Compiler Techniques for Concurrent Multithreading with Hardware Speculation Support *

Zhiyuan Li, Jenn-Yuan Tsai†, Xin Wang, Pen-Chung Yew, and Bess Zheng

Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

†Department of Computer Science
University of Illinois
Urbana, IL 61801

Abstract. Recently proposed concurrent multithreading architectures employ sophisticated hardware to support speculation on control and data dependences as well as run-time data dependence check, which enables parallelization of program regions such as while-loops which previously were ignored. The new architectures demand compilers to put more emphasis on the formation and selection of parallel threads. Compilers also play an important role in reducing the cost of run-time data dependence check. This paper discusses these new issues.

1 Introduction

Increasing density of VISI microprocessors not only continues to shorten the circuit latency but also provides more transistors on a single chip. Each new generation of microprocessors introduces more sophisticated mechanisms to support instruction-level parallelism. Future microprocessors will soon be able to issue and execute more than a dozen instructions per machine cycle. How to extract sufficient independent instructions per machine cycle from an ordinary program has become an increasingly difficult challenge. The currently predominant superscalar architectures adopt a single thread of control flow. Parallelism can be extracted only from within a relatively narrow window of consecutive instructions. Although independent instructions can be statically reordered and packed into a window, such code motion is constrained by programs' control structures. Growing evidences suggest that future processors need to take advantage of multiple threads of execution in order to find sufficient parallel operations [11]. Allowing multiple threads of execution is similar to, but not exactly like, placing multiprocessors on a single chip or on a multichip-module (MCM) [10]. This approach avoids some difficulties in traditional multiprocessor, such as high overhead in scheduling and synchronization.

Much of the responsibility of forming and scheduling parallel threads to execute on multiprocessors traditionally rests on compilers. There are several aspects to this responsibility:

* This work was supported in part by NSF CAREER Award CCR-9502541, NSF Grant MIP 9496320, a gift from Intel Corporation, and by the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D 346. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U. S. Army Intelligence Center and Fort Huachuca, or the U.S. Government.

- identifying independent operations;
- selecting and scheduling threads whose parallel execution could result in the shortest possible execution time; and
- inserting synchronization instructions to observe the dependences, if any, among parallel threads.

For traditional multiprocessors, control dependences greatly limit the compiler's ability to create parallel threads. For example, a compiler normally does not know how many iterations of a while-loop will be executed at run time and hence it cannot safely generate multiple threads to execute different loop iterations simultaneously. Recently, several hardware mechanisms have been proposed to allow speculative execution of multiple threads [3, 11, 2, 12]. A thread, whose execution may depend on run-time conditions, is allowed to execute before those conditions are resolved. Once those conditions are resolved, a correctly speculated thread can then write its results to the memory. On the other hand, an incorrectly speculated thread is squashed. Such *concurrent multithreading* architectures may also provide hardware for run-time data dependence check [11, 12]. These new hardware features create new parallelization opportunities to the compiler, which may result in a myriad of potential parallel threads. The selection and scheduling of parallel threads is expected to have a great impact on the program's performance.

In this paper, we discuss several issues regarding compiler optimizations for concurrent multithreading architectures with hardware support for speculative execution and for run-time data dependence check. We use two particular designs, namely *multiscalar*[11] and *superthreaded processors*[12], as examples to show the implications of such hardware on compiler techniques. In the next section, we describe these two microarchitectures and their execution models. In Section 3, we examine how data dependence analysis can be applied to while-loops and how the removal of loop-carried data dependences can improve the effectiveness of speculative execution of while-loops. In Section 4, we discuss how static data dependence analysis can reduce the need for synchronization and the frequency of incorrect speculation. In Section 5, we explore the issue of parallel threads selection. In Section 6, we describe our current experimentation effort. We summarize our discussion in Section 7.

2 Speculative Concurrent Multithreaded Architectures

2.1 Multiscalar architecture

The multiscalar paradigm [3, 11] exploits thread-level parallelism with aggressive hardware support for both control and data speculation. The compiler for the multiscalar processor must partition the control flow graph of a program into threads, each to be executed by a processing unit at run-time. The control and data flow information between threads is stored in *thread descriptors*. With the help of the thread descriptors, the hardware of the multiscalar processor can rapidly traverse the control flow graph of a program and assign threads to processing units on the fly.

Figure 1 shows the general microarchitecture of a multiscalar processor. The processor consists of multiple processing units, which are connected to each other with a unidirectional ring. Each processing unit has its own register file and functional units. A processing unit can pass register data to its down-stream processing units via the

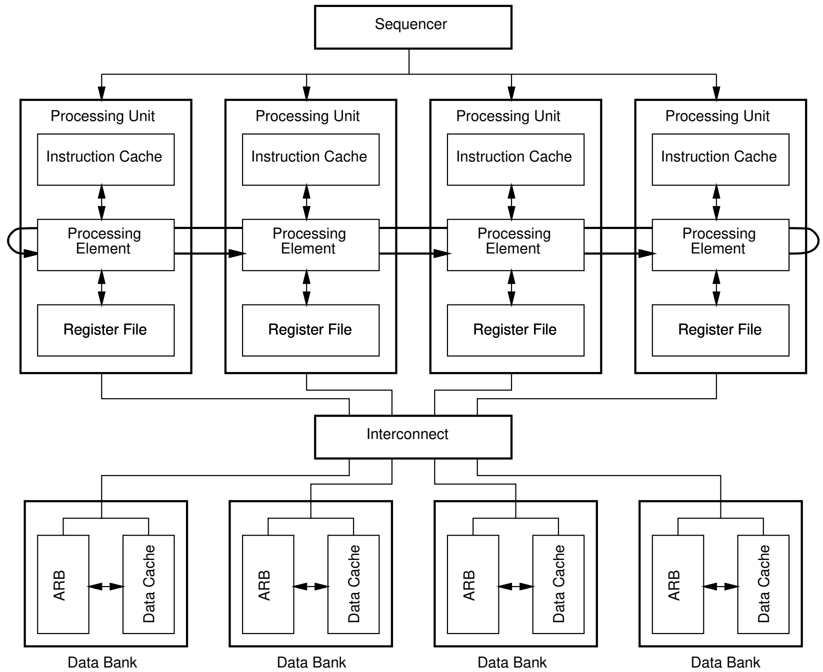

Fig. 1. The microarchitecture of the multiscalar processor [8]

unidirectional ring connection. The sequencer reads information from the thread descriptors and assigns the threads to the processing units in the sequential order of the original program.

To exploit more potential parallelism between the threads, the multiscalar processor allows multiple threads to be executed in parallel with speculation on both control dependences and data dependences. For control speculation, the sequencer predicts which thread should be executed next according to the control flow and assigns that thread to the next processing unit down-stream. If the control speculation later turns out to be incorrect, the processor will squash the speculative thread and its following threads, and resume the correct thread sequence. For data speculation, a thread can load data from a memory location with the expectation that the concurrent predecessor threads will not store a value to the same memory location later. However, if any predecessor thread executes a store operation to the memory location, i.e., if a data dependence is violated, the later thread must be squashed and re-started from the beginning of the thread.

The multiscalar processor uses an *Address Resolution Buffer* (or ARB) to hold the results of the speculative memory operations by the currently active threads and to detect violations of data dependences. The store data held in the ARB can be written back to the data cache only when the thread that executes the store operation becomes a non-speculative thread. The ARB also keeps track of all load and store operations performed by each active thread. A data dependence violation is detected if a thread writes to a memory location whose corresponding ARB entry records an earlier load operation by a successor thread.

2.2 Superthreaded architecture

The superthreaded processor [12] is similar to the multiscalar processor, but it does not speculate on data dependences. Instead, the superthreaded processor performs run-time data dependence checking for load operations. If a load operation is detected to be flow dependent on a store operation by a predecessor thread, it waits for the stored data from the predecessor thread. Checking and enforcing data dependences at run-time can avoid squashing caused by data dependence violations as in the multiscalar, and it can reduce the hardware complexity of detecting memory dependence violation.

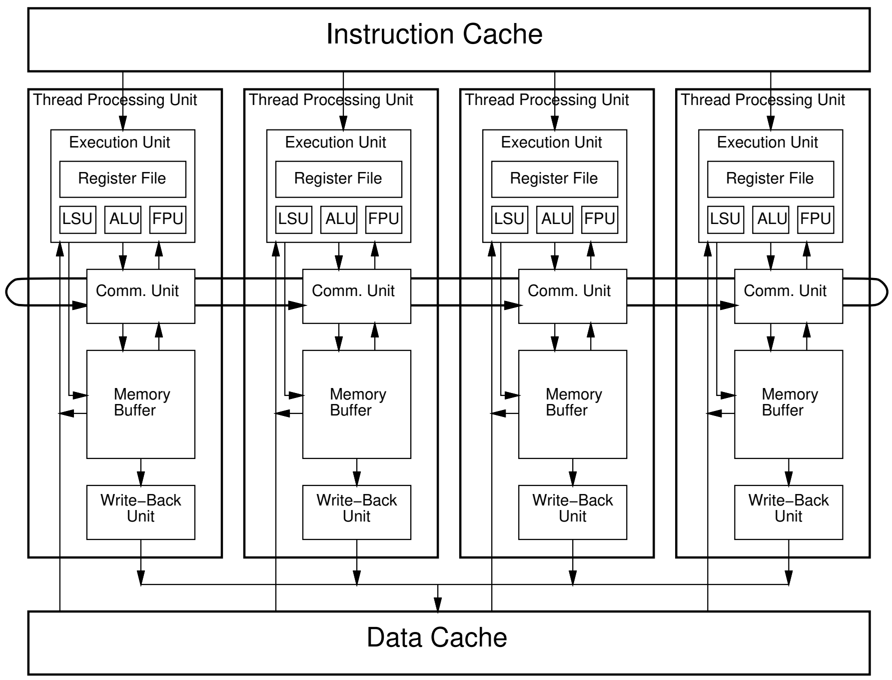

Fig. 2. The microarchitecture of the superthreaded processor

Figure 2 shows the microarchitecture of a superthreaded processor. Like a multiscalar processor, the multiple processing units are connected with a unidirectional ring. Each processing unit can forward the addresses and the data of its store operations to the down-stream units via the unidirectional ring connection. Each processing unit has its own memory buffer to keep the addresses and the data of its own store operations as well as those sent by the up-stream units.

The superthreaded processor uses a thread pipelining execution model to initiate new threads and to enforce data dependences between concurrent threads. As shown in Figure 3, the execution of a thread is partitioned into *continuation stage*, *target-store-address-generation (TSAG) stage*, *computation stage*, and *write-back stage*. The continuation stage is responsible for computing recurrence variables, such as loop index variables, and forking the next thread. A thread can fork a successor thread with speculation on the control flow. The TSAG stage computes the addresses of store operations upon which the successor threads could be data dependent. Those addresses are

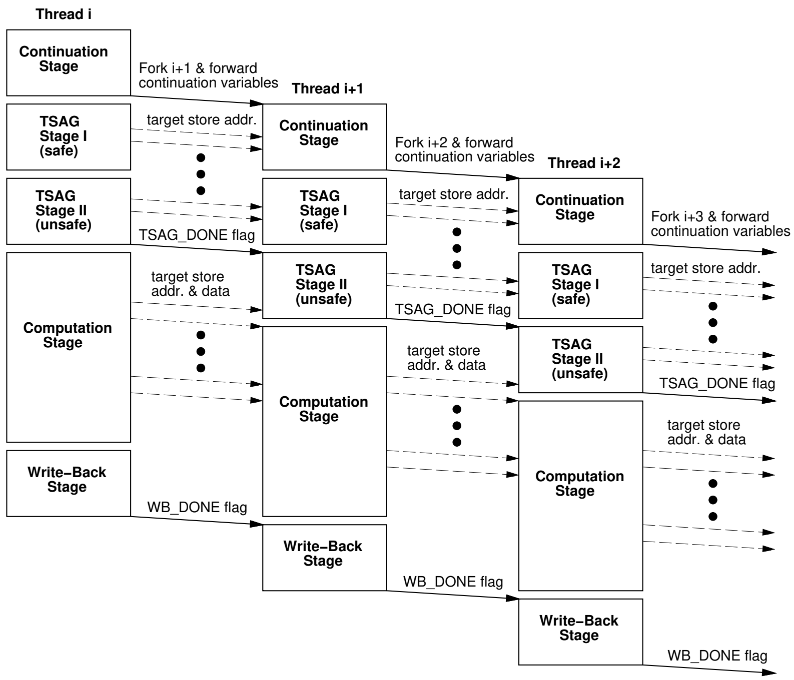

Fig. 3. The pipelined execution of the superthreaded architecture

called *target store addresses* and are forwarded to the memory buffers of the successor threads for run-time dependence checking. To guarantee the correctness of run-time dependence checking, a thread cannot perform any load operation that may be dependent on its predecessor threads until the predecessor threads complete the TSAG stage.

The computation stage performs the remaining computation of the thread. When a thread executes a load operation whose address matches that of a target store entry in the memory buffer, the thread either reads the data from that entry if it is available or waits until the data is received by the thread's processing unit. The write-back stage is performed by the write-back unit automatically after a thread completes its execution. All the data written by the thread to the buffer are to be written back to the data cache at this stage. The write-back stages of contiguous threads are performed in the program sequential order to preserve the non-speculative memory state and to honor the output and anti-dependences between the threads.

3 Analyzing and Removing Loop-Carried Data Dependences in While-Loops

Multiscalar and superthreaded architectures automatically squash an incorrectly speculated thread. As a result, the compiler can focus on loop-carried data dependences when analyzing the parallelism, leaving the loop-carried control dependence to the speculation hardware. Loop-carried data dependences penalize the speculative execution of while-loops mainly in two ways. The true dependences, i.e., flow dependences between loop iterations, reduce the degree of overlap between parallel threads. No hardware mechanisms alone can eliminate such a performance bottleneck. The output and anti-dependences also penalize the performance of such architectures, because the multiple copies of the same variable at different processing units must be written back to the data cache in the sequential order, which is a potential performance bottleneck.

In this section, we first present a technique for pointer arithmetics removal which enables data dependence analysis on variables referenced through pointer arithmetics. At the same time, this technique also removes loop-carried data dependences due to such pointer arithmetics. We then discuss the removal of output and anti-dependences via variable privatization, an issue not examined previously in the context of while-loops. In this paper, a while-loop refers to any loop whose iteration count is not determined before the loop exit condition is tested true.

3.1 Pointered arrays subscriptization

A central issue in identifying parallel threads is the analysis of loop-carried data dependences. Traditional data dependence analysis assumes that arrays are indexed by subscripts. In C programs, most often array references are not indexed by well-formed linear subscripts, but rather are represented by pointers. Pointer arithmetic is used to index through array sections. If not transformed, such references would render the traditional algorithms useless. By rewriting pointered array references in a subscripted form, a process called *subscriptization*, the compiler can apply known algebraic algorithms for the data dependence test [1]. The previous techniques for cleaning up array subscripts do not deal with array references which are *not* already in subscripted forms [5].

The code segment in Figure 3.1(a) is from `eqntott` in SPEC92 benchmarks. For clarity of exposition, we use source programs as examples whenever appropriate. In

```
for (j = lo = base; (lo += qsz) < hi;)
   if ((*qcmp)(arr_j, lo) > 0)
      j = lo;
   if (j != base)
   {   /* swap j into place */
      for (i = base, hi = base + qsz; i < hi;)
      {    c = *j;
          *j++ = *i;
          *i++ = c;
      }
   }
               (a)
```

```
for (arr_j = lo = base; (lo += qsz) < hi;)
   if ((*qcmp)(j, lo) > 0)
      arr_j = lo;
   if (arr_j != base)
    {  for (inew = 0; inew < qsz; inew++)
       {  c = arr_j[inew];
          arr_j[inew] = arr_i[inew];
          arr_i[inew] = c;
       }
    }
                      (b)
```

Fig. 4. Transforming a code segment from `eqntott`.

this example, i and j are pointers to two array sections. Whether the same array elements are modified and used in different loop iterations determines whether parallel threads should be created to execute different iterations. For this case, we first perform symbolic and predicate analysis to determine that, within the second for-loop, the two array sections addressed by i and j are separate, because their beginning positions are apart by at least the value of qsz. We then create two new array names, arr_i and arr_j, to represent these two sections. In order to apply traditional data dependence test algorithms, we transform the code segment as shown in Figure 3.1(b). Note that, if the compiler cannot determine whether array sections addressed by different pointers overlap, the above transformation is still valid. However, arr_i and arr_j will be potential aliases in this case. Since we have determined that arr_i and arr_j are separate, we can safely apply traditional data dependence test algorithms to these two arrays separately, which show that the array references do not cause loop-carried data dependences.

Pointer arithmetics removal also serves as one way, among others, to eliminate artificial loop-carried flow dependences caused by updates to pointer values. Such pointer updates may severely decrease the parallel overlap between successive loop iterations. In the example given above, the increments of i and j create loop-carried flow dependences which reduce the parallel overlap of successive iterations. The increment of i in particular nearly sequentializes the loop because the i value is used for comparison with hi in every iteration. After we transform the array references to subscripted forms, the updates to i and j are removed. The values of i and j are dead after the second for-loop. So, their last values need not be saved. After c is privatized (*c.f. next subsection*), $inew++$ becomes the only sequential portion of the second for-loop.

Both multiscalar and superthreaded processors provide run-time data dependence check to guarantee safe computation. Moreover, their synchronization hardware allows parallel threads to have loop-carried data dependences. The degree of parallelism is the main concern. Hence, even if certain loop-carried data dependences due to *array* references do exist, pointer arithmetics removal may still contribute to the elimination of sequential bottlenecks.

Our general strategy for the elimination of pointer arithmetics is as follows:

1. Identify pointers whose values are incremented or decremented in the loop body.
2. Analyze the address range of these pointers for their degree of overlap, which affects the profitability of the transformation.
3. Designate a primary induction variable, say i, whose initial value is 0.
4. Apply induction variable recognition algorithms to the pointers identified above to see whether they are induction variables whose values can be written in closed forms in terms of i.
5. If the last step succeeds and the transformation is estimated as profitable, a new array name is created for each pointer variable identified above. Each pointer dereference in the loop body is replaced by array references indexed by the closed forms computed above.
6. Insert code after the loop to save last values of pointers if they are live.

3.2 Variable privatization

If a variable's updated value in one iteration of the while-loop can never reach a later iteration, then a distinct copy of the variable can be created for each processing unit, either by register allocation or by renaming. Loop-carried output and anti-dependences can thus be eliminated. Both multiscalar and superthreaded hardware dynamically rename variables in their buffers, i.e., the *memory buffer* in the superthreaded processor and the *address resolution buffer (ARB)* in the multiscalar. Hence, unlike traditional multiprocessors, loop-carried output and anti-dependences do not require explicit synchronization on superthreaded and multiscalar architectures.

Nonetheless, variable privatization by compilers is still desirable for both architectures. When a compiler privatizes a variable, it can be stored in a register instead of in the ARB or the memory buffer with a much faster access time. In order to simplify the hardware for run-time dependence check, the buffer sizes should be kept small. However, too small a buffer size will incur too frequent buffer overflows. When a buffer overflow occurs, the thread must be stalled until all early threads are completed because the run-time dependence check will no longer function properly. Therefore, the less demand on memory buffer or ARB, the less frequently the buffer will overflow. Moreover, any data that are updated in the buffer must eventually be written back to the memory. This write-back by different threads is sequentialized to guarantee program correctness. In order to reduce the write-back overhead, we also need to keep the size of the memory buffer and the ARB small. By allocating the private variables to registers, the buffer size can be reduced.

The following code segment is from program `eqntott` in SPEC92.

```
int cmppt (a, b)
PTERM *a[], *b[];
     register int i, aa, bb;

     for (i = 0; i < ninputs; i++) {
       aa = a[0]->ptand[i];
       bb = b[0]->ptand[i];
          if (aa == 2)
            aa = 0;
          if (bb == 2)
            bb = 0;
          if (aa != bb) {
             if (aa < bb) {
               return (-1);
             }
             else {
               return (1);
             }
          }
    }
    return (0);
}
```

The above for-loop contains premature exits, which is considered as a while-loop in this paper. By speculating on `aa != bb` being false, both the superthreaded processor and the multiscalar can create parallel threads to execute the loop iterations. The variables aa and bb are privatizable and, hence, they can be allocated to registers on different processing units.

If the compiler allocates a privatized variable to a register, it must examine whether the variable is live after the termination of the while-loop. If so, then the last value must be written back to the memory location. If the variable is unconditionally updated in every iteration, then the last thread, which aborts all successive speculated threads, is responsible for the store. The store instruction can be inserted in the beginning of the branch which is executed when the exit condition is tested true.

If the variable is conditionally updated in each iteration, then it is better not to privatize the variable. The following code segment from program `compress` in SPEC95 shows such an example. Variable `p` is not suitable for privatization, although it is only involved in output dependences.

```
char * rindex(s, c)
register char *s, c;
{
char *p;
for (p = NULL; *s; s++)
    if (*s == c)
               p = s;
return(p);
}
```

4 Reducing Synchronization and Misspeculation Penalties

In the superthreaded processor, data synchronization is done by forwarding *target store* addresses to succeeding threads in the *target-store-address-generation* stage. If a succeeding thread issues a read reference whose address matches a forwarded target store address, then the thread must wait until data is forwarded or until the preceding thread is completed. If the preceding thread updates the data more than once, only the last updated value needs to be forwarded. Here we see two potential sources of performance penalties due to synchronization. One is the cost of forwarding target store addresses and the other is the data waiting time. However, if the succeeding threads do not have matches of the target store addresses at run-time, then no data waiting penalty will occur.

On the multiscalar, data synchronization for flow dependences between threads can be done in two forms, either by waiting at ARB or by register forwarding. There are no synchronization instructions to force a thread to wait. Instead, the ARB hardware can speculate, based on memory reference history, that a flow dependence may occur at a particular memory address [9]. The hardware then forces the thread which may be the sink of the dependence to wait for a flag associated with that memory address. As soon as a preceding thread stores data to that address, the flag is raised, which permits the waiting thread to proceed. Note that if a preceding thread must store the data several times to the memory address, the flag may be raised before the last store is completed. The ARB will detect the violation of flow dependence and the processing unit will squash the thread which is the sink of the dependence. Also note that if the ARB speculates a synchronization incorrectly, the waiting thread will wait until its preceding thread finishes execution and is released. Both incorrect speculations and premature flag-raising can potentially be costly.

Synchronization can also be done via register forwarding. On the multiscalar, each processing unit has its own register file. Each register's content is forwarded from one processing unit to the next, using the same register number. Each register has a bit which is automatically set and tested by the hardware to indicate the availability of the forwarded value. These bits are transparent to the software. The registers of a processing unit can be dynamically configured at the thread creation time, with *creation masks*, as *filtered* and *nonfiltered* registers. A filtered register is forwarded when the register is written within the current thread. A nonfiltered register is forwarded without being modified. The compiler can use the register files to satisfy a loop-carried flow dependence as follows. The thread which writes a variable which is to be read by a later thread writes the value to a filtered register, say $r4$. The thread that uses that value then reads $r4$ instead of reading from the ARB. If the threads which execute the source and the sink are not immediately adjacent to each other, those threads in between should mark $r4$ as nonfiltered. Using this form of synchronization, the compiler must precisely analyze the loop-carried flow dependence, and the dependence distance must be calculated. Also, if the variable causing the dependence can potentially be updated several times before it is read by the dependence sink, then the register should not be forwarded before the last update is done.

The problem of identifying last writes for the purpose of synchronization has previously been discussed [8, 6]. Analysis of last writes has also been proposed to support array privatization and other optimizations [7] When it is not clear which writes are last writes, the compiler needs to find a program point that *post-dominates* all potential last writes in order to safely forward the data. The following code segment serves as an example.

```
for (i = 0; i++) {
      if (!c1]) break;
      ...
      c1 = ...
      ...
      if (c2) c1 = ...
      /* place to forward c1 */
      ...
}
```

For the superthreaded processor, the general compiler strategy for data forwarding is as follows:

1. Identify stores that are potential sources of loop-carried flow dependences between threads. Mark them as target stores.
2. Group those stores which access the same location.
3. For each group, identify the program point for forwarding the data.
4. Insert code in the target-store-address-generation stage to forward target store addresses.
5. Insert code to forward target store data at the selected program points.

In the example above, there are two target stores to $c1$. The address of $c1$ is a target store address that should be forwarded in the target-store-address-generation stage. The data of $c1$ is forwarded at the marked program point.

For the multiscalar, the general compiler strategy for data forwarding is as follows:

1. Identify stores that are potential sources of loop-carried flow dependences between threads.

2. Group those stores which access the same location.
3. For each group, allocate a register and mark it as filtered.
4. For that group, identify and mark the program point for forwarding the data.
5. Insert code to write the data to the filtered register at the marked forwarding program point.
6. Insert code to read the forwarded register value.

In the example above, at the marked forwarding point, an instruction should be inserted to write $c1$ to a filtered register, say $r4$. The read of $c1$ in the if-statement is then replaced by a read from $r4$.

5 Partitioning Parallel Threads

By combining multiple processing units with hardware speculation, concurrent multithreading architectures provide many opportunities for parallel execution which are unavailable to superscalar processors and traditional multiprocessors. Particularly important is their highly efficient thread creation. Starting a new thread can take as few as a couple of instructions, which makes it profitable to execute fine-grain threads. The low thread start-up overhead also makes loop-carried data dependences more tolerable. Loops with a modest amount of overlap among the iterations may still be speeded up by having the iterations executed in parallel. Still, in order to maximize the performance, it is important that processing units spend less time on synchronization. Moreover, it is essential that the idle time of the processing units is minimized. A processing unit becomes idle if its current thread is forced to wait for the completion of the previous thread. Such waiting may be either due to a buffer overflow, the presence of flow dependences between threads, or load unbalancing between threads. On the multiscalar, when the hardware detects that two threads violate data dependences, the succeeding thread must be squashed, which increases the unit's idle time.

Clearly, the partitioning and scheduling of parallel threads will have an impact on both the synchronization overhead and the idle time of processing units. In order to explore this issue further, we review the thread creation models of the multiscalar and superthreaded processors.

On the superthreaded processor, each thread is created by an explicit *fork* <*address*> instruction. Correspondingly, an explicit *abort* instruction squashes all future threads. This instruction is inserted by the compiler at the program points where branching decisions are made and an incorrect speculation can be detected. On the multiscalar, a *task tag* is inserted at the program point where a thread may begin its execution. A task tag contains several control targets which the current program point may potentially lead to. At run time, the sequencer, using a prediction table, picks one of the targets at which it starts a new thread. Figure 5 shows an example, in which thread 1 and thread 2 will run in parallel, but which branch becomes thread 2 is determined by hardware using execution history as a guide. Thread squashing is also done by the sequencer when it finds a branch prediction is incorrect. In Figure 5, suppose target $t1$ was speculated but $t2$ is actually the branch target, then the sequencer automatically squashes thread 2 starting at $t1$.

From the above, it is clear that for the superthreaded processor, the compiler is responsible for both thread partitioning and thread speculation, while for the multiscalar, the compiler is responsible only for thread partitioning. Insertion of target tags is straightforward once the thread boundaries are identified.

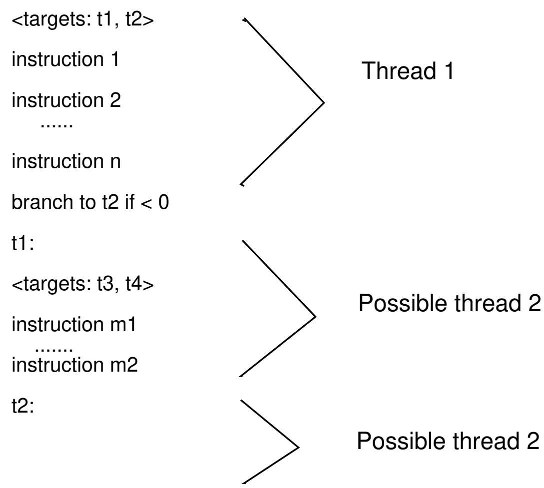

Fig. 5. Multiple potential thread targets on the multiscalar processor

On both multiscalar and superthreaded processor, each thread can fork only one successor thread during its lifetime. This restriction, a clear distinction from traditional multiprocessing models, is imposed so that the speculation hardware can be implemented efficiently. Due to the speculation support, virtually any kinds of program segments can be executed as individual threads, subject to the *single successor rule* mentioned above. The followings are several common examples:

- parallel sections;
- loop iterations, including while-loops; and
- function calls.

In the above, parallel sections can take a variety of forms. A section can be as small as a basic block, or as large as a function call or a whole loop. A section can also be one branch of a conditional statement.

The single successor rule, however, has a few important implications:

- A thread cannot simultaneously speculate both branches of a branch instruction.
- For a nested loop, once an outer loop iteration forks a new thread from the outer loop, it will not fork new threads to execute inner loop iterations.
- For a loop which contains function calls, once a loop iteration forks a new thread to execute the next iteration, it cannot fork a new thread to execute any section of the loop body, e.g. a function call.
- If threads are forked to execute function calls, then these threads cannot fork new threads to execute loops which may be recognized as parallel.

These restrictions are imposed as a tradeoff for efficient thread start-up and speculation. Due to these restrictions, how threads are partitioned may have a significant impact on a program's performance. For example, if the compiler knows that the inner loop has more parallelism than the outer loop, then threads should be created for the inner loop instead of the outer loop, and vice versa. Thread partitioning is perhaps the most complex issue in compilation for the multiscalar and superthreaded processors as in multiprocessors. Currently we are pursuing the following studies:

– Estimating the working set size of a thread to allow the working set to fit in the memory buffer or the ARB;
– Estimating the amount of parallelism based on the data dependence graph;
– Constructing a hierarchical task graph [4] which reflects speculation possibilities;
– Affirming the existence of data dependences in addition to affirming data independences, which is important for parallelism estimate and for avoiding excessive incorrect data speculation on the multiscalar.
– Interprocedural analysis to support the above studies.

6 An Integrated Compiler for Experimentation

Since concurrent multithreaded processors, such as the multiscalar and the superthreaded processors, are basically microprocessors which allow a large number of instructions issued per machine cycle, it is important to compare their performance against superscalars and VLIW architectures. Interestingly, the multiscalar and superthreaded architectures also have a strong resemblance to a small-scale, tightly-coupled multiprocessor with additional support for thread-level speculation. Hence, many traditional compilation techniques for both superscalars, VLIW and multiprocessors could be used for such architectures as well.

In general, traditional compilation techniques for superscalars and VLIW focus primarily on exploiting instruction-level parallelism in the back end compilers. On the other hand, traditional compilation techniques for multiprocessors focus primarily on exploiting loop-iteration level parallelism in the front end parallelizing compilers. To allow both loop-iteration level and instruction-level parallelism to be exploited on concurrent multithreaded architectures, we need an integrated compiler that has at least the capability of both the front end parallelizing compilers and the back end compilers that deal with the instruction-level parallelism.

Because it is a major effort, and also very time consuming, to develop such an integrated compiler from scratch that can match today's state-of-the-art parallelizing compilers and back end compilers, we instead leverage existing compilers for our purpose. The Agassiz compiler project is an effort to integrate state-of-the-art parallelizing compilers to back end compilers. A data structure that can store high-level information (HLI) is developed which allows the results of elaborate program analysis in the front end parallelizing compiler to be exported and used in the back end compiler to exploit instruction-level parallelism and machine-specific optimizations.

The current Agassiz prototype (Figure 6) targets both Fortran and C. It leverages gcc/g77 common back ends and a parallelizing compiler that can deal with both C and Fortran. The HLI data structure includes loop-carried data dependences, array dataflow information, and alias information for high-level program structures such as loops and procedures. It can be easily exported from any parallelizing compilers and imported into the gcc/g77 back end compiler.

The front end parallelizing compiler and the gcc/g77 back end compiler are modified to include compiler techniques specific to the multiscalar and superthreaded architectures as described earlier.

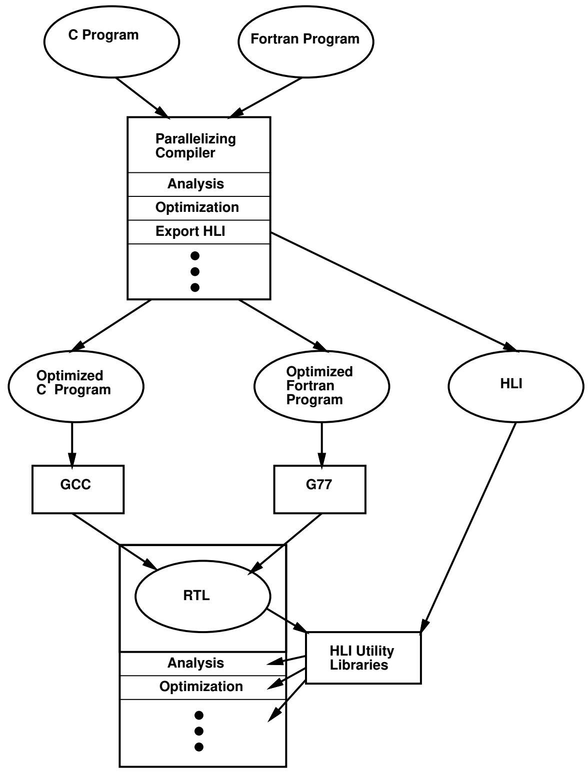

Fig. 6. The Agassiz compiler system

7 Conclusion

The multiscalar and superthreaded architectures provide hardware speculation support for concurrent multithreading. The multiscalar relies more on hardware for data dependence check and for thread speculation, while the superthreaded processor relies more on compiler information. The followings are a list of compiler techniques which can benefit both architectures.

- Pointered array subscriptization to convert pointered arrays to subscripted arrays.
- Pointer arithmetics removal to reduce loop-carried data dependences.
- Variable privatization to reduce buffer overflow and write-backs.
- Last-write identification for efficient data forwarding between successive threads.
- Program parallelism and workload analysis to assist thread partitioning.

Since the superthreaded architecture does not rely on hardware to speculate on data dependences, it requires more compiler analysis on data dependences for synchronization. A more precise data dependence analysis at compile time will also allow more parallel threads to be created. On the other hand, the multiscalar aggressively speculates on data dependences and thus aggressively creates parallel threads. A more precise data dependence analysis at compile time can lead to better thread partitioning and reduced thread squashing.

Lastly, several of the compiler techniques presented in this paper can potentially benefit superscalar processors. It is interesting to compare the impact of these techniques on concurrent multithreading versus superscalars.

References

1. Randy Allen and Steve Johnson. Compiler C for vectorization, parallelization, and inline expansion. In *Prof. of SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 241–249, June 1988.
2. Pradeep K. Dubey, Kevin O'Brien, Kathryn O'Brien, and Charles Barton. Single-program speculative multithreading (SPSM) architecture: Compiler-assisted fine-grained multithreading. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 109–121, June 27–29, 1995.
3. Manoj Franklin and Gurindar S. Sohi. The expandable split window paradigm for exploiting fine-grained parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 58–67, May 19–21, 1992.
4. M. Girkar and C. Polychronopoulos. The HTG: An intermediate representation for programs based on control and data dependences. CSRD Technical Report No. 1046, Univ. of Illinois at Urbana-Champaign, 1991.
5. Justiani and L. J. Hendren. Supporting array dependence testing for an optimizing/parallelizing c compiler. In *Proc. of the 1994 International Conference on Compiler Construction. Volume 749 of Lecture Notes in Computer Science*. Springer Verlag, April 1994.
6. Z. Li. Compiler algorithms for event variable synchronization. In *Proc. of the Fifth International Conference on Supercomputing (ACM)*, June 1991.
7. D.E. Maydan, S.P. Amarasinghe, and M.S. Lam. Array data-flow analysis and its use in array privatization. In *Proc. of the 20th ACM Symp. on Principles of Programming Languages*, pages 2–15, January 1993.
8. S. P. Midkiff and D. A. Padua. Compiler algorithms for synchronization. *IEEE Transactions on Computers*, C-36(12):1485–1495, December 1987.
9. Andreas I. Moshovos, Scott E. Breach, T. N. Vijaykumar, and Guri. S. Sohi. Submitted for a blind review to a conference.
10. B. A. Nayfeh and K. Olukotun. Exploring the design space for a shared-cache multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 166–175, April 1994.
11. Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 22–24, 1995.
12. Jenn-Yuan Tsai and Pen-Chung Yew. The superthreaded architecture: Thread pipelining for run-time data dependence checking and control speculation. Technical Report TR 96-037, University of Minnesota, Dept. of Computer Sicence, Minneapolis, MN, 1996.