Performance Study of a Concurrent Multithreaded Processor *

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

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

[‡]Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

{*jtsai,zjiang,ness,yew*}*@cs.umn.edu*

Abstract

The performance of a concurrent multithreaded architectural model, called superthreading [15], is studied in this paper. It tries to integrate optimizing compilation techniques and run-time hardware support to exploit both thread-level and instruction-level parallelism, as opposed to exploiting only instruction-level parallelism in existing superscalars. The superthreaded architecture uses a thread pipelining execution model to enhance the overlapping between threads, and to facilitate data dependence enforcement between threads through compiler-directed, hardware-supported, thread-level control speculation and run-time data dependence checking. We also evaluate the performance of the superthreaded processor through a detailed trace-driven simulator. Our results show that the superthreaded execution model can obtain good performance by exploiting both thread-level and instruction-level parallelism in programs. We also study the design parameters of its main system components, such as the size of the memory buffer, the bandwidth requirement of the communication links between thread processing units, and the bandwidth requirement of the shared data cache.

1 Introduction

Recent concurrent multithreaded architectures (CMAs) such as the multiscalar [3, 11], the M-machine [2], the simultaneous multithreaded architecture [16], and others [4, 8, 13] have shown that exploiting thread-level parallelism is a viable approach to improve the scalability of existing single-threaded superscalar architectures. Using multiple threads of control to fetch and execute instructions from different locations of a programs simultaneously allows compilers and processors to exploit more parallelism from multiple instruction windows. Among these CMA models, some of them [16, 8] only support concurrent execution of loosely-coupled threads similar to a multiprocessor-on-a-chip, while others [4, 11, 2] allow more tightly-coupled threads to execute in parallel with hardware support for direct data transfer between threads. Some of them also provide thread-level control and data speculation to exploit high-level program structures such as DO-While loops and objects referenced through pointers.

In this paper, we study the performance of a CMA model with tightly-coupled threads, called superthreading [15]. It integrates compilation techniques and run-time hardware support to exploit both thread-level and instruction-level parallelism in programs. The superthreaded architecture uses a thread pipelining execution model to enhance the overlapping between threads. It also provides compiler-directed thread-level control and data speculation and run-time data dependence checking. Using compiler-directed thread-level control and data speculation allows it to have less hardware complexity than some prior CMA models [11, 13] for thread-level control and data speculation. However, it requires more sophisticated compiler technology to achieve good performance. Fortunately, its similarity to a multiprocessor-on-a-chip allows it to leverage many existing parallelizing compiler techniques.

In this paper, we briefly present the architectural and program execution model of the superthreaded architecture (refer to [15] for more details) in section 2. We then identify the required compilation techniques to support thread-level control and data speculation, and also the issues related to the exploitation of instruction-level parallelism within each thread in section 3. We applied those compiler techniques by hand to some SPEC benchmarks and measure their performance through a detailed trace-driven simulator. The results are shown and discussed in sections 4. We then

*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.

conclude the paper in section 5.

2 Superthreaded Architecture and Program Execution 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 second-level (L2) instruction cache and the data cache. Each thread processing unit has its own first-level (L1) instruction cache, program counter, register file, and execution unit.

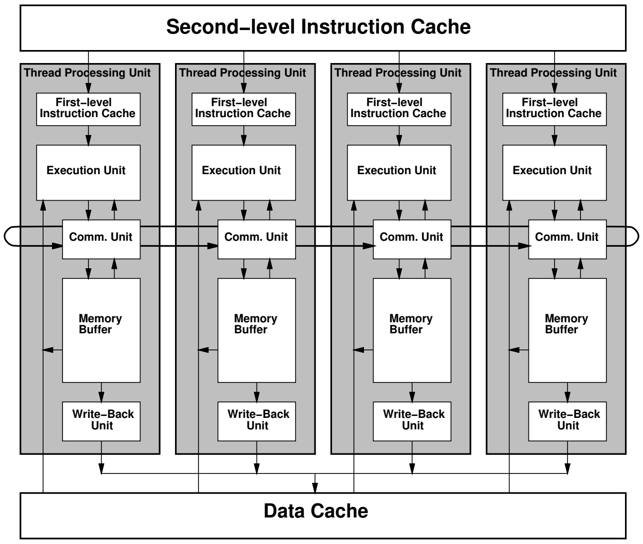

Figure 1. The organization of a superthreaded processor

2.1 Thread Processing Unit

The instruction fetch and execution portion of a thread processing unit is similar to a conventional superscalar to exploit instruction-level parallelism within each thread (see Figure 2). To provide sufficient instruction bandwidth, each thread processing unit has its own first-level (L1) instruction cache. The instruction fetch logic is also equipped with a branch target buffer [6] to eliminate branch delay and to support branch prediction and instruction-level speculative execution. Instructions fetched from the instruction cache are stored in the instruction queue before they are decoded and dispatched.

The instruction dispatch and completion unit decodes instructions from the instruction queue and dispatch them to the reservation stations of the appropriate functional units in order. Instructions can then be executed out of order when their operands are available. To support speculative execution and in-order instruction completion, the instruction dispatch and completion unit uses an reorder buffer [9] to buffer instruction results before they are committed. The reorder buffer also serves as an rename buffer to provide later instructions with uncommitted results which they are flow (read-after-write) dependent on.

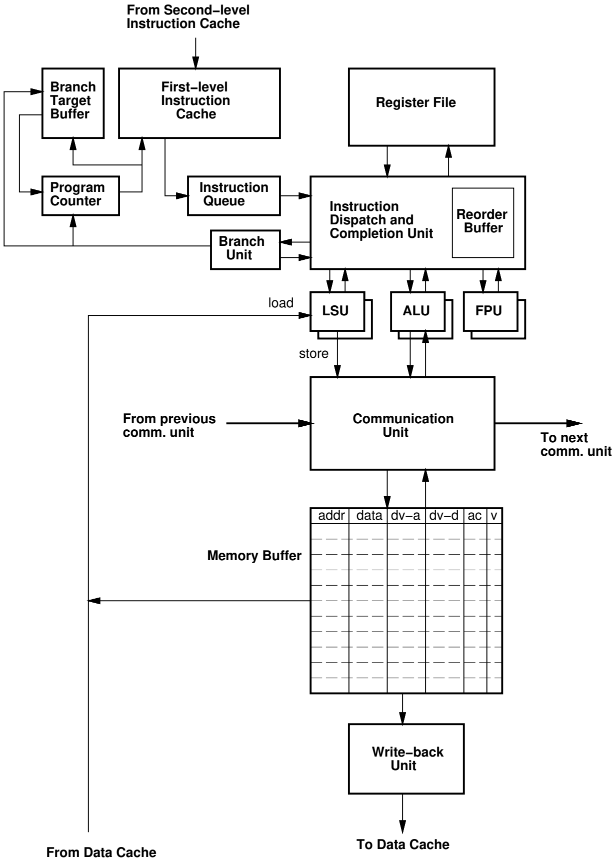

Figure 2. The block diagram of a thread processing element

Each thread processing unit also has a communication unit for transferring commands and data between thread processing units. A memory buffer is provided for run-time data dependence checking between threads. The memory buffer is also used to store speculative and private data from each thread for supporting compiler-directed, thread-level control and data speculation. For more details, please refer to [15].

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. This sequential thread execution order allows it to use a very simple and fast unidirectional ring to connects all thread processing units (without using a more complicated interconnection such as a crossbar switch).

2.2 Thread Pipelining Execution Model

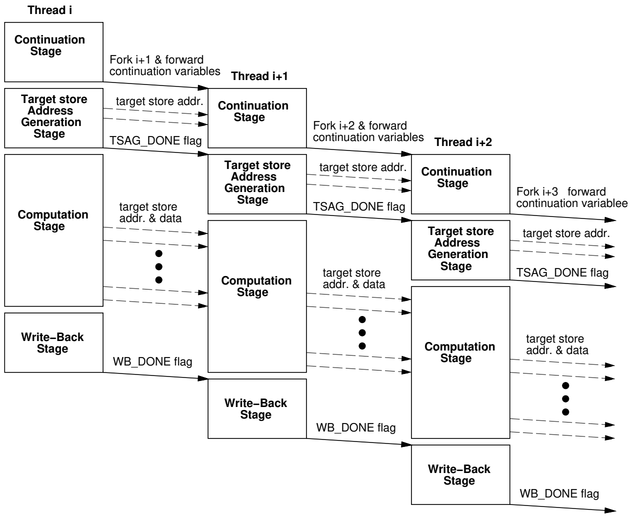

Figure 3. The pipelined execution of contiguous threads

The superthreaded architecture uses a thread pipelining execution model to enhance the overlapping between threads, and to facilitate run-time data dependence enforcement between 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 for forking a new thread on the next thread processing unit. Upon the execution of the fork instruction, all of the recurrence values 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 of the write operations upon which the concurrent successor threads could be data dependent. The addresses of those write operations are called *target store addresses*. They are identified by the compiler through a conventional data dependence analysis. These store addresses are computed at runtime, and are forwarded to the memory buffers of the successor threads for run-time dependence checking through an *allocate_ts* instruction. To guarantee the correctness, 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.

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 into its memory buffer by the predecessor thread using a *store_ts* instruction. 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 (including target stores) in the memory buffer will be committed and written to the cache memory. The write-back stages 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 can retire the thread and become idle until it is scheduled a new thread again.

2.3 An Example Superthreaded Program

The code segment shown in Figure 4.a 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.b 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*

```
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++;
}
```

(a)

```
/* 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 == 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:
```

(b)

Figure 4. (a) An example code segment from 124.m88ksim and (b) its superthreaded code

instruction. The original value of `i` is saved in `i_1` for later use. The continuation stage ends with a *fork* instruction to initiate a 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. 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 executing the *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 of 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 indefinitely 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 Superthreading

Since a superthreaded architecture exploits both thread-level and instruction-level parallelism, the compiler needs to leverage existing front end parallelizing compiler techniques for thread-level parallelism, and back end optimization techniques for instruction-level parallelism. In addition, the compiler must generate threads at the appropriate level to achieve the maximum performance. Since an extensive discussion on the compiler issues and techniques are beyond the scope of this paper, in the following subsections, we briefly address those two levels in some details.

3.1 Some Thread-Level Compilation Issues

Some conventional parallelizing compiler techniques such as function inlining, induction variable substitution, alias analysis, data dependence analysis, variable privatization, loop unrolling and interchange, etc. can also be used for the superthreaded architecture.

The compiler then partitions a program into threads, and each thread into multiple stages for thread pipelining [7, 14]. It also generates target store instructions for run-time data dependence checking. The compiler needs to increase the execution overlap of concurrent threads by minimizing the stalls caused by data dependences between threads. To do this, it needs to schedule target stores as early as possible, and schedule load instructions that may be data dependent on the target stores of some predecessor threads as late as possible [1].

The following superthreading-specific optimization techniques [14] are also found to be useful:

Conversion of Data Speculation to Control Speculation Hardware support for full data speculation can be very expensive, because it needs a buffer (called *Address Resolution Buffer* in multiscalar [3, 11]) to keep all *load* and *store* addresses from all active threads in order to detect data dependence violations. To avoid using such expensive hardware for data speculation, and to avoid wasting useful computation when a thread is squashed due to data dependence violation, the superthreaded architecture enforces data dependences between threads instead. 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 (see Section 4.4).

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

For threads 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 read-after-write data dependences that may occur only if the predecessor thread takes certain control paths that execute the writes, the compiler does not generate any target store addresses for the writes. Instead, the compiler inserts an *abort_future* instruction in the control paths that execute the writes. If the predecessor thread does take that control path, the *abort_future* instruction will be executed and the successor threads that may depend on the writes will be squashed and re-executed. Note that the hardware does not perform run-time data dependence checking for the control-speculated data dependences, even if they are undisambiguable at compiler time. The compiler just conservatively assumes that they are dependent and will be violated if the control paths that execute the writes are actually taken at runtime.

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 a heap memory manager, such as the *malloc* and *free* functions provided by the standard C library. Such heap memory management can cause potential data dependences between threads. To eliminate such data dependences, 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 become independent and can be executed in parallel. Note that a memory block can be allocated in one TPU and freed in another since the distributed heap memory managers still share the same memory space.

Using Critical Sections for Order-irrelevant Operations Data dependences between threads are often caused by order-irrelevant operations on a shared variable or data structure. Examples of order-irrelevant operations include adding a value to a variable (reduction operations) and inserting a node to a list in which the order is not important. Using target stores to enforce order-irrelevant operations will serialize the concurrent execution of the multiple threads. To avoid such a problem, we can place order-irrelevant operations on shared data in critical sections.

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.

3.2 Some Instruction-Level Compilation Considerations

Since each thread processing unit uses a superscalar execution model, the compiler must generate threads at the appropriate level, so there will be sufficient instruction-level parallelism left in each thread while exploiting thread-level parallelism. To provide each thread processing unit with sufficient workload to exploit instruction-level parallelism, the compiler would include as many instructions in a thread as possible. On the other hand, the size of a thread cannot be too large to avoid overflowing the memory buffer [15].

In addition, the compiler would partition a program at a level where the maximum thread-level parallelism is available. For some programs, however, the best thread level is at the inner-most loops which may have a small amount of instructions in each loop iteration. For such programs, the compiler has to trade off between the thread-level and the instruction-level parallelism in order to achieve the maximum combined performance gains.

In general, if a 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 with branch speculation or software pipelining [5]. In the 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 thread 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.

4 Performance Evaluation

4.1 Methodology

We evaluate the performance of the superthreaded architecture using a trace-driven simulator. We transform benchmark programs into their corresponding superthreaded programs at the source level. In the transformed superthreaded programs, special superthreading instructions, such as *fork* and *store_ts*, are represented as function calls to specific subroutines. The transformed superthreaded program is compiled by the SGI C compiler. The program is then instrumented by *Pixie* [10] to generate instruction and memory reference traces. The simulator executes the instrumented program on the host SGI machine and collects the traces. During the trace collection phase, the simulator converts function calls which represent the superthreading instructions into the corresponding superthreading instructions for simulation.

After the trace collection phase, the converted instruction traces are fed into the corresponding thread processing units in the simulator. The instruction fetch and execution portion of a thread processing unit is organized similar to a superscalar processor with a branch target buffer for branch prediction and a reorder buffer for out-of-order execution. Functional units are fully pipelined and have the same latencies as those of MIPS R10000.

The interconnection between thread processing units is modeled as a unidirectional ring. Each communication unit can forward a configurable number of commands or target store entries per cycle to the down-stream communication unit. The memory buffer is fully-associative and has a configurable number of ports. The memory buffer uses a two-stage pipeline structure, so every read and write to the memory buffer will take two cycles.

The simulator also includes data cache module, which is shared by all thread processing units. To provide sufficient cache access bandwidth, the cache is non-blocking and can be made of multiple-port, multiple-bank, or both. In the following performance evaluation results, the basic configuration for the data cache is given as follows: 64 kbytes, write back, 4-way set associative, 16-byte block, two read ports and one write port. A memory access hit in the cache takes two cycles, while the cache miss penalty is 12 cycles. The number of interleaved banks is scaled up with the total issue rate (number of thread units $\times$ issue rate per thread unit) of the processor.

4.2 Benchmarks

We use two GNU utilities (*wc* and *cmp*), two SPEC92 floating pointer programs (*alvinn* and *ear*) and four SPEC95 integer programs (*m88ksim*, *gcc*, *compress*, and *ijpeg*) for our performance study. All of these programs are written in C. Since the compiler development for the superthreaded processors requires both a front end parallelizing compiler and a back end optimizing compiler and is still under way [7, 14], we manually transformed the most time-consuming routines of those programs into the superthreaded codes at the source level. This approach allows us to evaluate and study various architectural issues, design parameters, and the feasibility of the superthreaded execution model before its compiler is fully functional. Even though in the manual transformation we try to mimick compiler algorithms as closely as possible, the transformed programs should still be interpreted as the results of a very good compiler with profiling information. On the other hand, the manual transformations are applied only 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. We thus restrict our performance comparison in varying architectural and design parameters, and not against the other CMA models.

Table 1 shows the input sets used for the simulation, the dynamic instruction counts of the original programs, and the dynamic instruction counts of those most time-consuming routines before and after they are transformed into the superthreaded form. Table 1 also shows the IPCs of the original programs run a single-threaded, single-issued processor model, which is used as our base model for computing speedup.

4.3 Performance Comparison

We run the benchmark programs on the simulator with the number of thread processing units ranging from 1 to 8, and the issue rate within each thread processing unit ranging from 1 to 8. Models with a

Program	Input set	Insn. Count Total (original)	Insn. Count Transformed (before)	Insn. Count Transformed (after)	Percent Increase	Base Model IPC
wc	expr.c from gcc	3.98M	3.97M (99.7%)	6.15M	54.9%	0.69
cmp	expr.c from gcc	5.13M	5.12M (99.8%)	5.27M	10.9%	0.78
052.alvinn	ref(20 iterations)	613.6M	543.5M (88.6%)	559.4M	2.9%	0.68
056.ear	short	812.7M	802.0M (98.6%)	846.9M	5.6%	0.89
124.m88ksim	train	195.2M	148.0M (75.8%)	156.7M	5.9%	0.71
126.gcc	jump.i	218.0M	64.7M (29.8%)	64.8M	0.2%	0.64
129.compress	train	50.6M	46.4M (90.2%)	52.1M	12.3%	0.75
132.ijpeg	test	856.1M	598.1M (69.9%)	711.8M	19.0%	0.81

Table 1. Benchmark Programs and their dynamic instruction counts.

# of TPUs × Issue rate	1 × 1	1 × 2	1 × 4	1 × 8	2 × 1	2 × 2	2 × 4	2 × 8	4 × 1	4 × 2	4 × 4	4 × 8	8 × 1	8 × 2	8 × 4	8 × 8	1 × 16	16 × 1
Reorder buffer size	8	16	32	64	8	16	32	64	8	16	32	64	8	16	32	64	128	8
Pending conditional branches	1	1	2	4	1	1	2	4	1	1	2	4	1	1	2	4	8	1
ALUs	1	2	4	8	1	2	4	8	1	2	4	8	1	2	4	8	16	1
FPUs	1	1	2	4	1	1	2	4	1	1	2	4	1	1	2	4	8	1
LSUs	1	1	2	4	1	1	2	4	1	1	2	4	1	1	2	4	8	1
Communication bandwidth	-	-	-	-	1	1	2	2	1	2	2	3	2	2	3	4	-	2
Write back bandwidth	1	1	1	2	1	1	2	4	1	2	4	8	2	4	8	16	4	4
Data cache banks	1	1	1	2	1	1	2	4	1	2	4	8	2	4	8	16	4	4

Table 2. Simulation configurations for thread processing units of different processor models

single-threaded processing unit are equivalent to a comparable superscalar processor. Note that we run the original benchmark programs on single-threaded models, and run the transformed superthreaded programs, which have more run-time overhead than the original programs, on multiple-threaded models.

In every model, each thread processing unit uses a 1024-entry, 4-way associative branch target buffer and a 96-entry fully-associative memory buffer. The detailed simulation configuration for each model is given in Table 2. In general, the reorder buffer (instruction window) size, the level of branch speculation over unresolved conditional branches, and the number of functional units are scaled up with the issue rate of each thread processing unit, while the communication bandwidth, write back bandwidth (for store data), and the number of interleaved cache banks are scaled up with the total issue rate of each combination.

To compare the relative performance (speedup) among different superthreaded configurations, and between superthreaded processors and superscalars, we use a single-threaded, single-issue processor as our baseline. This allows us to see the improvement due to superscalar approach, and the additional improvement obtained through superthreading. We do not use IPC for the performance comparison because multiple-thread models have relatively higher IPCs than single-thread models (due to the instruction overhead in the superthreaded programs). Figures 6 and 7 show the speedup of different configurations. For each benchmark program, we show both the speedup of the entire program and the speedup of the transformed portion of the program.

From the result, we can see that the superthreaded model can further improve the performance of a single-threaded (1 TPU) superscalar architectural model for most of the benchmark programs. For programs with high thread-level parallelism such as 052.alvinn and 056.ear, the combined speedup from superscalar and superthreading can exceed 10. On the other hand, programs with intensive loop-carried data dependences such as 129.compress cannot benefit much from the superthreaded execution model.

Overall, while the single-threaded model with 4-issue superscalar achieves an average speedup (entire program) of 2.34, the superthreaded model can further improve the average speedup to 4.04 (73% improvement) with 4 TPUs and to 4.6 (97% improvement) with 8 TPUs. For models with 8-issue per TPU, the superthreaded models can improve the average speedup from 3.18 to 4.97 (56% improvement) with 4 TPUs and to 5.5 (73% improvement) with 8 TPUs. If we consider only the transformed portion of the programs, the superthreaded models with 8 TPUs can improve the av-

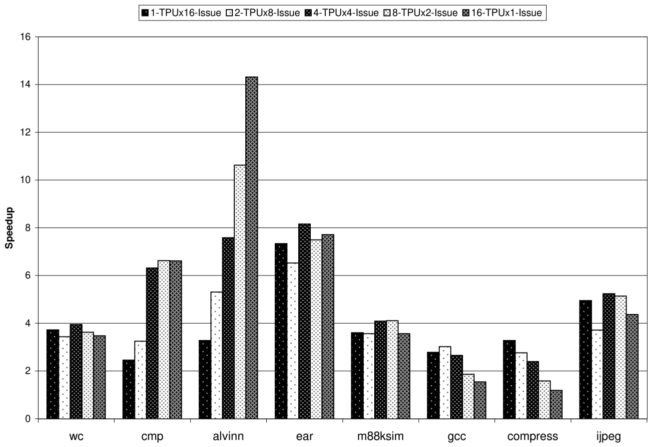

Figure 5. Performance comparison of models with a total issue rate of 16

erage speedup from 2.36 to 5.63 (139% improvement) for 4-issue per TPU and from 3.18 to 6.43 (102% improvement) for 8-issue per TPU.

Figure 5 compares speedup (transformed portion) of different models with a total issue rate of 16. We can see that models combining superthreading and superscalar (4-TPU×4-Issue or 8-TPU×2-Issue) can achieve better speedup than the pure superscalar model (1-TPU×16-Issue). In addition, the clock cycle time of the pure superscalar model could be longer than that of the combined models, because the 16-issue superscalar model uses a large (128 entries) instruction reorder buffer for dynamic instruction scheduling and supports higher level (8 branches) of branch prediction and speculation.

4.4 Memory Buffer Usage

The memory buffer of each thread processing unit could be very expensive because it needs to use associative hardware for run-time data dependence checking. For this reason, the memory buffer cannot be too large. Table 3 shows the maximum number of memory buffer entries used in each benchmark program executed in a superthreaded processor with different numbers of thread processing units. Note that the memory buffer may collect more target store entries from the predecessor threads as the number of thread processing units increases. For *gcc*, speculative and private data is stored in the main memory. The numbers shown in Table 3 do not include such entries in the main memory. The results show that using a small number of memory buffer entries (no more than 100) to buffer target stores and local stores can be sufficient and effective for most programs.

Program	Number of TPUs		
	2	4	8
wc	4	4	4
cmp	4	4	4
052.alvinn	72	73	73
056.ear	25	25	25
124.m88ksim	26	26	26
126.gcc	3	3	3
129.compress	20	24	32
132.ijpeg	26	26	26

Table 3. Maximum number of memory buffer entries used in each program

4.5 Communication Bandwidth Requirement

The communication units and the unidirectional ring allow a thread to receive commands and target stores from its predecessor threads, and to forward commands and target stores to its successor threads for thread initiation/synchronization and run-time data dependence checking. To keep the thread initiation and the run-time data dependence checking from becoming the critical path in the concurrent multithreaded execution, a communication unit may need to process and forward more than one request per cycle as the issue rate and the number of thread processing units increase. However, increasing the bandwidth of communication units could be very expensive. Therefore, it is very important to know the bandwidth requirement of the communication unit for the benchmark programs.

Program	Communication Bandwidth			
	1	2	3	4
wc	3.56	3.95	4.00	4.00
cmp	5.66	6.29	6.41	6.34
052.alvinn	5.90	6.01	6.04	6.06
056.ear	7.43	7.77	7.83	7.85
125.m88ksim	3.07	3.30	3.34	3.34
126.gcc	2.17	2.17	2.17	2.17
129.compress	2.32	2.39	2.39	2.39
132.ijpeg	3.39	3.54	3.56	3.57

Table 4. Speedup of a 4-TPU, 4-issue superthreaded processor with different communication bandwidths

In the communication bandwidth study, we use a superthreaded processor model consisting of four 4-issue thread processing units, which we think is a reasonable configuration for the next generation of micropro-

read ports	write ports	banks	wc contention % r	wc contention % w	wc speed-up	cmp contention % r	cmp contention % w	cmp speed-up	052.alvinn contention % r	052.alvinn contention % w	052.alvinn speed-up	056.ear contention % r	056.ear contention % w	056.ear speed-up
1	1	2	3.20	34.56	3.95	20.40	15.99	6.19	51.81	14.73	5.60	64.40	27.36	4.28
1	1	4	2.38	16.33	3.95	19.43	16.12	6.29	33.57	16.48	5.91	47.76	34.88	5.70
1	1	8	2.27	16.32	3.95	18.54	16.12	6.29	22.10	8.73	5.96	36.02	30.20	6.55
2	1	2	0.04	34.57	3.95	0.00	16.00	6.19	11.16	13.90	6.00	25.18	26.11	6.87
2	1	4	0.04	16.32	3.95	0.00	16.13	6.29	6.11	16.46	6.01	10.20	32.59	7.77
2	1	8	0.04	16.31	3.95	0.00	16.12	6.29	3.52	8.99	6.01	4.65	28.66	7.89
3	2	2	0.00	14.53	3.95	0.00	0.00	6.29	3.45	6.35	6.01	6.25	16.45	7.84
3	2	4	0.00	3.82	3.95	0.00	0.00	6.29	1.43	1.71	6.02	1.16	5.59	7.96
3	2	8	0.00	3.82	3.95	0.00	0.00	6.29	0.69	0.81	6.02	0.43	3.94	7.96
Cache hit ration %			99.95	99.98		99.72	99.98		92.25	99.88		99.99	99.98	

read ports	write ports	banks	124.m88ksim contention % r	124.m88ksim contention % w	124.m88ksim speed-up	126.gcc contention % r	126.gcc contention % w	126.gcc speed-up	129.compress contention % r	129.compress contention % w	129.compress speed-up	132.ijpeg contention % r	132.ijpeg contention % w	132.ijpeg speed-up
1	1	2	34.52	17.07	3.21	18.38	22.07	2.13	27.37	11.03	2.24	53.41	20.40	2.97
1	1	4	20.87	24.38	3.27	13.56	31.01	2.14	22.70	12.10	2.25	37.72	29.24	3.28
1	1	8	14.92	17.42	3.27	11.62	25.66	2.15	20.90	7.19	2.26	29.50	23.30	3.38
2	1	2	2.03	16.43	3.29	0.90	23.19	2.16	0.65	11.48	2.39	15.93	20.25	3.45
2	1	4	0.96	24.93	3.30	0.27	32.34	2.17	0.20	13.26	2.39	6.36	28.64	3.54
2	1	8	0.60	17.63	3.30	0.18	27.29	2.17	0.11	8.86	2.39	3.36	23.29	3.55
3	2	2	0.10	7.88	3.29	0.13	9.69	2.17	0.02	2.58	2.39	2.76	10.34	3.55
3	2	4	0.05	2.10	3.30	0.04	7.21	2.17	0.00	2.06	2.39	0.98	5.42	3.56
3	2	8	0.04	0.87	3.30	0.02	5.05	2.17	0.00	0.00	2.39	0.30	4.03	3.56
Cache hit ration %			99.61	98.24		99.41	99.37		99.11	98.52		99.81	99.50	

Table 5. Speedups and percentages of request blocked due to port contention

cessors, as our base model. We run the benchmark programs on the base processor model with different communication bandwidths ranging from 1 request per cycle to 4 requests per cycle. The simulation result are shown in Table 4. We see that for most of the programs, the speedup increases a noticeable amount as the communication bandwidth increases from 1 request per cycle to 2 requests per cycle. However, the speedup increase saturates beyond 3 and 4 requests per cycle. We conclude that a communication bandwidth of 2 requests per cycle would be sufficient for a 4-TPU, 4-issue superthreaded processor.

4.6 Data Cache Bandwidth Requirement

As the total issue rate increases in the superthreaded architectural models, it requires higher data cache bandwidth to support multiple loads and stores per cycle. Techniques to improve cache bandwidth include non-blocking, multi-port, and interleaved multi-bank [12]. Non-blocking improves the cache performance by reducing the bandwidth degradation due to misses. Multi-port (duplicated ports) and multi-bank caches can serve multiple requests per cycle. In general, a multi-port cache is more effective than a multi-bank cache because it has less port contention. However, it is much more expensive than a multi-bank cache.

To determine the cache bandwidth requirement for a superthreaded processor, we run the benchmark programs through the simulator configured with different combinations of multi-port and multi-bank. Again, we use a 4-TPU, 4-issue superthreaded processor as our base model. For multi-port cache, we use three different configurations: one read port/one write port, two read ports/one write port, and three read ports/two write ports. For multi-bank cache, the number of interleaved banks ranges from 2 to 8. Table 5 shows the simulation results. It shows the percentage of requests blocked due to port contention, which happens when more requests than the available ports are send to the same bank in the same cycle. The results also show the cache hit ratio for reads and writes. Note that the cache hit ratios would not be affected by the number of ports or banks.

We can see that most of the request contention ratio decreases as the number of ports or banks increases. For some programs, however, the contention ratio for the single write port increases as the number of banks increase from 2 to 4. This is because the write-back (from the memory buffer to the data cache) bandwidth will be limited to 2 writes per cycle when the cache module has only two banks and one write port in each bank. With fewer write-back requests per cycle, the contention ratio for the write port could be reduced, but eventually the total execution time may increase due to the slower write-back.

It is also very interesting to see that when the contention ratio is less than 30%, further reducing the contention ratio by adding banks or ports would not improve the speedup much. This is because when the contention ratio is not very high, a blocked request has a good chance to be served in the next cycle. As a result, delaying a few cache accesses for one or two cycles would not degrade the performance much. On the other hand, for programs with high instruction throughput and with a contention ratio higher than 50% such as 056.ear, increasing the number of cache ports and banks can improve their speedup significantly. Overall, a 4-way interleaved cache module with two read ports and one write would provide sufficient bandwidth for the benchmark programs studied.

5 Conclusions

Supporting concurrent execution of multiple threads on a single chip is a promising approach to boost the performance of existing superscalar microprocessors. The superthreaded architectural model takes advantage of optimizing compilation techniques and run-time hardware support to exploit both thread-level and instruction-level parallelism with compiler-directed thread-level control and data speculation and run-time dependence checking.

We evaluate the performance of the superthreaded architecture through a detailed trace-driven simulator. The simulation results show that the superthreaded architectural model can further improve the performance of a single-threaded superscalar processor.

We also study the size of the memory buffer and the bandwidth requirement for the communication unit and the shared data cache. Our results show that a memory buffer of fewer than 100 entries is sufficient for all of the benchmark programs studied. For a 4-TPU, 4-issue superthreaded processor, the required communication bandwidth between two connected thread processing units is about 2 requests per cycle for most of the programs. On the other hand, the required cache bandwidth varies from programs to programs. For programs with high instruction throughput, a 4-bank data cache with two read ports and one write port would be needed to provide sufficient cache bandwidth.

References

[1] D.-K. Chen and P.-C. Yew. Statement reordering for doacross loops. In *Proceedings of International Conference on Parallel Processing*, volume Vol. II, pages 24–28, August 1994.

[2] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee. The m-machine multicomputer. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 146–156, November 29–December 1, 1995.

[3] M. Franklin and G. 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] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. 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.

[5] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, June 22–24, 1988.

[6] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17(1):6–22, January 1984.

[7] Z. Li, J.-Y. Tsai, X. Wang, P.-C. Yew, and B. Zheng. Compiler techniques for concurrent multithreading with hardware speculation support. In *Proceedings of the 9th Workshop on Languages and Compilers for Parallel Computing, LNCS #1239*, pages 175–191, August 1996.

[8] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, October 1–5, 1996.

[9] J. E. Smith and A. R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 36–44, June 1985.

[10] M. D. Smith. Tracing with pixie. Technical report, Stanford University, Stanford, California 94305, November 1991. Technical Report CSL-TR-91-497.

[11] G. S. Sohi, S. 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] G. S. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, April 8–11, 1991.

[13] J. G. Steffan and T. C. Mowry. The potential for thread-level data speculation in tightly-coupled multiprocessors. Technical report, Computer Science Research Institute, University of Toronto, February 1997. Technical Report CSRI-TR-350.

[14] J.-Y. Tsai, Z. Jiang, and P.-C. Yew. Program optimization for concurrent multithreaded architecture. In *Proceedings of the 10th Workshop on Languages and Compilers for Parallel Computing*, August 1997.

[15] J.-Y. Tsai and P.-C. 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.

[16] D. M. Tullsen, S. J. Eggers, and H. 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.

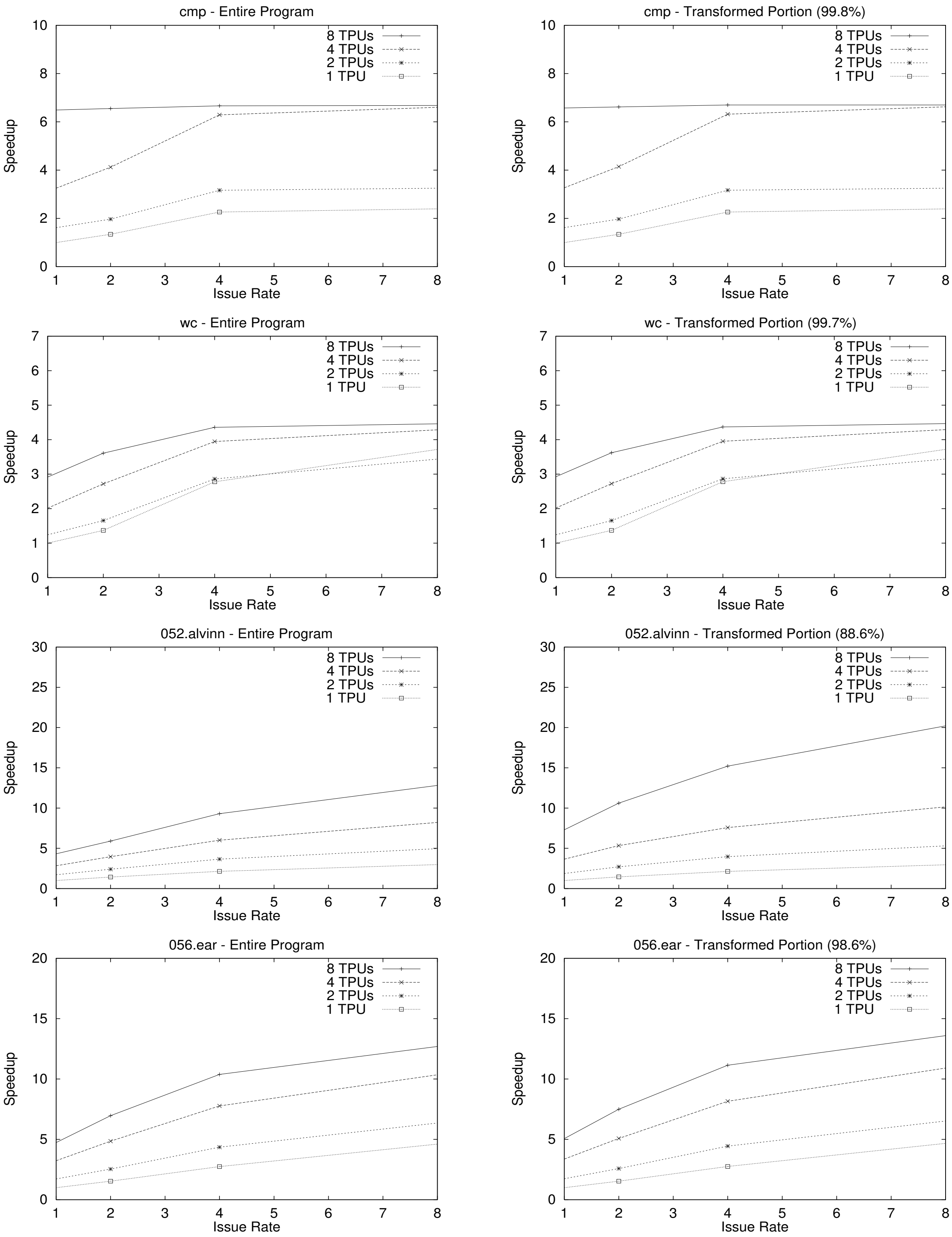

Figure 6. Performance comparison for wc, cmp, 052.alvinn and 056.ear

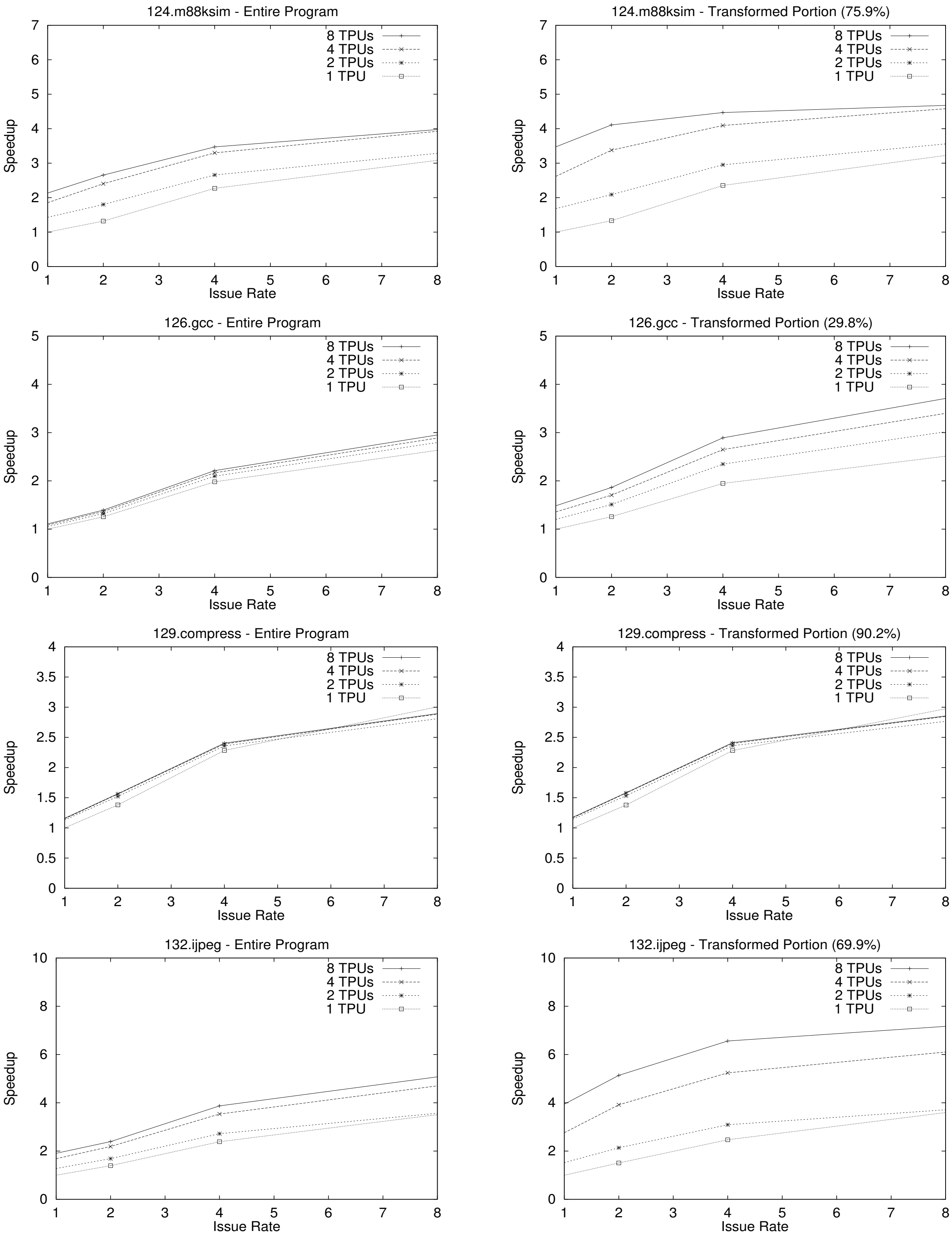

Figure 7. Performance comparison for 124.m88ksim, 126.gcc, 129.compress and 132.ijpeg