->append(step_var);

// add third argument
sprintf(input_name, "%s_stop", tf->index()->name());
var_sym* stop_var = pst_detect->new_var(type_signed, input_name);
stop_var->set_param();
pst_detect->params()->append(stop_var);

// make the body of the detect procedure
tree_node_list *detect_body = new tree_node_list;

// first insert a copy of the original loop itself
tree_for *tfc = tf->clone(pst_detect);
detect_body->push(tfc);

// create tree_proc for detect_procedure, and install
// the body into the tree_proc
tree_proc *tp_detect = new tree_proc(detect_body, pst_detect);
ps_detect->set_block(tp_detect);

// set the current procedure for the builder library
// using the following command
block::set_proc(tp_detect);

// set the lower bound, step, and upper bound of the
// for loop to argument values
(tfc->lb_op()).remove();
tfc->set_lb_op(operand(input_var));
(tfc->step_op()).remove();
tfc->set_step_op(operand(step_var));
(tfc->ub_op()).remove();
tfc->set_ub_op(operand(stop_var));

// the following call to get_args inserts
// calls to data structures within the runtime
// library to initialize local variables so they
// have the same values as in the main procedure

get_args(tfc, 1);


// this block of code genertes all the necessary
// additional code needed in the body of the loop
// to perform detection
{

       // set detect_flag = 1 so that the runtime system
       // begins detection (in parallel)
       block detect_flag_bl(detect_flag_vs);
       block set_detect_flag_bl(detect_flag_bl = block(1));
       tree_node *set_detect_flag_tn = set_detect_flag_bl.make_tree_node(tfc);
       tfc->parent()->insert_before(set_detect_flag_tn, tfc->list_e());
```

```
                // memory barrier to make sure that the runtime
                // system reads this value of detect_flag
                instruction *mb = new in_ldc(ptGenericFunc, operand(),
                                 immed(ps_memory_barrier));
                instruction *mb_in = new in_cal(type_v0, operand(),
                                 operand(mb));
                tree_instr *mb_ti = new tree_instr(mb_in);
                tfc->parent()->insert_before(mb_ti, tfc->list_e());

                // insert code into loop body to return if
                // detect_flag is equal to 2

                in_rrr *in_ret = new in_rrr(
                        io_ret, type_void, operand(), operand(tfc->index()));
                block index_bl(tfc->index());
                block     check_code_bl(block::IF(block(detect_flag_bl     ==     block(2)),
block(in_ret)));
                tree_node *check_code_tn = check_code_bl.make_tree_node(tfc->body());
                tfc->body()->insert_before(check_code_tn, tfc->body()->head());

                // insert code at end of loop to wait if detect_flag is not 2
                label_sym *wait_ls = pst_detect->new_label("wait");
                in_lab *wait_il = new in_lab(wait_ls);
                tree_instr *wait_ti =  new tree_instr(wait_il);

                in_bj *wait_branch = new in_bj(io_jmp, wait_ls);
                //tree_instr *wait_branch_ti = new tree_instr(wait_branch);
                block    if_detect_flag_bl(block::IF(block(detect_flag_bl    !=    block(2)),
block(wait_branch)));
                tree_node *if_detect_flag_tn = if_detect_flag_bl.make_tree_node(tfc);

                tfc->parent()->insert_after(if_detect_flag_tn, tfc->list_e());
                tfc->parent()->insert_after(wait_ti, tfc->list_e());
        }

        // insert return
        in_rrr *in_ret = new in_rrr(
                io_ret, type_void, operand(), operand(tfc->index()));
        tree_instr *ti_ret = new tree_instr(in_ret);
        detect_body->append(ti_ret);

        // increment count
        count++;

        // number the instructions in this procedure, since new
        // ones have been created
        tp_detect->number_instrs();

        return tp_detect;

}
```

generate_slave_code.cc

```
#include <duds.h>

/* This procedure inserts the address prediction/check for lod/str instructions
 * and (if necessary) the code needed for str instructions to save the
 * value to the write buffer.
 */
void instr_insert_address_prediction(
tree_for *tf,
instruction *in,
instruction_count *s,
proc_symtab *pst_slave,
var_sym *write_pointer_vs,
var_sym *ncancel_vs)
{

        // make sure there are no memcpy instructions
        // these instructions could be handled by breaking
        // them up into lod and str instructions, but
        // we haven't implemented that
  assert( in->opcode() != io_memcpy );

        // if the instruction is a lod or str,
        // generate the appropriate code
  if(in->opcode() == io_lod
        || in->opcode() == io_str) {

        in_rrr *inls = (in_rrr *) in;
        var_sym *actual_vs, *pvs;

        // obtain the address of the lod/str in a
        // variable  (actual_vs)
        if( inls->src1_op().is_instr() ) {

                instruction *tmp_ins = inls->src1_op().instr();
                type_node *tmp_typ = inls->src1_op().type();
                actual_vs = new var_sym(tmp_typ, "promote");
                pst_slave->add_sym(actual_vs);
                tmp_ins->remove();
                inls->src1_op().remove();
                tmp_ins->set_dst(actual_vs);

        } else if( inls->src1_op().is_symbol() ) {
                actual_vs = (var_sym *) inls->src1_op().symbol();
        } else {
                assert(FALSE);
        }



        instruction *init, *iter_num, *ref_num;

        // in case of lod
        if(in->opcode() == io_lod) {

                // create variables predict and delta needed to for
                // address calculations
                var_sym *predict_vs = new var_sym(type_unsigned_long, "predict");
                pst_slave->add_sym(predict_vs);
                block predict_bl(predict_vs);
                var_sym *delta_vs = new var_sym(type_signed_long, "delta");
                pst_slave->add_sym(delta_vs);

                // create builder structures for the following variables
                block read_count_bl(s->read_count);
                block delta_bl(delta_vs);
                block read_deltas_bl(read_deltas_vs);
                block ncancel_bl(ncancel_vs);
                block actual_bl(actual_vs);

                // initialize read delta before for loop
                block init_read_delta_bl(delta_bl = read_deltas_bl[read_count_bl]);
                tree_node *init_read_delta_tn = init_read_delta_bl.make_tree_node(tf);
                tf->parent()->insert_before(init_read_delta_tn, tf->list_e());

                // initialize predicted address before for loop
                // by accessing the profiled address data structures
                // in the runtime system.
                block read_addr_prof_bl(read_addr_prof_vs);
                block read_array_access_bl(read_addr_prof_bl[block(0)][read_count_bl]);
                instruction                       *read_array_access                         =
read_array_access_bl.make_instruction(tf);
                block      factor_bl(block(delta_vs)      *      (block(tf->lb_op())      -
block(inner_start_vs))/block(tf->step_op()));
                instruction *factor = factor_bl.make_instruction(tf);
                instruction  *init_read_address  =  new  in_rrr(io_add,  type_unsigned_long,
operand(predict_vs), operand(read_array_access), operand(factor));
                tree_instr *init_read_address_ti = new tree_instr(init_read_address);
                tf->parent()->insert_before(init_read_address_ti, tf->list_e());

                // cast the predicted address from an unsigned long to
                // the type of the lod, can't use builder for this
                // since it inserts its own casts.
                instruction *predict_vs_cast = new in_rrr(io_cvt,
                        actual_vs->type(),
                        operand(), operand(predict_vs));
                inls->set_src1(operand(predict_vs_cast));

                // insert code to check predicted address
                block   check_predict_bl(ncancel_bl   =   ncancel_bl   &   (predict_bl   ==
actual_vs));
                tree_node *check_predict_tn = check_predict_bl.make_tree_node(tf);

                // increment the predicted address, don't use builder for this
                instruction *incr_predict = new in_rrr(io_add,
                        type_unsigned_long,
                        operand(predict_vs),
                        operand(predict_vs),
                        operand(delta_vs));
                tree_instr *incr_predict_tn = new tree_instr(incr_predict);

                // insert the generated instructions
                inls->parent()->parent()->insert_before(check_predict_tn,   inls->parent()-
>list_e());
                s->local_count++;
                inls->parent()->parent()->insert_after(incr_predict_tn,     inls->parent()-
>parent()->tail());

                // increment counts
                s->local_count++;
                s->read_count += 1;

        } else {

                // in case of a str instruction ..

                // create variables predict, delta, value, needed for
                // address calculations and value preservation
                var_sym *predict_vs = new var_sym(type_unsigned_long, "predict");
                pst_slave->add_sym(predict_vs);
                block predict_bl(predict_vs);
                var_sym *delta_vs = new var_sym(type_signed_long, "delta");
                pst_slave->add_sym(delta_vs);
                var_sym *value_vs = new var_sym(type_double, "value");
                pst_slave->add_sym(value_vs);

                // creat block structures for the following variables
                block write_double_count_bl(s->write_double_count);
                block delta_bl(delta_vs);
                block write_double_deltas_bl(write_double_deltas_vs);
                block actual_bl(actual_vs);
                block ncancel_bl(ncancel_vs);

                // initialize write double delta before for loop
                // using data structures from the runtime library
                block                init_write_double_delta_bl(delta_bl                 =
write_double_deltas_bl[write_double_count_bl]);
```

```
               tree_node                  *init_write_double_delta_tn                  =
init_write_double_delta_bl.make_tree_node(tf);
               tf->parent()->insert_before(init_write_double_delta_tn, tf->list_e());

               // initialize predicted address before for loop
               // using data structures from the runtime library
               block write_double_addr_prof_bl(write_double_addr_prof_vs);
               block
write_double_array_access_bl(write_double_addr_prof_bl[block(0)][write_double_count_bl]);
               instruction                 *write_double_array_access                  =
write_double_array_access_bl.make_instruction(tf);
               block      factor_bl(block(delta_vs)      *      (block(tf->lb_op())      -
block(inner_start_vs))/block(tf->step_op()));
               instruction *factor = factor_bl.make_instruction(tf);
               instruction     *init_write_double_address     =     new     in_rrr(io_add,
type_unsigned_long,        operand(predict_vs),        operand(write_double_array_access),
operand(factor));
               tree_instr          *init_write_double_address_ti          =          new
tree_instr(init_write_double_address);
               tf->parent()->insert_before(init_write_double_address_ti, tf->list_e());

               // cast the predicted address from an unsigned long to
               // the type of the lod, can't use builder for this
               // since it inserts its own casts.
               instruction *predict_vs_cast = new in_rrr(io_cvt,
                       actual_vs->type(),
                       operand(), operand(predict_vs));
               inls->set_src1(operand(predict_vs_cast));

               // save the current value of the memory at predicted address
               instruction *predict_cast = new in_rrr(io_cvt,
                               write_pointer_vs->type(), operand(),
                               operand(predict_vs));
               block save_value_bl;
               save_value_bl.set(block(write_pointer_vs).dref()                            =
block(predict_cast).dref());
               tree_node *save_value_tn = save_value_bl.make_tree_node(tf);

               // increment the write_pointer
               instruction     *one     =     new     in_ldc(type_signed,     operand(),
immed(DOUBLE_TO_CHAR));
               instruction  *incr_write_pointer  =  new  in_rrr(io_add,  write_pointer_vs-
>type(), operand(write_pointer_vs), operand(one), operand(write_pointer_vs));
               tree_instr *incr_write_pointer_tn = new tree_instr(incr_write_pointer);

               // insert code to check predicted address
               block   check_predict_bl(ncancel_bl   =   ncancel_bl   &   (predict_bl   ==
actual_vs));
               tree_node *check_predict_tn = check_predict_bl.make_tree_node(tf);

               // increment the predicted address, don't use builder for
               // this
               instruction *incr_predict = new in_rrr(io_add, type_unsigned_long,
                       operand(predict_vs), operand(predict_vs), operand(delta_vs));
               tree_instr *incr_predict_tn = new tree_instr(incr_predict);

               // insert instructions into for loop body
               inls->parent()->parent()->insert_before(save_value_tn,      inls->parent()-
>list_e());
               s->local_count++;
               inls->parent()->parent()->insert_after(incr_write_pointer_tn,         inls-
>parent()->list_e());
               s->local_count++;
               inls->parent()->parent()->insert_after(incr_predict_tn,     inls->parent()-
>parent()->tail());
               s->local_count++;
               inls->parent()->parent()->insert_after(check_predict_tn,    inls->parent()-
>list_e());
```

```
                // increment counters
                s->local_count++;
                s->write_double_count += 1;
        }

  }

  return;

}

/* This procedure inserts the address prediction and
 * checking mechanisms for each load and store in the
 * for loop, by applying instr_insert_addr_prediction
 * to each instruction.
 */
instruction_count *insert_address_prediction(
tree_for *tf,
instruction_count *s,
proc_symtab *pst_slave,
var_sym *write_buffer_vs)
{

        // first create and initialize ncancel to 1 at the
        // beginning of the loop.
        // ncancel is used to detect any address mispredictions
        var_sym *ncancel_vs = new var_sym(type_unsigned, "ncancel");
        pst_slave->add_sym(ncancel_vs);
        block set_ncancel_bl;
        set_ncancel_bl.set(block(ncancel_vs) = block(1));
        tree_node *set_ncancel_tn = set_ncancel_bl.make_tree_node(tf);
        tf->parent()->insert_before(set_ncancel_tn, tf->list_e());

        // create and set write_pointer, which points to the
        // next free address in the write buffer
        var_sym *write_pointer_vs = new var_sym(type_double->ptr_to(), "write_pointer");
        pst_slave->add_sym(write_pointer_vs);
        block set_write_pointer_bl;
        set_write_pointer_bl.set(block(write_pointer_vs) = block(write_buffer_vs));
        tree_node *set_write_pointer_tn = set_write_pointer_bl.make_tree_node(tf);

        tf->parent()->insert_before(set_write_pointer_tn, tf->list_e());


        // initialize the counts
        s->read_count = 0;
        s->write_double_count = 0;

        // now traverse the body of the for loop
        tree_node_list *tnl = tf->body();
        int length = tnl->count();

        for(int i=0; i<tnl->count(); i++) {

                tree_node *tn = (*tnl)[i];
                if( tn->is_instr() ) {
                        tn->print();
                        s->local_count = 0;
                        tree_instr *ti = (tree_instr *) tn;
                        instr_insert_address_prediction(
                                        tf,
                                        ti->instr(), s,
                                        pst_slave,
                                        write_pointer_vs,
                                        ncancel_vs);
                        i+= s->local_count;
                }
        }

        // generate code for conditional return if there was a
        // misprediction
        instruction *whatever = new in_rrr(io_sub, type_signed,
               operand(), operand(tf->lb_op()), operand(tf->step_op()));
        in_rrr *in_ret = new in_rrr(
               io_ret, type_void, operand(), operand(whatever));
        // tree_instr *ti_ret = new tree_instr(in_ret);
        block cond_ret_bl(block::IF(block(block(ncancel_vs) == 0), block(in_ret)));
        tree_node *cond_ret_tn = cond_ret_bl.make_tree_node(tf);
        tf->parent()->insert_after(cond_ret_tn, tf->list_e());

        return s;

}


/* as the name suggests ..
 */
tree_proc *create_slave_loop(
file_set_entry *fse,
tree_for *tf,
instruction_count *s)
{

        // a variable that keeps track of the number of
        // slave procedures successfully generated, so
        // it can give them different names.
        static int count=0;
        char name[100];
        sprintf(name, "slave%d", count);

        // create the procedure and install it into the
        // file symbol table
        proc_sym *ps_slave
               = fse->symtab()->new_proc(ftSlave, src_c,
                      name);
        ps_slave->set_fse(fse);

        // create a procedure symbol table for the procedure
        proc_symtab *pst_slave = new proc_symtab(name);
        fse->symtab()->add_child(pst_slave);

        // create the arguments of the procedure, namely
        // starting index, the step, and the upper bound

        char input_name[100];

        // add first argument
        sprintf(input_name, "%s_input", tf->index()->name());
        var_sym* input_var = pst_slave->new_var(type_signed, input_name);
        input_var->set_param();
        pst_slave->params()->append(input_var);

        // add second argument
        sprintf(input_name, "%s_step", tf->index()->name());
        var_sym* step_var = pst_slave->new_var(type_signed, input_name);
        step_var->set_param();
        pst_slave->params()->append(step_var);

        // add third argument
        sprintf(input_name, "%s_stop", tf->index()->name());
        var_sym* stop_var = pst_slave->new_var(type_signed, input_name);
        stop_var->set_param();
        pst_slave->params()->append(stop_var);
        // add fourth arguemnt
        sprintf(input_name, "write_buffer");
        var_sym *write_buffer_vs = pst_slave->new_var(type_double->ptr_to(), input_name);
        write_buffer_vs->set_param();
         pst_slave->params()->append(write_buffer_vs);

         // make the body of the slave procedure
        tree_node_list *slave_body = new tree_node_list;

         // first insert a copy of the original loop itself
        tree_for *tfc = tf->clone(pst_slave);
        slave_body->push(tfc);

         // create tree_proc for  profile procedure, and install
         // the body into the tree_proc
        tree_proc *tp_slave = new tree_proc(slave_body, pst_slave);
        ps_slave->set_block(tp_slave);

         // set the current procedure for the builder library
         // using the following command
        block::set_proc(tp_slave);

         // set the lower bound, step, and upper bound of the
         // for loop to argument values
        (tfc->lb_op()).remove();
        tfc->set_lb_op(operand(input_var));
        (tfc->step_op()).remove();
        tfc->set_step_op(operand(step_var));
        (tfc->ub_op()).remove();
        tfc->set_ub_op(operand(stop_var));

         // the following call to get_args inserts
         // calls to data structures within the runtime
         // library to initialize local variables so they
         // have the same values as in the main procedure
        get_args(tfc, 0);

        // insert address prediction code
        insert_address_prediction(tfc,s,pst_slave,write_buffer_vs);

        // insert the return statement
        in_rrr *in_ret = new in_rrr(
               io_ret, type_void, operand(), operand(tfc->index()));
        tree_instr *ti_ret = new tree_instr(in_ret);
        slave_body->append(ti_ret);

        // increment the counter
        count++;

        // number the newly created instructions in the
        // procedure
        tp_slave->number_instrs();

        return tp_slave;

}
```

Appendix C

This section contains source code in C for the runtime system compiler. The header files are lib.h, lib.arithmetic.h, lib.detecct.h, lib.master.h, lib.slave.h and the source code files are, lib.arithmetic.c, lib.detect.c, lib.master.c, lib.slave.c.

lib.h:

```
#define MEMORY_BARRIER_REQUIRED

#define CACHE_LINE_FACTOR 3

#define MAX_NUM_MEM_REFS 50
#define MAX_PAR_ITERS 1000
#define MAX_ARGS 50
#define NUM_PROF_ITERS 3
#define NUM_THREADS 3

typedef int (*spec_function)(int, int, int, double *);


/* lib.detect variables */
extern int detect_flag;
extern unsigned long read_addr_prof[NUM_PROF_ITERS][MAX_NUM_MEM_REFS];
extern unsigned long write_double_addr_prof[NUM_PROF_ITERS][MAX_NUM_MEM_REFS];
extern long read_deltas[MAX_NUM_MEM_REFS];
extern long write_double_deltas[MAX_NUM_MEM_REFS];
extern int num_par_iters;

/* lib.master variables */

extern int *flags;
extern int *finished;
extern spec_function spec_loop;
extern int num_read_refs;
extern int num_write_double_refs;
extern int inner;
extern int inner_start;
extern int inner_step;
extern int inner_stop;
extern int int_args[MAX_ARGS];
extern double double_args[MAX_ARGS];
extern void *pointer_args[MAX_ARGS];

/* macros */

#define FLAGS(n) flags[(n) << CACHE_LINE_FACTOR]
#define FINISHED(n) finished[(n) << CACHE_LINE_FACTOR]
#define THREADS(x) threads[((x)%NUM_THREADS) << CACHE_LINE_FACTOR]
```

lib.arithmetic.h:

```
#define min(x,y)  ( (x) > (y) ? (y) : (x) )
#define max(x,y)  ( (x) < (y) ? (y) : (x) )

struct gcd_struct {
        long a;
        long b;
        long gcd;
        long x;
        long y;
};
```

```
typedef struct gcd_struct  gcd_struct;

void gcd(gcd_struct *g);

int intersect(
unsigned long p,
long s,
unsigned long q,
long t);
```

lib.detect.h

```
void detect(void *x);
```

lib.master.h

```
void pass_loop_specifics(

spec_function spec_loop_function,
int num_read_refs_input,
int num_write_double_refs_input,
int inner_start_input);

void spec_init();

int spec_try(
int inner_start_input,
int inner_step_input,
int inner_stop_input);

void spec_quit();
```

lib.slave.h:

```
int stop(int tid);
void spec_do(void *x);
```

lib.arithmetic.c:

```
#include <lib.arithmetic.h>

/* #define PRINT_INTERSECT */

/* This procedure calculates the gcd (a,b) of two numbers a and b.
 * It also calculates x and y such that x*a+y*b=(a,b).
 * inputs are passed through the structure g, and results are
 * returned through the same structure.
 */
void gcd(gcd_struct *g) /* based on Aho, Hopcroft, Ullman, p. 301 */
{

        long x0=1, y0=0, x1=0, y1=1, i=1;
        long a0=g->a, a1=g->b;

        /* special case a1=b=0 */
        if (a1==0) {
          g->gcd=a0;
          g->x=x0;
          g->y=y0;
          return;
        }

        while(a0%a1 != 0) {

                long q, temp;

                q = a0/a1;

                temp = a1;
                a1 = a0 - q*(a1);
                a0 = temp;

                temp = x1;
                x1 = x0 - q*x1;
                x0 = temp;

                temp = y1;
                y1 = y0 - q*y1;
                y0 = temp;

        }

        g->gcd = a1;
        g->x = x1;
        g->y = y1;

}

long mod(       /* find minimum non-negative value of a + b * k */
long a,                 /* if non exists, then return a */
long b)
{

        if( b > 0 )
                return (a >= 0) ? a%b : (a%b) + b;
        else if( b < 0 )
                return (a >= 0) ? a%b : (a%b) - b;

        /* means b == 0 */

        return a;


}

int intersect( /* returns -1 if there is no intersection for positive displacements */
unsigned long p,        /* returns the positive displacement if there is intersection */
long s,         /* do not use this for s==0 and t==0 at the same time */
unsigned long q,
long t)

{

        gcd_struct g;
        long diff;
        long d, m, n, l;
        long x, y, x1, y1, x2, y2;
        long valid1, valid2, same, zero;


        /*
        if(p==q && s==t) {
                return -1;
        } else if(p==q) {

        }
        */
        if (q > p) {
                unsigned long utemp;
                long itemp;

                utemp = p;
                p = q;
                q = utemp;

                itemp = s;
                s = t;
                t = itemp;
        }

        diff = ((long) (p-q));

        g.a = -s; g.b = t;

#ifdef PRINT_INTERSECT
        printf("before gcd\n");
#endif
        gcd(&g);
        d = g.gcd;
#ifdef PRINT_INTERSECT
        printf("gcd passed\n");
#endif

        if( diff % d  != 0)
                return -1;

        l = diff/d;
        x = l * g.x;
        y = l * g.y;

        /*      by now
         *      p + s * x = q + t * y  (non-zero solution)
         */

        if(s == 0)
                return y>0 ? y : -1;

        if(t == 0)
                return x>0 ? x : -1;

        m = t/d;
        n = s/d;

        x1 = mod(x,m);
        y1 = (diff + s * x1)/t;
        y2 = mod(y,n);
        x2 = (t * y2 - diff)/s;

#ifdef PRINT_INTERSECT
                printf("** intersect: p=%u, s=%d, q=%u, t=%d\n** intersect: x=%d, m=%d,
y=%d, n=%d, \n** intersect: (%d,%d) (%d,%d)\n",p,s,q,t,x,m,y,n,x1,y1,x2,y2);
#endif

        /* special case when the solutions at multiples of (m,n) */
        zero = (x1 == 0 && y1 == 0) || (x2 == 0 && y2 == 0);
        if(zero) {
                if(m*n < 0 || m==n)
                        return -1;

                return (m > 0 && n > 0) ? max(m,n) : -min(m,n);
        }

        valid1 = ( x1 >= 0 && y1 >= 0);
        valid2 = ( x2 >= 0 && y2 >= 0);
        same = ( x1 == x2 && y1 == y2 );

        /* look for another special case */
        if(valid1 && valid2 && same)
                        return max(x1,y1);

        if(valid1) {
                if(valid2) {
                        /* implies not same */
                        return (x1*y2-x2*y1)/((y2-y1)-(x2-x1));
                } else {
                        return max(x1,y1);
                }
        } else {
                if(valid2)
                        return max(x2,y2);
                else
                        return -1;
        }

        return 0;
```

lib.detect.c

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <lib.h>
#include <lib.arithmetic.h>
#include <lib.detect.h>

/* #define PRINT_OK */

int detect_flag=0;
int detect_flag;
unsigned long read_addr_prof[NUM_PROF_ITERS][MAX_NUM_MEM_REFS];
unsigned long write_double_addr_prof[NUM_PROF_ITERS][MAX_NUM_MEM_REFS];
long read_deltas[MAX_NUM_MEM_REFS];
long write_double_deltas[MAX_NUM_MEM_REFS];
int num_par_iters;

/* This procedure is called by the thread running detect().
 * It performs detection on the values loaded in the data structures
 * above
 */

int perform_detect(void)
{

        /* detect pattern
         * returns the number of parallelizable iterations
         */

        int i;
        int local_num_par_iters=0;
        int local_num_read_refs = num_read_refs;
        int local_num_write_double_refs = num_write_double_refs;
        int constant_non_zero_deltas=1;
        unsigned long local_write_double_first_iteration[MAX_NUM_MEM_REFS];
        unsigned long local_read_first_iteration[MAX_NUM_MEM_REFS];

#ifdef PRINT_OK
        printf("entering perform_detect\n");
#endif
        for(i=0; i<local_num_write_double_refs; i++) {
```

```
        int temp1, temp2;

        local_write_double_first_iteration[i] =
                (write_double_addr_prof[0])[i];

        temp1 = ((int) (write_double_addr_prof[1])[i])
                - ((int) (write_double_addr_prof[0])[i]);
        temp2 = ((int) (write_double_addr_prof[2])[i])
                - ((int) (write_double_addr_prof[1])[i]);

        if( temp1 != temp2 || temp1 == 0 ) {
                constant_non_zero_deltas = 0;
        } else {
                write_double_deltas[i] =  temp1;
        }
}

for(i=0; i<local_num_read_refs; i++) {

        int temp1, temp2;


        local_read_first_iteration[i] =
                (read_addr_prof[0])[i];

        temp1 = ((int) (read_addr_prof[1])[i])
                - ((int) ((read_addr_prof[0])[i]));
        temp2 = ((int) (read_addr_prof[2])[i])
                - ((int) ((read_addr_prof[1]))[i]);

        if( temp1 != temp2 ) {
                constant_non_zero_deltas = 0;
        } else {
                read_deltas[i] =  temp1;
        }
}

/* determine whether prediction is possible */

if(constant_non_zero_deltas) {

          /* insert WAW and WAR dependence check
           * and replace code below
           * local_num_par_iters is the number of
           * parallelizable iterations
           */

          local_num_par_iters = -1;

          for(i=0; i<local_num_write_double_refs; i++) {

                int j;
                unsigned long p=local_write_double_first_iteration[i];
                long s=write_double_deltas[i];
                /* Inter-iteration Output dependency checks
                 * i.e. WAW
                 */

                 for(j=0; j<i; j++) {
                        unsigned long q=local_write_double_first_iteration[j];

                        long t=write_double_deltas[j];
                        int intersect_result;
                        intersect_result = intersect(p,s,q,t);

                        if( intersect_result > -1 ) {
                                   local_num_par_iters
```

```
                                        = (local_num_par_iters == -1) ?
                                          intersect_result :
                                          min(local_num_par_iters, intersect_result);
                                  }

                        }

                        /* Inter-iteration True and Anti dependency
                         * checks, i.e. RAW and WAR
                         */

                        for(j=0; j<num_read_refs; j++) {
                                unsigned long q=local_read_first_iteration[j];
                                long t=read_deltas[j];
                                int intersect_result;
#ifdef PRINT_OK
                             printf("comparing  W%d  and  R%d  p  s  q  t:  %lu  %ld  %lu
%ld\n",i,j,p,s,q,t);
#endif
                                intersect_result = intersect(p,s,q,t);
#ifdef PRINT_OK
                          printf("finished intersect\n");
#endif

                                if( intersect_result > -1 ) {
                                        local_num_par_iters
                                        = (local_num_par_iters == -1) ?
                                              intersect_result   :   min(local_num_par_iters,
intersect_result);
                                }
                        }
                }

        } else { /* no prediction possible */

                local_num_par_iters=0;
        }

#ifdef PRINT_OK
        printf("exiting perform_detect\n");
#endif

        return local_num_par_iters;

}

void detect(void *x)
{
        /* detect_flag is initailly set to zero by proc 0
         * when proc 0 requests detection, then it changes detect_flag
         * to one. Next it changes back to 2 to indicate that the
         * prediction calculation has been completed
         */

        int i, j, constant_non_zero_deltas, hazard, local_num_par_iters;
        unsigned local_read_first_iteration[MAX_NUM_MEM_REFS];
        unsigned local_write_double_first_iteration[MAX_NUM_MEM_REFS];

        while(1) {

                int flag = detect_flag;
asm("mb");


                if( flag < 0 ) {

                        /* means that the program is over */
                        break;
                } else if( flag == 1 ) {

                        num_par_iters = perform_detect();
                        detect_flag = 2;
asm("mb");


                } else if(flag == 0 || flag == 2) {

                        /* ok */

                }

        }

        return;
}
```

lib.master.c

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
#include <lib.h>
#include <lib.master.h>
#include <lib.detect.h>
#include <lib.slave.h>
#include <lib.arithmetic.h>

/* #define PRINT_OK */

int num_read_refs;
int num_write_double_refs;

/* signalling flags */
int *flags, *finished;

/* argument passing protocol */
spec_function spec_loop;

int inner;
int inner_start;
int inner_step;
int inner_stop;

int int_args[MAX_ARGS];
double double_args[MAX_ARGS];
void *pointer_args[MAX_ARGS];

/* threads */
pthread_t threads[NUM_THREADS << CACHE_LINE_FACTOR];
int thread_args[NUM_THREADS << CACHE_LINE_FACTOR];

/* macros */
#define THREADS(x) threads[((x)%NUM_THREADS) << CACHE_LINE_FACTOR]
#define THREAD_ARGS(x) thread_args[((x)%NUM_THREADS) << CACHE_LINE_FACTOR]

void pass_loop_specifics(
spec_function spec_loop_input,
int num_read_refs_input,
int num_write_double_refs_input,
int inner_start_input)
{

        assert(num_read_refs <= MAX_NUM_MEM_REFS);
        assert(num_write_double_refs <= MAX_NUM_MEM_REFS);

        spec_loop = spec_loop_input;
        num_read_refs = num_read_refs_input;
        num_write_double_refs = num_write_double_refs_input;
        inner_start = inner_start_input;

        return;

}

void master_memory_init(void)
{

        int i;

        /* create signalling flags */

        flags = (int *) malloc(sizeof(int) * (NUM_THREADS << CACHE_LINE_FACTOR));
        finished = (int *) malloc(sizeof(int) * (NUM_THREADS << CACHE_LINE_FACTOR));

        /* initialize signalling flags */
        for(i=0; i<NUM_THREADS; i++) {
                FLAGS(i) = 0;
                FINISHED(i) = 0;
        }

        /*
        int_args = (int *) malloc(sizeof(int) * MAX_ARGS);
        double_args = (double *) malloc(sizeof(double) * MAX_ARGS);
        pointer_args = (void **) malloc(sizeof(void *) * MAX_ARGS);
        read_deltas = (long *) malloc(sizeof(long) * MAX_NUM_MEM_REFS);
        write_double_deltas = (long *) malloc(sizeof(long) * MAX_NUM_MEM_REFS);
        read_addr_prof   =   (unsigned   long   **)   malloc(sizeof(unsigned   long   *)   *
NUM_PROF_ITERS);
        write_double_addr_prof  =  (unsigned  long  **)  malloc(sizeof(unsigned  long  **)  *
NUM_PROF_ITERS);
        for(i=0; i<NUM_PROF_ITERS; i++) {
                read_addr_prof[i] = (unsigned long *)
                        malloc(sizeof(unsigned long) * MAX_NUM_MEM_REFS);
                write_double_addr_prof[i] = (unsigned long *)
                        malloc(sizeof(unsigned long) * MAX_NUM_MEM_REFS);
        }
        */

asm("mb");

        return;

}

void master_threads_init(void)
{

        int i;

        /* detect thread */
        THREAD_ARGS(NUM_THREADS-1) = NUM_THREADS-1;
        pthread_create(&(THREADS(NUM_THREADS-1)), (void *) NULL, (void *) detect, (void *)
&(THREAD_ARGS(NUM_THREADS-1)));


        /* slave threads */
        for(i=1; i<NUM_THREADS-1; i++) {
                THREAD_ARGS(i) = i;
                pthread_create(&(THREADS(i)),  (void  *)  NULL,  (void  *)  spec_do,  (void *)
&(THREAD_ARGS(i)));
```

```
        }

        return;

}

/* loop independent */
void spec_init(void)
{

        int i;

        master_memory_init();

        master_threads_init();

        return;

}

/* check whether we should parallelize the rest of the loop */
int should_parallelize (
int num_par_iters,
int inner,
int inner_step,
int inner_stop) {

  int max_iters;
  int rest_iters;
  int bound;

  rest_iters = (inner_stop-inner)/inner_step;

  max_iters = (num_par_iters == -1) ?
    rest_iters :
    min(rest_iters,
        num_par_iters-(inner-inner_start)/inner_step);

  bound = max(NUM_THREADS-2,rest_iters);

  return max_iters>=bound;
}

/* loop independent */
int spec_try(
int inner_input,
int inner_step_input,
int inner_stop_input)
{

        int cancel;
        int local_inner;
        int local_num_par_iters = num_par_iters;

#ifdef PRINT_OK
        printf("spec_try: num_par_iters = %d\n", local_num_par_iters);
#endif

        /* if detection is not possible, then return */
        if(local_num_par_iters == 0) {
                return inner_input;
        }

        inner = inner_input;
        inner_step = inner_step_input;
        inner_stop = inner_stop_input;

        if (!should_parallelize(local_num_par_iters,
                              inner,inner_step,inner_stop))
          return inner;

        cancel=0;

        /* begin speculative execution
         * pass arguments
         */

        /* start speculative execution */
        FLAGS(0) += 1;
        FINISHED(0) += 1;

asm("mb");


        /* wait until all other threads have started
         * this condition is needed to ensure that
         * the finished flags reflect the state of the
         * current loop and not the previous loop
         */

#ifdef PRINT_OK
        printf("spec_try: waiting for speculative threads to start\n");
#endif
        while( FLAGS(NUM_THREADS-2) < FLAGS(0) ) {
asm("mb");
        }
#ifdef PRINT_OK
        printf("spec_try: done waiting for speculative threads to start\n");
#endif


        /* wait until all other threads have
         * finished and then calculate final i
         */

        while(1) {

                int s, t;
                s = FINISHED(NUM_THREADS-2);
                t = FINISHED(0);

asm("mb");

                if(s >= t) {
                        local_inner = inner_stop;
                        break;
                } else if (s < 0) {
                        int k;
                        for(k=1; k<=NUM_THREADS-2; k++) {
                                if(FINISHED(k) < 0) {
asm("mb");

                                        k++;
                                        break;
                                }
                        }
                        local_inner = stop(k-1);
                        break;
                }
         }

        return local_inner;
        printf("spec_try: all threads completed\n");
}

void spec_quit() /* loop independent */
```

```
{

        int j;

        FLAGS(0) = -1;
        detect_flag = -1;

asm("mb");


        for(j=1; j<NUM_THREADS; j++) {
                pthread_join(THREADS(j), NULL);
        }

         return;
}
```

lib.slave.c:

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <lib.h>
#include <lib.slave.h>

int start(int tid) {

        int local_inner_step, local_par_iters, local_inner_start;
        local_inner_step = inner_step;
        local_par_iters = (inner_stop-inner)/(local_inner_step * (NUM_THREADS-2));
        local_inner_start = inner + (tid-1) * local_par_iters * local_inner_step;

#ifdef PRINT_OK
        printf("start(%d), local_par_iters = %d\n", tid, local_par_iters);
#endif
        return local_inner_start;
}

int stop(int tid) {

        int local_inner_step, local_par_iters, local_inner_stop;

        local_inner_step = inner_step;
        local_par_iters = (inner_stop-inner)/(local_inner_step * (NUM_THREADS-2));
        local_inner_stop
                = (tid == NUM_THREADS-2) ?
                inner_stop : inner+ tid * local_par_iters * local_inner_step;

#ifdef PRINT_OK
        printf("stop(%d), local_par_iters = %d\n", tid, local_par_iters);
#endif
        return local_inner_stop;

}

void spec_double_undo(int tid, int iter, double *write_double_rv)
{

        /* CHECK TO VERFY THAT THE INDICES ARE INITIALIZED
         * AND INCREMENTED PROPERLY
         */

        int i;
        int local_inner_start = inner_start;
        int local_inner_step = inner_step;
        int local_inner;
        long local_write_double_deltas[MAX_NUM_MEM_REFS];
        double *address[MAX_NUM_MEM_REFS];
        double *buffer_pointer;

        /* load addresses/deltas into local buffers */
        for(i=0; i<num_write_double_refs; i++) {
                local_write_double_deltas[i]
                  = write_double_deltas[i];
                address[i] = (double *)
                        ( write_double_addr_prof[0][i]
                        + iter * local_write_double_deltas[i] );
        }

        local_inner = stop(tid);
        buffer_pointer = write_double_rv + iter * num_write_double_refs;

        for(i=iter-1; i>=0; i--) {
                int k;
                local_inner -= local_inner_step;
                for(k=num_write_double_refs-1; k>=0; k--) {
                        address[k]   =   (double   *)   (((unsigned   long)   address[k])-
local_write_double_deltas[k]);
                        buffer_pointer--;
                        *(address[k]) = *buffer_pointer;
                }
        }

        return;
}

void  spec_loop_wrap_up(int  *my_finished,  int  *prev_finished,  int  cancel,  int  tid,  int
iter, double *write_double_rv)
{

        int s, t;

          /* wait until previous processors finish */
        while(1) {
                s = (*prev_finished);
                t = (*my_finished);
asm("mb");

                if( s >= t || s < 0)
                        break;
        }

        if( s < 0 || cancel ) {
                (*my_finished) = -1;
asm("mb");

                spec_double_undo(tid, iter, write_double_rv);
        } else {
                /* then signal next processor
                 * to complete */
                (*my_finished) += 1;
asm("mb");


        }

        return;

}



void spec_loop_start(int tid, double *write_double_rv)
```

```
{

         int k, iter=0;
        int cancel=0;
         int local_inner_start;
        int local_inner_step;
         int local_inner_stop;
        int local_inner;
        spec_function spec_loop_local;

        /* flags */
        int *my_flag = &(FLAGS(tid));
        int *prev_finished = &(FINISHED(tid-1));
        int *my_finished = &(FINISHED(tid));

        /* signal next processor to begin execution */
        (*my_finished) = (*my_flag);
         (*my_flag) += 1;
asm("mb");


        /* download parameters locally */
        local_inner_start = start(tid);
        local_inner_step=inner_step;
        local_inner_stop = stop(tid);
        local_inner = local_inner_start;
        spec_loop_local = spec_loop;

#ifdef PRINT_OK
        printf("%d: starting speculative execution (%d, %d)\n", tid, local_inner_start,
local_inner_stop);
#endif

        local_inner = (*spec_loop)(local_inner_start, local_inner_step, local_inner_stop,
write_double_rv);

        iter += (local_inner-local_inner_start)/local_inner_step;

        /* check to see if there are any mispredicted addresses */
        if(local_inner == local_inner_start-local_inner_step) {
                cancel=1;
        }
#ifdef PRINT_OK
        printf("%d: finished speculative execution, cancel = %d\n", tid, cancel);
#endif
        spec_loop_wrap_up(my_finished, prev_finished, cancel, tid, iter, write_double_rv);

        return;
}

/* loop independent */
void spec_do(void *x)
{

        int k;

        int tid = *((int *) x);
        int *my_flag = &(FLAGS(tid));
        int *prev_flag = &(FLAGS(tid-1));
        double *write_double_rv = (double *)
                malloc(sizeof(double) * MAX_PAR_ITERS * MAX_NUM_MEM_REFS);

         while(1) {

                int counter = (*my_flag);
                 int t = (*prev_flag);
asm("mb");
```

```
                /* wait until previous processors completed */
                if( t < 0 ) {
                        /* means time to quit, signal next processor */
                        (*my_flag) = -1;
asm("mb");

                        break;
                } else if( t > counter ) {
                       spec_loop_start(tid, write_double_rv);
               }
        }

        return;
}
```