Program Optimization for Concurrent Multithreaded Architectures *

Jenn-Yuan Tsai[†], Zhenzhen Jiang[‡], and Pen-Chung Yew[‡]

[†]Department of Computer Science
University of Illinois
Urbana, IL 61801
j-tsai1@uiuc.edu

[‡]Department of Computer Science
University of Minnesota
Minneapolis, MN 55455
{zjiang,yew}@cs.umn.edu

Abstract. This paper presents some compiler and program transformation techniques for concurrent multithreaded architectures, in particular the superthreaded architecture [9], which adopts a thread pipelining execution model that allows threads with data dependences and control dependences to be executed in parallel. In this paper, we identify several important program analysis and transformation techniques that allow the superthreaded architecture to exploit more parallelism in programs with less run-time overhead. We evaluate the performance of the superthreaded architecture and the effectiveness of the program transformation techniques by manually compiling several benchmark programs and running them through a trace-driven, cycle-by-cycle superthreaded processor simulator. The simulation results show that a superthreaded processor can achieve promising speedups for most of the benchmark programs with the proposed program transformation techniques applied.

1 Introduction

Single-threaded execution model, i.e., using a single program counter to traverse the control flow of a program, has been used in the processor architectures long since the first processor was built. The currently predominant superscalar processor architectures, even though they can issue and execute multiple instructions per cycle, are still using the single-threaded execution model. It is very easy for the compiler to translate a source program into its machine codes for processors with a single-threaded execution model, since it is the execution model assumed by most of the existing programming languages. It is also easier to design and implement a single-threaded processor because no data communication and resource arbitration are needed between multiple threads. However, as the issue rate of future microprocessors continues to increase, it has become more and more difficult for the compiler to find sufficient instruction-level parallelism

* This work is supported in part by the National Science Foundation under Grant No. MIP 9610379; by the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order No. D 346; and by a gift from Intel Corporation.

from a single instruction stream. In addition, the hardware cost for performing data dependence checking and instruction reordering at run-time will grow quadratically as the issue rate increases. Therefore, it is very difficult to build a single-threaded microprocessor with an issue rate greater than ten instructions per cycle.

Recently, a number of concurrent multithreaded execution models [5, 8, 10, 3, 2, 9] have been proposed as an alternative to the single-threaded execution model for the future generations of microprocessors. Using multiple threads of control to fetch and execute instructions from different locations of a program simultaneously allows compilers and processors to exploit more parallelism from multiple instruction windows. In a sense, the concurrent multithreaded architecture (CMA) is similar to having a multiprocessor on a single chip. However, CMAs are more tightly coupled than having a multiprocessor on a single chip. Multiple thread processing units can communicate and synchronize through special hardware support, or directly through their register files instead of through memory system as in a multiprocessor. Hence, it can support more aggressive data communication and synchronization mechanisms between threads. In addition, the concurrent multithreaded processors are targeted for general-purpose programs, which are mostly written in C and usually have complicated control flows and implicit data dependences that cannot be analyzed by the compiler. Most of the proposed multithreaded architectural models provide some hardware support for thread-level speculation [5, 8, 2, 9] and run-time data dependence checking between threads. [8, 2, 9].

To take advantage of multiple threads of control, the compiler needs to generated multiple-threaded codes from source programs. For numerical programs written in Fortran, we can leverage existing powerful parallelizing compilers which are originally developed for multiprocessors to extract loop-level parallelism and generate multiple-threaded code. For non-numerical programs which are mostly written in C or C++, existing parallelizing compilers lack effective techniques to parallelize those programs because of the complicated data structures and control flows in the programs. These complicated data structures and control flows often force a compiler to made conservative assumptions which can reduce run-time performance substantially. Concurrent multithreaded architectures provides necessary hardware support to perform run-time dependence checking and/or to speculate on the control or data dependences. Therefore, new compiler techniques are needed to take advantages of such hardware support. Those compiler techniques are more aggressive than those in conventional parallelizing compilers.

In this study, we use a concurrent multithreaded architecture, called the *Superthreaded architecture*, proposed in [9]. The superthreaded architecture uses a thread pipelining execution model to enhance overlapping between threads and to facilitate data dependence enforcement between threads through hardware supported thread-level control speculation and run-time data dependence checking. Although the program transformation techniques identified through the study are targeted for the superthreaded architecture, most of them are also very useful for the other concurrent multithreaded architectures.

In the rest of this paper, Section 2 gives an overview of the superthreaded architecture and its execution model. In Section 3, we present the program transformations techniques for the superthreaded execution model. Section 4 summarizes the case study of generating superthreaded codes for the selected benchmark routines and shows the simulated execution results of the transformed programs. We conclude the paper in Section 5.

2 An Overview of the Superthreaded Architectural Model

In its general form, a superthreaded processor consists of a number of thread processing units, which are connected to each other through a unidirectional ring as shown in Figure 1. The thread processing units share the instruction cache and the data cache. Each thread processing unit has its own program counter, local register file, and execution unit. In addition, each thread processing unit has a communication unit for transferring commands and data between thread processing units, and a memory buffer for performing run-time data dependence checking and for buffering speculative store data. There is also a shared register file which includes some global registers and a lock register.

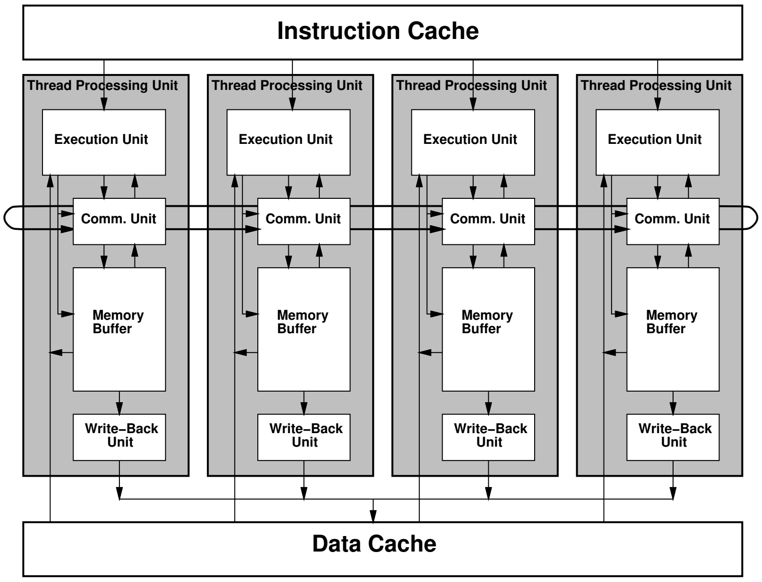

Fig. 1. The organization of a superthreaded processor

The compiler statically partitions the control flow graph of a program into threads. Each thread corresponds to a portion of the control flow graph. A program starts execution from its entry thread. The entry thread can then fork a successor thread on the next thread processing unit, which in turn can fork its own successor thread. This process continues until all the thread processing units are busy. Forking a thread only requires a few cycles, which allows a thread to have a very light weight.

When multiple threads are executing on a superthreaded processor, the oldest thread in the sequential order is referred to as the *head thread*. All the other threads derived from it are called *successor threads*. After the head thread completes its computation, it will retire and release the thread processing unit. Its immediate successor thread becomes the new head thread. The completion and retirement of the threads must follow the program sequential execution order.

2.1 Thread Pipelining Execution Model

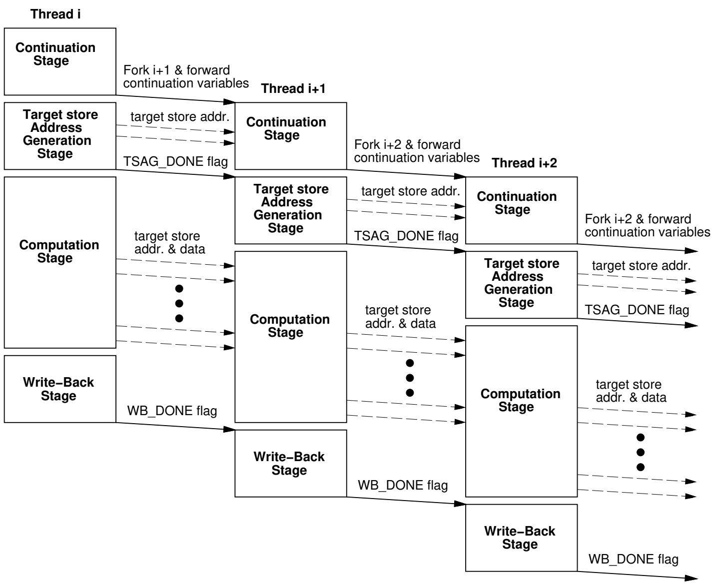

Fig. 2. The pipelined execution of contiguous threads

The superthreaded architecture uses a thread pipelining execution model to enhance overlapping between threads and to facilitate data dependence enforcement between threads. As shown in Figure 2, 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 for forking a new thread on the next thread processing unit. Upon the execution of the fork instruction, all of the recurrence value computed at the continuation stage, as well as the target store addresses and data (to be described later) received from the predecessor thread, will be forwarded to the successor thread through the communication unit. A thread can fork a successor thread with control speculation, such as in a WHILE loop. If, later, the control speculation is evaluated to be incorrect, the thread will issue an *abort_future* instruction (generated by the compiler) to kill all the successor threads.

The TSAG stage computes addresses for store operations upon which the concurrent successor threads could be data dependent. Those addresses are called *target store addresses*. They are generated by the compiler through a data dependence analysis and will be forwarded to the memory buffers of the successor threads for run-time dependence checking. An *allocate_ts* instruction is used to allocate an entry in the memory buffer and to forward the target store address to the successor thread. To guarantee the correctness of run-time dependence checking, a successor thread cannot perform any load operation which potentially can be data dependent on those store operations until its predecessor thread has completed the TSAG stage and forwarded all of the target store addresses to its memory buffer. This is enforced by placing a synchronization instruction *release_tsag_done* at the end of the TSAG stage to send a *tsag_done* flag to its successor thread, and by placing a *wait_tsag_done* instruction in the successor thread before its load operations that may be data dependent on its predecessor thread.

The computation stage performs the main computation of the thread. When a thread executes a load operation whose address matches that of a target store entry in its memory buffer, the thread will either read the data from the entry if it is available or wait until the data is forwarded by the predecessor thread. When a thread performs the last store operation to a target store address, it will use a *store_ts* instruction to forward the store data to the successor thread. If a thread is completed normally without being aborted by the predecessor thread, it will end with a *stop* instruction. After executing the *stop* instruction, a thread will wait until it becomes the head thread and then perform the write-back stage.

In the write-back stage, all the store data in the memory buffer will be written to the cache memory at this stage. The write-back stages of contiguous threads are performed in the program sequential order to preserve non-speculative memory state and to eliminate output and anti-dependences between threads. After performing the write-back stage, a thread processing unit will retire the thread and become idle until it is scheduled a new thread again.

2.2 Superthreaded Programs

The code segment shown in Figure 3 is one of the most time-consuming loops in the SPECint95 benchmark *124.m88ksim*. This is a while loop with exit conditions in the loop head as well as in the loop body. There is a potential read-after-write data dependence across loop iterations caused by the variable `minclk`.

Figure 4 shows the superthreaded code for the loop. In this code, each thread corresponds to a loop iteration. In the continuation stage, each thread increments the recurrence variable `i` and forwards its new value to the next thread processing

```
while ( funct_units[i].class != ILLEGAL_CLASS ) {
    if( f->class == funct_units[i].class ) {
        if ( minclk > funct_units[i].busy ) {
            minclk = funct_units[i].busy;
            j = i;
            if ( minclk == 0 ) break;
        }
    }
    i++;
}
```

Fig. 3. An example code segment from *124.m88ksim*

unit using a *store_ts* instruction. The original value of `i` is saved in `i_1` for later use. The continuation stage ends with a *fork* instruction to initiate the successor thread.

In each thread, there is only one target store corresponding to the update of the variable `minclk`. The address of the variable `minclk` is forwarded to the next thread in the TSAG stage. Since the TSAG is not dependent on predecessor threads, it can proceed immediately after the continuation stage. However, the computation stage needs to wait until it receives the *tsag_done* flag from the predecessor thread. This is enforced by the *wait_tsag_done* instruction.

In the computation stage, a thread first checks if the first exit condition is true. If it is true, the thread will abort the successor threads by using an *abort_future* instruction, and then jump out of the loop. Otherwise, the thread will perform the computation of the loop body. In the computation, the update to the variable `minclk` is performed by a *store_ts* instruction, which will forward the result to the successor threads. If the control path that executes the *store_ts* is not taken, the thread will execute a *release_ts* instruction to release the target store entry so that the successor threads will not wait for the target store data. If both exit conditions are false, the thread will execute a *stop* instruction, and then start the write-back, which is performed by the hardware automatically.

3 Compilation for the Superthreaded Architecture

Although the superthreaded architecture provides hardware support for control speculation and run-time data dependence checking, it relies on the compiler to extract thread-level parallelism and to generate superthreaded codes. There are two major tasks for the compiler: (i) partitioning a sequential program into threads for concurrent execution, and (ii) identifying target stores. Both tasks require powerful program analysis and transformation techniques. Fortunately, many of these techniques have been developed in traditional parallelizing compilers. We can leverage those techniques for superthreaded processors.

```
/* Continuation Stage */
L1:
    i_1 = i;
    store_ts(&i,i_1+1);
    fork L1;

/* Target-Store-Address-Generation Stage */
    allocate_ts(&minclk);
    wait_tsag_done;
    release_tsag_done;

/* Computation Stage */
    if (funct_units[i_1].class == ILLEGAL_CLASS ) {
        abort_future;
        i = i_1;
        goto L2;
    }

    if ( f->class == funct_units[i_1].class ) {
        if ( minclk > funct_units[i_1].busy ) {
            store_ts(&minclk, funct_units[i_1].busy);
            j = i_1;

            /* if minclk is zero, break to terminate search */
            if ( minclk == 0 ) {
                abort_future;
                i = i_1;
                goto L2;
            }
        } else
            release_ts(&minclk);
    } else
        release_ts(&minclk);

    stop;

/* Write-back Stage */
/* -> performed automatically after stop */
/* End of thread pipelining */

L2:
```

Fig. 4. The superthreaded code for the example shown in Figure 3.

In addition to generating superthreaded codes, the compiler can further enhance parallelism between threads and reduce run-time data dependence checking overhead by applying some specific program transformations for superthreaded processors.

In this section, we first summarize the basic program analysis and transformation techniques required by the compiler to generate the superthreaded code, and then describe the advanced program transformation techniques to enhance the performance of superthreaded processors.

3.1 Basic Compiler Techniques for Superthreading

The compiler partitions a program into threads by examining data dependences between neighboring or repeated portions of its control flow graph. To perform such thread partitioning, the compiler can use the following traditional program analysis and transformation techniques:

- *Function inlining:* If a thread includes function calls that may cause data dependences between threads, the compiler can use function inlining to allow data flow analysis and instruction movement across function boundaries for thread pipelining.
- *Induction variable substitution:* A loop may have several induction variables, which are variables incremented or decremented by a constant value in each iteration. They will cause loop-carried data dependences. The compiler needs to generate target store instructions for those induction variables. To reduce the number of target stores, the compiler can select one induction variable as the primary induction variable, and replace other induction variables with linear functions of the primary induction variable. By applying such induction variable substitution, the compiler needs to generate only one target store for the primary induction variable. The overhead for data forwarding and synchronization can be greatly reduced.
- *Alias analysis:* Programs written in C or C++ usually use many pointers and dynamic data structures. Without any alias analysis, the compiler must assume all pointers are possible aliases when it analyzes data dependences and identifies target stores. Although the superthreaded architecture can handle aliases by performing run-time data dependence checking, more precise alias information will result in fewer target stores generated.
- *Data dependence analysis:* Data dependence analysis is a very important technique used in parallelizing compilers to determine if a loop should be executed in parallel. For superthreaded processors, data dependence analysis can help to identify target stores for run time data dependence checking and synchronization.
- *Variable privatization:* A private variable is a variable whose updated value does not reach a later thread, even though the variable may be written and then read in a later thread. The compiler need not generate target store instructions for the update of the private variables because they will not cause read-after-write data dependence across threads. For this purpose,

the compiler can perform analysis (on both scalars and arrays) to identify privatizable variables [11, 6].

- *Loop unrolling/interchange:* Loop unrolling and loop interchange are common techniques used in parallelizing compilers to increase the granularity of parallel loops. In the thread pipelining execution model, by applying loop unrolling or loop interchange, we can increase the size of the computation stage, which will be executed in parallel, and thus enable a larger portion of threads to be overlapped.

After partitioning a program into threads. the compiler then partitions each thread into multiple stages for thread pipelining. It also generates target store instructions to facilitate run-time data dependence checking. The compiler will try to increase the execution overlap of concurrent threads by minimizing the stalls caused by data dependences between threads. To do this, we will perform statement reordering [1] to schedule target stores as early as possible, and to schedule load instructions that may be data dependent on the target stores of some predecessor threads as late as possible.

3.2 Conversion of Data Speculation to Control Speculation

Some concurrent multiple-threaded architectures, such as *Multiscalar* [4, 8] and *SPSM* [2], provide hardware support for data speculation. With such hardware support, the compiler can speculate on potential data dependences between threads by assuming that they would not be violated, and rely on the hardware to detect violations at run-time. A dependence violation is detected if a thread writes to a memory location after a later thread has read from the same location. When this happens, the hardware will squash the later thread and all of its successor threads.

Hardware support for data speculation can be very expensive, because it needs a buffer (called *Address Resolution Buffer* in multiscalar) to keep all *load* and *store* addresses from all active threads in order to detect data dependence violations. To avoid using expensive hardware for data speculation such as the address resolution buffer, and to avoid wasting useful computation when a thread is squashed due to data dependence violation, the superthreaded architecture *enforces*, rather than *speculates*, data dependences between threads. It uses the memory buffer to perform both run-time data dependence checking and implicit data synchronization, and requires only store addresses and data to be buffered. Since there are far fewer store addresses than load addresses, the size of the memory buffer can be much smaller than the the address resolution buffer in multiscalar.

However, data speculation can be useful for some programs. For example, data dependences may occur only in certain control paths within each thread. By using data speculation, we can exploit the potential parallelism if the control paths that contain the data dependences are not taken at runtime.

For threads that can use data speculation to exploit parallelism, the compiler can perform the data speculation by converting it to control speculation, which is supported by the superthreaded architecture. To speculate on a read-after-write data dependence that will occur only if the predecessor thread takes certain control paths that execute the write, the compiler does not generate any target store addresses for the write. Instead, the compiler inserts an *abort_future* instruction in the control paths that execute the write. If the predecessor thread does take that control path, the *abort_future* instruction will be executed and the successor thread that depends on the write will be squashed and re-executed.

3.3 Distributed Heap Memory Management

Programs written in C often need to allocate dynamic memory space at runtime. Dynamic memory space is allocated and deallocated through the heap memory manager, such as *malloc* and *free* functions provided by the standard C library. The heap memory manager usually uses a free list to keep track of the free memory blocks available for allocation. The allocation or deallocation of a memory block will modify the free list and the memory blocks. Such heap memory management can cause potential data dependences between threads and create a bottleneck for thread parallelization.

To eliminate such data dependences, we can distribute the management of the dynamic heap memory to each thread processing unit; that is, each thread processing unit maintains its own free list to keep track of the free memory blocks owned by the thread. With the heap memory management distributed to each thread processing unit, the allocation and deallocation of memory blocks on concurrent threads can be made to be independent of each other and can be executed in parallel.

To support distributed heap memory management, we need to modify the heap memory management routines. The new heap memory management routines will maintain a free memory block list for each thread processing unit. When a heap memory management routine is called, it must be given the ID of the thread processing unit. The routine will then allocate or deallocate a memory block from or to the free list associated with the thread processing unit. Note that the distributed heap memory management still uses a single heap memory space. A memory block allocated by one thread processing unit can be used and deallocated by another thread processing unit. The multiple free lists may become unbalanced after the program has run for a while. If any free list runs out of its free memory blocks, the heap memory manager will invoke a re-distribution routine to balance the number of free memory blocks among the thread processing units.

3.4 Using Critical Section for Order-independent Operations

Data dependences between threads are often caused by order-independent operations on a shared variable or data structure. Examples of order-independent operations include adding a value to a variable (reduction operations) and inserting a node to a list in which the order is irrelevant. Using target stores to enforce order-independent operations will serialize the concurrent execution of the multiple threads. To avoid such a problem, we can place order-independent operations in critical sections.

There are two requirements to ensure the correct execution of critical sections for order-independent operations. First, the results of the order-independent operations should bypass the memory buffer and be written directly into the data cache so that the other threads can get the updated data. Second, only non-speculative threads (threads which will not be aborted by its predecessor threads) can enter the critical section and perform order-independent operations out of order. This is to prevent a speculative thread from performing an order-independent operation and writing uncommitted results to the data cache.

3.5 Memory Buffering in the Main Memory

We use memory buffers to save target store addresses and data for run-time data dependence checking, and to buffer uncommitted store data generated during speculative execution. The memory buffer can also be used to store private variables and to eliminate anti- and output-dependences caused by private variables between threads. Due to the limitation of hardware resources, the memory buffer will have a fixed size. The compiler can estimate the usage of the memory buffer and partition threads accordingly to avoid memory buffer overflow. When a memory buffer overflows, the thread must be stalled until all of its predecessor threads are completed before it can resume execution.

To exploit parallelism from a large outer loop and to avoid memory buffer overflow, the compiler can buffer the uncommitted store data as well as the private store data in the main memory. For this purpose, a special store instruction is provided which can bypass the memory buffer in the thread processing unit and directly write the data to the data cache.

While a thread may use the cache or the main memory to buffer uncommitted and private store data, it still needs the memory buffer in the thread processing unit to store target store addresses and data for run-time data dependence checking. When a thread becomes the head thread and has completed its computation stage, it has to perform the write-back for the store data buffered in the main memory.

4 Performance Evaluation

4.1 Methodology

To study the performance of the superthreaded architecture and the effectiveness of the program transformation techniques, we implemented a simulator that models a superthreaded processor. We manually transform benchmark programs into their corresponding superthreaded programs at the source level by applying the program transformation techniques described in the previous section. In the transformed programs, special superthreading instructions, such as *fork* and *store_ts*, are represented as function calls to specific subroutines, so that they could be recognized by the simulator. The transformed programs are compiled by the SGI C compiler. The programs are then instrumented by *pixie* [7] to generate instruction and memory reference traces. The simulator executes the instrumented programs on the host SGI machine and collects the traces. During the trace collection phase, the function calls which represent the superthreading instructions are converted into the corresponding superthreading instructions for simulation.

The execution unit of a thread processing unit is organized as a traditional RISC processor which can execute one instruction per machine cycle in the program sequential order. Load and jump instructions will cause one cycle delay if the compiler cannot find an independent instruction to fill the delay slot. The execution of a thread can be stalled by the following situations:

- Waiting for the *tsag_done* flag or the *wb_done* flag from the predecessor thread.
- Waiting for a target store data from previous threads, or a lock held by another thread.
- The queue between the execution unit and the communication unit is full.
- The queue between the communication unit and the next communication unit is full.
- The memory buffer is full.

The interconnection between thread processing units is modeled as a unidirectional ring. Each communication unit can forward one command word or one target store entry per cycle to the down-stream communication unit. The memory buffer is fully-associative and can process one load operation and one store operation (including target stores from the predecessor threads) per cycle. The memory buffer uses a two-stage pipeline structure, so every read and write to the memory buffer will take two cycles, and a read or a write to data cache will also take two cycles.

4.2 Benchmarks

Table 1 lists the benchmark programs and the inputs used in our performance study. We use two GNU unix utilities (*wc* and *cmp*), two SPEC92 floating pointer programs (*052.alvinn* and *056.ear*) and five SPEC95 integer programs (*124.m88ksim*, *126.gcc*, *129.compress*, *130.li*, and *132.ijpeg*) for our performance study. All of these programs are written in C. Since the compiler development for the superthreaded processors still under way, we manually transformed the most time-consuming routines of those programs into the superthreaded codes at the source level for simulation. Table 1 shows the total dynamic instruction counts of the original programs, and the dynamic instruction counts of these most time-consuming routines before and after they are transformed into the superthreaded form. The increased dynamic instruction counts for these routines reflect the run-time overhead for thread pipelining as well as the overhead

Program	Input set	Insn. Count Total (original)	Insn. Count Transformed (before)	Insn. Count Transformed (after)	Percent Increase
wc	expr.c from gcc	3.98M	3.97M (99.7%)	6.15M	54.9%
cmp	expr.c from gcc	5.13M	5.12M (99.8%)	5.27M	10.9%
052.alvinn	ref(20 iterations)	613.6M	543.5M (88.6%)	559.4M	2.9%
056.ear	short	812.7M	802.0M (98.6%)	846.9M	5.6%
124.m88ksim	train	195.2M	148.0M (75.8%)	156.7M	5.9%
126.gcc	jump.i	218.0M	64.7M (29.8%)	64.8M	0.2%
129.compress	train	50.6M	46.4M (90.2%)	52.1M	12.3%
130.li	train	275.9M	135.3M (49.0%)	188.6M	39.4%
132.ijpeg	test	856.1M	598.1M (69.9%)	711.8M	19.0%

Table 1. Benchmark Programs and their dynamic instruction counts.

Transformations	wc	cmp	alvinn	ear	m88ksim	gcc	compress	li	ijpeg
Function inlining							√	√	
Induction variable substitution	√	√	√		√				√
Loop unrolling	√	√	√	√	√				
Loop interchanging			√						
Statement reordering to increase overlap	√				√	√	√	√	
Converting data speculation to control speculation						√	√	√	
Distributed heap memory management						√		√	
Critical section for order-irrelevant operation	√					√			
Memory buffering in the main memory						√			

Table 2. Program transformations used in manually transformed programs

caused by parallelism enhancement. These numbers would be lower if we perform the program transformation for thread pipelining at the instruction level.

To faithfully mimick the action of the superthreaded compiler, we limited ourselves to use only those compiler techniques described in Section 3 during manual transformation. Table 2 summarizes the program transformation techniques that we actually employed for the benchmarks.

Program	wc		cmp		alvinn		ear	
# of units	4	8	4	8	4	8	4	8
Routine(1)	1.95	2.71	3.34	6.15	3.74	7.48	3.02	4.53
Routine(2)					3.70	6.62	3.70	5.57
Routine(3)					3.69	6.62	3.54	5.31
Routine(4)					3.40	6.90	3.66	5.47
Routine(5)					1.47	1.47	3.32	4.98
Routine(6)					3.89	7.76	3.38	5.09
(1) - (6)	1.95	2.71	3.34	6.15	3.61	7.17	3.53	5.29

Program	m88ksim		gcc		compress		li		ijpeg	
# of units	4	8	4	8	4	8	4	8	4	8
Routine(1)	3.23	5.03	1.31	1.39	1.04	1.04	0.89	0.89	3.83	6.87
Routine(2)	1.25	1.28	2.05	3.27	1.26	1.26	1.50	1.49	3.48	5.59
Routine(3)	2.06	2.01					0.83	0.89	1.60	1.64
Routine(4)	1.15	1.15					0.83	0.84	3.73	5.24
Routine(5)	3.77	7.53							2.69	4.22
Routine(6)	3.88	3.88							2.35	4.21
Routine(7)									2.86	4.81
(1) - (7)	2.72	3.53	1.37	1.50	1.13	1.13	0.96	0.96	2.91	4.35

Table 3. Speedups of a superthreaded processor over a single-thread RISC processor

4.3 Results

Table 3 shows the speedups of a superthreaded processor, with 4 or 8 thread processing units, over a single-threaded, single-issue RISC processor for the most time-consuming routines in each of the benchmark programs. The superthreaded processor and the single-threaded RISC processor use the same hardware configurations in instruction execution. However, the RISC processor simulator executes the original benchmark programs, while the superthreaded processor simulator executes the transformed superthreaded codes, which have more run-time overhead than the original programs as shown in Table 1.

Wc and *cmp* spend most of their execution time in one loop. Both loops are very small and have some loop-carried data dependences. For these two loops, we performed loop unrolling and used target store instructions as well as critical sections to enforce data dependences. Even with the loop-carried data dependences, these two programs can achieve good speedups in the superthreaded execution model.

The two SPEC92 floating pointer programs *052.alvinn* and *056.ear* spend most of their execution time in several parallel loops. By executing iterations of these loops concurrently on multiple thread processing units, we can obtain very good speedups for these two programs. To increase the granularity of threads, we perform loop unrolling for some of these loops. We also perform loop interchanging for routine *hidden_input* in *052.alvinn* to avoid memory buffer overflow.

In *124.m88ksim*, the routines that have been transformed are Routine (1) *killtime*, Routine (2) *ckbrkpts*, Routine (3) *alignd*, Routine (4) *test_issue*, Routine (5) *loadmem*, and Routine (6) *rdwr*. Both *ckbrkpts* and *test_issue* have a small WHILE loop with exit conditions. These loops have a small number of iterations before they exit, and therefore their speedups are very limited. *Killtime*, *alignd*, *loadmem*, and *rdwr* have more speedups because their loops have more iterations. However, there are cross-iteration data dependences in the loops of *alignd*, which cause the speedup to be saturated when more than 4 thread processing units are used.

For *126.gcc*, we generated the superthreaded codes for the two most time-consuming passes: Routine (1) *cse* (common subexpression elimination) and Routine (2) *regclass* (register class). For *cse*, we exploit the thread-level parallelism for the outermost loop, which processes an extended basic block in each iteration. To exploit parallelism between iterations, data speculation is necessary because basic block boundaries could be modified during the process. We converted the data speculation to control speculation to exploit its potential parallelism. However, at the run-time about one-third of the basic blocks are modified and need to be re-processed. Hence, only a small speedup is obtained. In *regclass*, there is a major loop which scans all instructions and computes the cost for pseudo-registers. This loop only has a few data dependences between loop iterations. We enforce the data dependences by using target store instructions and critical sections. The speedup for this loop turns out to be reasonably good. One notable observation is that most of the innermost loops in *gcc* are inherently sequential due to the algorithm used in *gcc*. The speedup obtained from software pipelining on these innermost loops is known to be be very small.

In *129.compress*, most of the execution time is spent in two loops, one is in Routine (1) *compress* and the other one is in Routine (2) *decompress*. These two loops have very limited loop-level parallelism because each iteration may be data-dependent on the results computed at the very end of the previous iteration. Although we can speculate on the data dependences with control speculation, only limited speedups can be obtained from these loops.

For *130.li*, we transformed four most time-consuming routines: Routine (1) *mark*, Routine (2) *sweep*, Routine (3) *xlygetvalue*, and Routine (4) *xlxgetvalue*. *Mark* has a doubly-nested WHILE loop with complex data dependences between iterations and with recursive function calls inside the loop body. We tried to exploit the limited amount of parallelism from inner loops and partitioned their execution into threads by using control speculation and a lot of target store instructions. However, the simulation results show that the run-time overhead slows down the execution. *Sweep* has a doubly-nested loop that has few loop-carried data dependences. By applying distributed heap memory, we can achieve a speedup of 1.50 from this loop. Both *xlygetvalue* and *xlxgetvalue* have a doubly-nested loop which is very parallel. However, during the actual execution, these loops are executed mostly with only one iteration. This causes the speedups to

Program	wc		cmp		alvinn		ear		m88ksim		gcc		compress		li		ijpeg	
# of units	4	8	4	8	4	8	4	8	4	8	4	8	4	8	4	8	4	8
Routine(1)	3	3	5	5	3	3	9	9	6	6	1	1	5	5	6	8	25	25
Routine(2)					3	3	25	25	13	13	3	3	24	32	7	7	23	23
Routine(3)					63	63	17	17	11	20					1	1	9	9
Routine(4)					31	31	9	9	2	2					1	1	9	9
Routine(5)					2	2	18	18	15	15							5	5
Routine(6)					73	73	17	17	3	3							4	4
Routine(7)																	16	16

Table 4. Maximum number of memory buffer entries used in each routine.

be less than 1 due to the overhead in thread pipelining.

132.ijpeg is an image compression program that has a lot of parallel loops or loops with only a few loop-carried data dependences. By using the target store mechanism, we can successfully partition the execution of those loops into concurrent threads and achieve very good speedups.

Notice that in these manual program transformation and simulation studies, the program transformation is done at the source code level. We believe if it is performed in the backend at the instruction level, a lot of unnecessary overhead can be reduced.

Memory Buffer Usage The memory buffer of each thread processing unit could be very expensive, because it needs associative hardware for run-time data dependence checking. For this reason, the memory buffer cannot be too large. Table 4 shows the maximum number of memory buffer entries needed in each transformed routine. For *gcc*, speculative and private data is stored in the main memory. The numbers shown in Table 4 do not include such entries in the main memory. The results show that only a small number of memory buffer entries (less than 100 entries) are needed to buffer target stores and local stores.

5 Conclusions

The superthreaded architecture is a powerful concurrent multithreaded architecture with hardware support for run-time data dependence checking and control speculation. To effectively utilize the hardware support and to achieve high performance, the superthreaded architecture relies on the compiler to extract thread-level parallelism and generate efficient superthreaded codes. In this paper we studied the compiler techniques for the superthreaded architecture. Through our study, we found that a lot of existing program analysis and transformation techniques developed for multiprocessor architecture can be used to generate concurrent threads for the superthreaded architecture. We also identified and developed several important program transformation techniques to exploit more parallelism in programs and to reduce run-time overhead for data communication and dependence checking.

We evaluate the performance of the superthreaded architecture and the effectiveness of the program transformation techniques by using a trace-driven, cycle-by-cycle simulator. The simulation results show that a superthreaded processor can achieve promising speedups for most of the benchmark programs with the proposed program transformation techniques applied.

References

1. Ding-Kai Chen and Pen-Chung Yew. Statement reordering for doacross loops. In *Proceedings of International Conference on Parallel Processing*, volume Vol. II, August 1994.
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. Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The m-machine multicomputer. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 146–156, November 29–December 1, 1995.
4. 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.
5. Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio Nishimura, Yoshimori Nakase, and Teiji Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 136–145, May 19–21, 1992.
6. Zhiyuan Li. Array privatization for parallel execution of loops. In *Proceedings of the 6th ACM International Conference on Supercomputing*, pages 313–322, July 1992.
7. M. D. Smith. Tracing with pixie. Technical report, Stanford University, Stanford, California 94305, November 1991. Technical Report CSL-TR-91-497.
8. 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.
9. Jenn-Yuan Tsai and Pen-Chung Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques, PACT '96*, pages 35–46, October 20–23, 1996.
10. Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 22–24, 1995.
11. Michael J. Wolfe. Optimizing supercompilers for supercomputers. Technical report, University of Illinois, October 1982. Technical Report UIUCDCS-R-82-1105.