Compiler Techniques for the Superthreaded 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. Several useful compiler and program transformation techniques for the superthreaded architectures [8] are presented in this paper. The superthreaded architecture adopts a thread pipelining execution model to facilitate runtime data dependence checking between threads, and to maximize thread overlap to enhance concurrency. In this paper, we present some important program transformation techniques to facilitate concurrent execution among threads, and to manage critical system resources such as the memory buffers effectively. We evaluate the effectiveness of those program transformation techniques by applying them manually on several benchmark programs, and using a trace-driven, cycle-by-cycle superthreaded processor simulator. The simulation results show that a superthreaded processor can achieve promising speedup for most of the benchmark programs.

1 Introduction

Recently, a number of concurrent multithreaded execution models [5, 7, 9, 3, 2, 8] are proposed as an alternative to the current single-threaded superscalar execution model for future generations of microprocessors. Using multiple threads of control to fetch and execute instructions from different locations within a program simultaneously allows compilers and processors to exploit more parallelism in the program. The concurrent multithreaded architecture (CMA) is more tightly coupled than a multiprocessor-on-a-chip. Its multiple thread processing units can communicate and synchronize directly among themselves (e.g. through their register files) without going through memory system as in a multiprocessor. Hence, its communication and synchronization overhead is much smaller than that in a multiprocessor. It allows CMAs to support much finer grain parallelism for more general-purpose programs. Those programs are typically written in C, and usually have complicated control flow and implicit data dependences that

* This work is supported in part by the National Science Foundation under Grant No. MIP 9610379, CDA 9502979; 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.

cannot be analyzed at compile time. Most of the proposed multithreaded architectural models provide some hardware support for thread-level speculation [5, 7, 2, 8] and run-time data dependence checking between threads [7, 2, 8].

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 parallelizing compiler techniques, which are originally developed for multiprocessors, to extract loop-level parallelism to generate multiple-threaded code. For non-numerical programs which are mostly written in C or C++, existing parallelizing compilers lack effective parallelization techniques because of the complicated data structures and control flows in those programs. It often forces 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. New compiler techniques are thus needed to take advantages of such hardware support. Those compiler techniques are more aggressive than those in conventional parallelizing compilers.

In this paper, we study a concurrent multithreaded architecture, called the *Superthreaded architecture* [8]. It uses a thread pipelining execution model to enhance overlapping between threads. It also provides hardware and software mechanism to facilitate run-time data dependence checking and its enforcement between threads, and to support thread-level control speculation. Although the program transformation techniques described in this paper are primarily for the superthreaded architecture, most of them are also applicable to other concurrent multithreaded architectures.

2 An Overview of the Superthreaded Architectural Model

The superthread architecture addresses the challenges in parallelizing general-purpose applications by tightly integrating compiler technology with run-time hardware support for thread-level speculation and data dependence checking. In its general form, a superthreaded processor consists of a number of thread processing units. The thread processing units are connected to each other through a unidirectional ring, which allows a thread processing unit to send commands or to forward store addresses and data to its down-stream thread units. Similar to a conventional microprocessor, a superthreaded processor has an on-chip instruction cache and a data cache that are shared by all thread processing units. However, each thread processing unit has its own program counter, local register file, and execution unit to fetch and execute instructions from an independent instruction stream. 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 before the thread leaves its speculation mode.

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 be 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

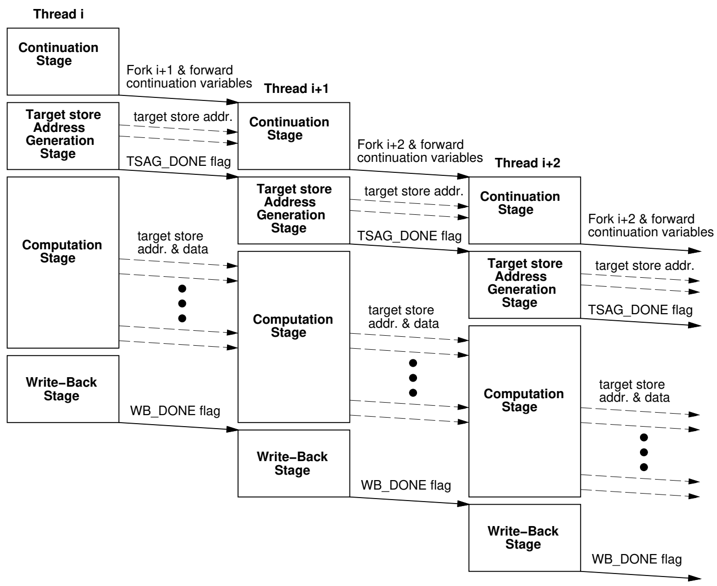

Fig. 1. The pipelined execution of contiguous threads

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 1, 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[2]. 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 2 is one of the most time-consuming loops in the SPECint95 benchmark *124.m88ksim*. This is a while loop with exit condi-

[2] A successor thread of the current thread is said to be concurrent if its distance to the current thread is less than the number of thread processing units, i.e., it could be concurrently executed on a down-stream thread processing unit before the current thread is retired

tions 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`.

```
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. 2. An example code segment from *124.m88ksim*

Figure 3 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 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

To fully utilize the hardware support for thread-level speculative execution and run-time data dependence checking, the superthreaded architecture relies on

```
/* 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. 3. The superthreaded code for the example shown in Figure 2.

the compiler to extract thread-level parallelism and to generate superthreaded codes. Given a sequential program, the compiler first partitions the execution of the program into threads for concurrent execution. The compiler then performs thread pipelining to facilitate run-time data dependence checking. 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.

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 present the program analysis and transformation techniques for program partitioning and thread pipelining, and then describe the advanced program transformation techniques to enhance the performance of superthreaded processors.

3.1 Basic Compiler Techniques for Superthreading

The compiler requires extensive data flow and dependence information to partitions a program at the appropriate level of granularity and to generate efficient superthreaded codes. The compiler can use the following traditional program analysis and transformation techniques to extract the desired information from programs, and to reduce the run-time overhead for data dependence checking.

- Function inlining: allow the compiler to perform data flow analysis and instruction movement across function boundaries for thread pipelining.
- Inductive variable substitution: eliminate loop-carried data dependences caused by inductive variables and reduce the number of required target stores.
- Alias analysis: provide alias information for analyzing data dependences caused by pointers.
- Data dependence analysis: identify loop-carried data dependences for thread-level parallelism estimation and target store generation.
- Variable privatization: eliminate data dependences caused by variables whose updated value do not reach a concurrent successor thread.

3.2 Program Partitioning

To achieve the best performance gain from superthreading, the compiler should partition a program at a level where sufficient thread-level parallelism is available. On the other hand, since each thread processing unit can also be superscalar, there should be sufficient instruction-level parallelism left in each thread. Therefore, the compiler must generate threads at the appropriate level where the maximum combined performance gains can be achieved from both superthreading and superscalar. In addition, the size of a thread cannot be too large in order to avoid overflowing the memory buffer [8].

To generate threads at the right level, the compiler examines data dependences between contiguous execution portions of a program and estimates the amount of thread-level parallelism at each level with respect to the data dependences. If the program has more than one level of loops that have a good amount of parallelism, the compiler will schedule the outer loop that would not overflow the memory buffer for thread-level execution, and leave the inner loop for each thread processing unit to exploit instruction-level parallelism. In case that only inner-most loops are suitable for thread-level execution, the compiler can perform loop unrolling or loop blocking to increase the size of each thread, and to provide each thread processing unit with sufficient workload to exploit instruction-level parallelism. Another technique to optimize program partitioning is loop interchange, which can be used either to interchange an outer parallel loop that will overflow the memory buffer with a inner sequential loop to allow the new inner loop to be executed in parallel, or to interchange a small inner parallel loop with an outer sequential loop to increase the size of each thread.

3.3 Thread Pipelining

After partitioning a program into threads, the compiler then further partitions each thread into pipelined stages to facilitate thread continuation and run-time data dependence checking. To facilitate thread pipelining, the compiler first analyzes data dependences between contiguous threads and marks store operations in each thread that could be data dependent by its concurrent successor threads as target stores. The compiler then moves instructions that are needed to compute the target store addresses, and their dependent instructions to the TSAG (target-store-address-generation) stage. For each target store address computed in the TSAG stage, the compiler places an *allocate_ts* instruction in the TSAG stage to forward the target store address to the concurrent successor threads. The target store addresses computed in the TSAG stage can be saved in registers and used in the following computation stage.

After the TSAG stage is formed, the remaining computation in a thread is its computation stage. The compiler replaces each last-store operation to a target store address with a *store_ts* instruction. In addition, if a control path taken at run-time will prevent some target store entries from being updated, the compiler needs to insert *release_ts* instructions in the control path to release these target store entries.

For threads that have many possible control paths, there may be many target store addresses that need to be generated at the TSAG stage, but only a few of them will actually be used in the computation stage. In this case, the overhead for the target store address generation might overshadow the performance gain from the parallel execution of multiple threads. To reduce the overhead in target store address generation, the compiler can move the condition tests to the TSAG stage and generate only target store addresses that will be used in the chosen control path. Like the target store addresses, the branch conditions precomputed at the TSAG stage can be saved in registers for later use in the computation stage.

The final step for thread pipelining is to extract the continuation stage from the TSAG stage. The continuation stage contains the computation that is essential to start the next thread. The data generated in the continuation stage will be forwarded directly to its successor threads before they are activated. Therefore, a thread can immediately load and use the data generated in the continuation stage of the previous thread without waiting for the previous thread to complete the TSAG stage. However, because the continuation stages of the contiguous threads are executed sequentially, the compiler must keep the continuation stage as small as possible. With these considerations, there are two criteria for instructions to be moved to the continuation stage. First, they only depend on the continuation stages of the predecessor threads. Second, they are needed in the TSAG stages of the current thread and the successor threads. In general, the continuation stage contains instructions to update the value of recurrent variables such as the loop index, which are usually needed for target store address computation.

After the continuation stage of a thread is extracted from its TSAG stage, there still may be instructions in the TSAG stage which are data dependent on its concurrent predecessor threads. In this case, the compiler needs to place a *wait_tsag_done* instructions before those instructions to delay their execution until the predecessor threads complete their TSAG stages and forward all of their target store addresses.

In the thread pipelining, the compiler will try to increase the execution overlap of concurrent threads by minimizing the stall caused by data dependences between threads. To do this, we can perform statement reordering [1] and schedule target stores as early as possible, or schedule load instructions that may be data dependent on the target stores of some predecessor threads as late as possible.

3.4 Advanced Compiler Techniques for Superthreading

Conversion of Data Speculation to Control Speculation Some concurrent multiple-threaded architectures, such as *Multiscalar* [4, 7] 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 that can benefit from this kind of data speculation, 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.

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.

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.

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 Simulation Results

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* [6] to generate instruction and memory reference traces.

The simulator executes the instrumented program on the host SGI machine and collects the traces generated by the program. During the trace collection phase, the function calls which represent the superthreading instructions will be converted to the actual superthreading instructions for simulation. The simulator can also recognize the starting of a new thread, which is marked by a special function call inserted in the transformed program. After detecting the starting mark of a new thread, the simulator will assign the instruction traces following the mark to the next thread processing unit. Using this mechanism, we can generate multiple-threaded instruction trace streams from a single-threaded instruction trace stream and simulate their execution on multiple thread processing units.

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 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 is still under way, we manually transformed the

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	147.1M (75.4%)	156.3M	6.2%
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

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

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)									2.35	4.21
Routine(7)									2.86	4.81
(1) - (7)	2.71	3.52	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

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.

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 (4) *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*, and Routine (5) *loadmem*. Both Routine (2) *ckbrkpts* and Routine (4) *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. Routine (1) *Killtime*, Routine (3) *alignd*, and Routine (5) *loadmem* have more speedups because their loops have more iterations. However, there are cross-iteration data dependences in the loops of Routine (3) *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 Routine (1) *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 Routine (2) *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*. Routine (1) *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. Routine (2) *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 Routine (3) *xlygetvalue* and Routine (4) *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 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 and the SGI back end optimizing compiler is used to exploit instruction level parallelism without any specific consideration for the superthreaded execution model. Some performance degradation could be resulted from such omission. If the program transformation is done in the back end at the instruction level, we can reduce the overhead for target store address generation by performing common subexpression elimination at instructions level and saving target store addresses in registers for later use in computation stages. In addition, we can perform computation reordering at instruction level to further increase computation overlap between thread.

5 Conclusions

The superthreaded architecture integrates compiler technology and run-time hardware support for cost-effective concurrent multithreading. This approach utilizes the program information available at compiler time to reduce run-time overhead and hardware complexity for thread creation and data dependence checking and speculation. On the other hand, the run-time hardware support allows the compiler to generate more aggressive concurrent threads for general-purpose applications. In this paper, we studied the compiler techniques for generating efficient superthreaded codes from sequential programs. In addition to traditional program analysis and transformation techniques developed for multiprocessor architectures, we 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 evaluated 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, pages 24–28, 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. M. D. Smith. Tracing with pixie. Technical report, Stanford University, Stanford, California 94305, November 1991. Technical Report CSL-TR-91-497.
7. 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.
8. 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.
9. 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.

This article was processed using the LaTeX macro package with LLNCS style