Parallelism exploitation in superscalar multiprocessing

N.-P. Lu
C.-P. Chung

Indexing terms: Parallel processing, Multiprocessing systems, Superscalar multiprocessing

Abstract: To exploit more parallelism in programs, superscalar multiprocessor systems, which exploit both fine-grained and coarse-grained parallelism, have been the trend in designing high-speed computing systems. Recently, the authors have developed a simulator for evaluating superscalar multiprocessor systems. This simulator models both a superscalar processor that can exploit instruction-level parallelism, and a shared-memory multiprocessor system that can exploit task-level parallelism. This simulator was used to run four applications chosen from the SPLASH-2 benchmark suite, and collected some performance data to investigate the parallelism exploitation capability of the superscalar multiprocessor systems in various configurations. It was observed that the instruction-level and task-level parallelism in programs can be exploited well by a moderate degree of superscalar processing and a high degree of multiprocessing. For example, the speedup of a 32-way multiprocessor with eight-issue processors can be over 200 relative to a single-issue uniprocessor.

1 Introduction

To exploit more parallelism in programs, superscalar multiprocessor systems have been the trend in designing high-speed computing systems. Examples of such systems include Cray SuperServer 6400 [1], Cray T3D System [2], Kendall Square Research KSR-1 [3], and Sun SparcCenter 2000 [4]. While superscalar processing exploits the instruction-level parallelism (ILP) within a processor, multiprocessing exploits task-level parallelism among processors. Superscalar multiprocessor systems provide enormous computing power by exploiting both fine-grained and coarse-grained parallelism in programs.

In designing a multiprocessor system, performance projection is an important task in making the multiprocessor architecture decisions. Three methods are commonly used to verify a multiprocessor system; prototyping, analytical modelling and simulation. Prototyping can predict most accurately the behaviour of the system, but it is in general the most time-consuming and costly. This is often done only at the last stage of system verification. Analytical modelling uses the simplified parameter set or probability distribution to model a system. However, due to the complexity of real systems, modelling is often too simple and naive to even approximate the actual system. In contrast, simulation can model the system at a variety of levels of detail, so that different aspects of the system can be studied in the desired detail. Simulation also allows the study of the behaviour of many design alternatives in a very short turn-around time and at a relatively low cost, reducing the development time and effort of the system.

Currently, there are many multiprocessor simulators available. Examples are Proteus [5], RPPT [6], Tango-Lite [7] and MINT [8]. However, these multiprocessor simulators model only the RISC processors with single instruction issuing, static scheduling and blocking loads. In contrast, current superscalar processors exploit high levels of instruction-level parallelism through techniques such as multiple instruction issue, dynamic scheduling, speculative execution and non-blocking memory access. Therefore, we believed that developing a simulator for performance evaluation of superscalar multiprocessor systems is absolutely necessary. After building such a superscalar multiprocessor simulator, we also used this simulator to run a number of benchmark programs to investigate the parallelism exploitation capability of the superscalar multiprocessor systems.

2 Superscalar multiprocessor simulator

We developed our superscalar multiprocessor simulator based on the MINT [8]. In this section, the MINT is first introduced and then the superscalar multiprocessor simulator, the SMINT (superscalar MINT), is presented in detail.

2.1 MINT. RISC multiprocessor simulator

Our superscalar multiprocessor simulator is based on MINT [8], a RISC multiprocessor simulator that supports the MIPS R3000 instruction set. MINT is a program-driven simulator as shown in Fig. 1. MINT controls the scheduling of processes so that the interleaving of memory references is the same as it would be on the simulated machine. In general, a program-driven simulator can be partitioned into two main

© IEE, 1998

IEE Proceedings online no. 19981955

Paper received 29th September 1997

The authors are with the Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan 30050, Republic of China

parts; a memory reference generator (also called the 'front end'), and a target system simulator (also called the 'back end'). The memory reference generator models the execution of an application program on some number of processors. When the program performs an interested operation, typically the generation of a memory reference, the front end sends an event to the back end. The back end models the system interconnect and the memory hierarchy. When the operations for an event complete, the back end signals the front end that some process can continue.

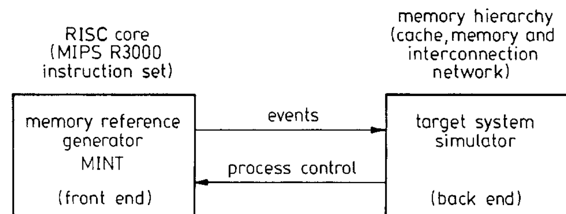

Fig. 1 *MINT simulator, a program driven simulator*

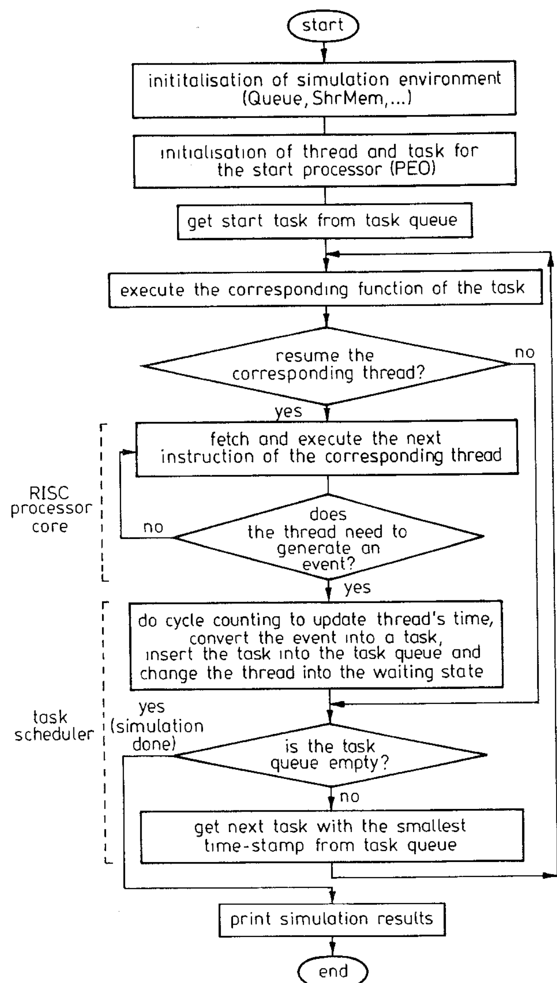

Fig. 2 *Simulation flow of MINT*

Fig. 2 shows the detailed execution flow of MINT. In the beginning, MINT initialises the simulation environment and allocates a thread, which represents the processor execution, and a task, which represents the back end simulation, for the start processor (PE0). After simulation initialisation, MINT executes the first task to wake up the first thread to execute processor instructions until an event is generated. Then, the generated event is converted into a corresponding task, and this task is inserted into the task queue to wait for scheduling. In the simulation, a task may create other tasks or fork threads to simulate multiprocessor execution. When a task completes, it may resume the execution of the corresponding thread or execute another task with the smallest time stamp. MINT runs until the task queue is empty, and an empty task queue means that the simulation is completed. Finally, MINT outputs the simulation results. Currently, MINT only supports MIPS R3000 RISC core. When linked to a proper back end that describes the memory hierarchy, MINT can simulate the multiprocessor system built with single-issue RISC processors.

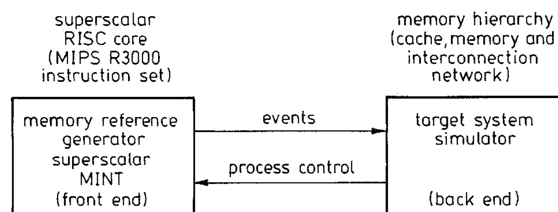

Fig. 3 *SMINT simulator*

2.2 SMINT. Superscalar multiprocessor simulator

To enable accurate simulation of multiprocessor systems using superscalar processors, we modified MINT to support superscalar processing. We call the modified simulator SMINT (superscalar MINT). SMINT uses the same MIPS R3000 instruction set as MINT and supports a variety of ILP features of contemporary superscalar microprocessors (Fig. 3). The processor core of SMINT has the following features:

(*a*) Superscalar execution; multiple instructions can be issued, executed and retired per cycle

(*b*) Dynamic instruction scheduling; instructions can be issued and executed out-of-order

(*c*) Register renaming; antidependencies and output dependencies can be eliminated

(*d*) Dynamic branch prediction and speculative execution; branch prediction enables the instruction scheduling window to grow beyond basic block boundaries.

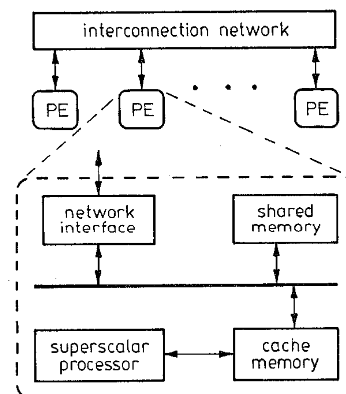

Fig. 4 *Superscalar multiprocessor system*

When linked to a proper back end that describes the memory hierarchy, SMINT can simulate the multiprocessor system built with superscalar processors. Fig. 4

shows an example of a superscalar multiprocessor system that SMINT can simulate. In this system, an interconnection network connects all the processing nodes. A processing node is composed of a superscalar processor, an instruction cache, a data cache, a shared memory module and a network interface.

Fig. 5 illustrates the processor microarchitecture that can be modelled by SMINT. The instruction control unit is the central controller of the superscalar processor. It handles instruction address generation, instruction fetching, interrupts, etc. In every cycle, it can fetch instructions from the instruction cache into the instruction window, decode the fetched instructions and dispatch the issuable instructions to the functional units for execution. Before dispatching instructions, the instruction control unit must detect data dependencies or resource conflicts among these instructions. In addition, the processor allows execution of instructions past unresolved conditional branches. A branch target buffer (BTB) with two-bit saturation counters is used to perform conditional branch prediction and support speculative execution.

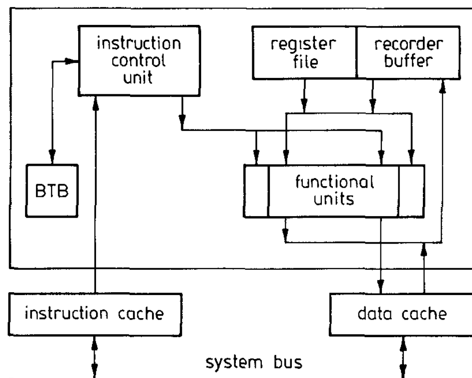

Fig. 5 *Superscalar processor modelled by SMINT*

To guarantee correct in-order execution results, the superscalar processor uses a reorder buffer to support precise interrupts and speculative execution. In addition to eliminating storage conflicts through register renaming, the reorder buffer is used to buffer speculated results and allow the processor to execute instructions past unresolved conditional branches. While the register file contains the in-order state data, the reorder buffer contains the look-ahead state data. If an exception occurs, the contents of the reorder buffer past the exception point are discarded, and the processor reverts to accessing the in-order state data in the register file after the exception handling. The processor then refetches and re-executes the correct instructions to generate correct in-order results. In the superscalar processor model, the execution time for all instructions is assumed to be one cycle. The processor with superscalar degree n is assumed to have n homogeneous function units, and it can fetch up to n instructions, execute up to n instructions and retire up to n instructions per cycle.

Fig. 6 shows the detailed simulation flow of SMINT. To support superscalar processing, the major modifications of MINT to SMINT include the superscalar processor core and the task scheduler for superscalar processing. The original MINT has about 23,000 lines of C codes; the modified SMINT adds more than 10,000 lines to the MINT code to implement superscalar processing. The simulation flow of SMINT is similar to the flow of MINT, except in the execution of threads. When the SMINT resumes a thread, the superscalar processor core will fetch, execute and retire instructions dynamically until the thread needs to generate an event. The event may be a look-ahead or an in-order memory access. As a result, the corresponding task can be look-ahead or in-order so that dynamic scheduling of processor executions and back end functions can be simulated. Currently, the superscalar processor core is well tested, and we are attempting to modify the task scheduler to support different memory consistency models and implement speculative memory access.

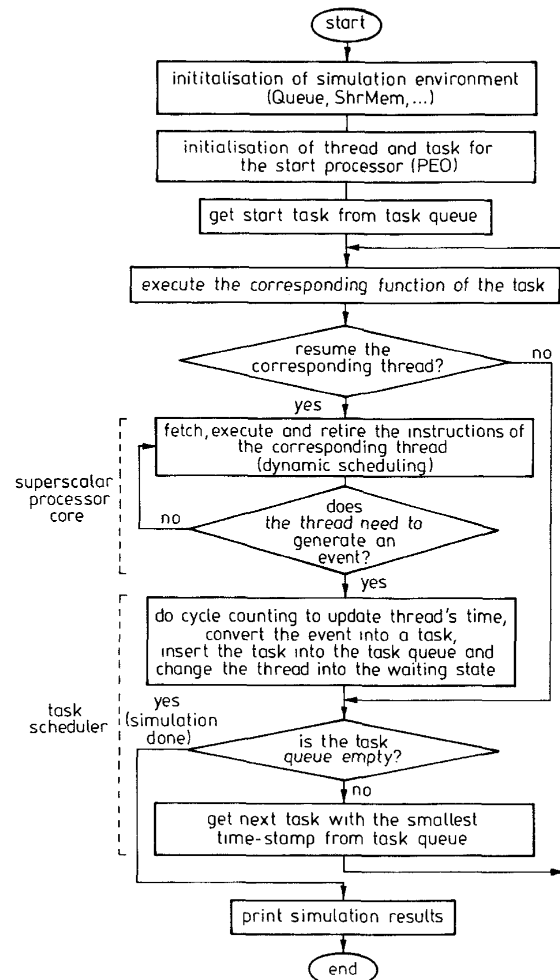

Fig. 6 *Simulation flow of SMINT*

3 Benchmark programs

In this Section, we describe the benchmark programs we used. The SPLASH-2 [9] benchmark program is widely accepted in studying centralised and distributed shared-address-space multiprocessors. The original SPLASH-2 programs are annotated by ANL macros [10] for multiprocessor execution. After expanding the macros into C codes by UNIX utility `m4` (the macro file is `c.m4.sgi`), the benchmark programs are compiled into MIPS object codes by `cc` of SGI IRIX System V.3. Then, the object codes are fed into our superscalar

multiprocessor simulator to produce simulation results. Fig. 7 outlines our simulation flow.

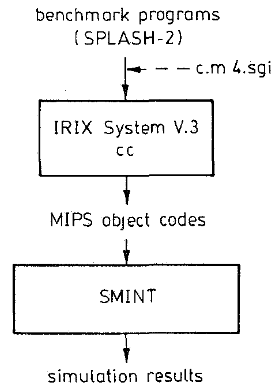

Fig. 7 *Execution flow of the simulation*

SPLASH-2 consists of a mixture of complete applications and computational kernels. It currently has eight complete applications and four kernels, which represent a variety of computations in scientific, engineering and graphics computing. In this research, we chose the following programs as our benchmarks:

3.1 FFT

The FFT kernel is a complex 1-D version of the radix-$\sqrt{n}$ six-step FFT algorithm described in [11], which is optimised to minimise interprocessor communication. The data set consists of the n complex data points to be transformed, and another n complex data points referred to as the roots of unity. Both sets of data are organised as $\sqrt{n} \times \sqrt{n}$ matrices partitioned so that every processor is assigned a contiguous set of rows, which are allocated in its local memory. Communication occurs in three matrix transpose steps, which requires all-to-all interprocessor communication. Every processor transposes a contiguous submatrix of $(\sqrt{n}/p) \times (\sqrt{n}/p)$ from every other processor and transposes one submatrix locally. The transpositions are blocked to exploit cache line reuse. To avoid memory hotspotting, submatrices are communicated in a staggered fashion, with processor i transposing first a submatrix from processor $i + 1$, then one from processor $i + 2$, etc.

3.2 LU

The LU kernel factors a dense matrix into the product of a lower triangular and an upper triangular matrices. The dense $n \times n$ matrix A is divided into an $N \times N$ array of $B \times B$ blocks ($n = NB$) to exploit temporal locality on submatrix elements. To reduce communication, block ownership is assigned using a 2-D scatter decomposition, with blocks being updated by the processors that own them. The block size B should be large enough to keep the cache miss rate low, and small enough to maintain good load balance. Fairly small block sizes ($B = 8$ or $B = 16$) strike a good balance in practice. Elements within a block are allocated contiguously to improve spatial locality, and blocks are allocated local to processors that own them.

3.3 Ocean

The Ocean application studies large-scale ocean movements based on eddy and boundary currents and is an improved version of the Ocean program in SPLASH [12]. The major differences are:

(i) it partitions the grids into square-like subgrids rather than groups of columns to improve the communication-to-computation ratio

(ii) grids are conceptually represented as 4-D arrays, with all subgrids allocated contiguously and locally in the nodes that own them

(iii) it uses a red–black Gauss–Seidel multigrid equation solver [13], rather than an SOR solver.

3.4 Radix

The integer radix sort kernel is based on the method described in [14]. The algorithm is iterative, performing one iteration for each radix r digit of the keys. In each iteration, a processor passes over its assigned keys and generates a local histogram. The local histograms are then accumulated into a global histogram. Finally, each processor uses the global histogram to permute its keys into a new array for the next iteration. This permutation step requires all-to-all communication. The permutation is inherently sender-determined, so keys are communicated through writes rather than reads.

In summary, Table 1 lists the code sizes of the benchmarks, and Table 2 provides the input problem sizes of the benchmarks that we used.

Table 2: Input problem sizes of benchmarks

Benchmark	Problem size
FFT	64K points
LU	256 × 256 matrix, 16 × 16 blocks
Ocean	130 × 130 ocean
Radix	256K integers, radix 1024

4 Simulation results

In this Section, we present the simulation results of parallelism exploitation collected by SMINT. To avoid the performance impact caused by the memory system, we assumed that the memory system is perfect (PRAM model [15]), so that all memory references complete in a single cycle. To exploit parallelism at different levels, we simulated the systems in (*a*) multiprocessing (*b*) superscalar processing (*c*) superscalar multiprocessing configurations as follows:

Table 1: Code sizes of SPLASH-2 benchmarks (in instruction words)

Benchmark	Integer	Floating	Load	Store	Branch	Jump	Syscall & sync	Total
FFT	6146 (51.3%)	272 (2.3%)	2075 (17.3%)	1319 (11.0%)	1409 (11.8%)	726 (6.1%)	35 (0.3%)	11982 (100%)
LU	6282 (51.9%)	194 (1.6%)	2053 (17.0%)	1295 (10.7%)	1503 (12.4%)	691 (5.7%)	76 (0.6%)	12094 (100%)
Ocean	16922 (44.7%)	1819 (4.8%)	10431 (27.6%)	3891 (10.3%)	3427 (9.1%)	1247 (3.3%)	105 (0.3%)	37842 (100%)
Radix	5621 (52.9%)	148 (1.4%)	1750 (16.5%)	1090 (10.3%)	1334 (12.5%)	664 (6.2%)	27 (0.3%)	10638 (100%)

4.1 Multiprocessing

Fig. 8 shows the speedup of multiprocessor systems with single-issue RISC processors. (Only the parallelisable portions of the benchmarks are measured.) From this Figure, one can see that all the benchmarks, except LU, can achieve a near linear speedup. When 32 processors are used, the speedup of FFT, Ocean and Radix are 29.67, 25.12 and 26.68, respectively. To explain why LU fails to achieve a near-linear speedup, Figs. 9–12 show the dynamic instruction mixes of the benchmarks. As the number of processors grows, LU spends much time in blocking due to synchronisation. Therefore, LU fails to achieve a near-linear speedup. For all the other benchmarks, the blocking time also restrains them from having a perfect linear speedup. Among these benchmarks, FFT obtains the highest speedup due to the least blocking time.

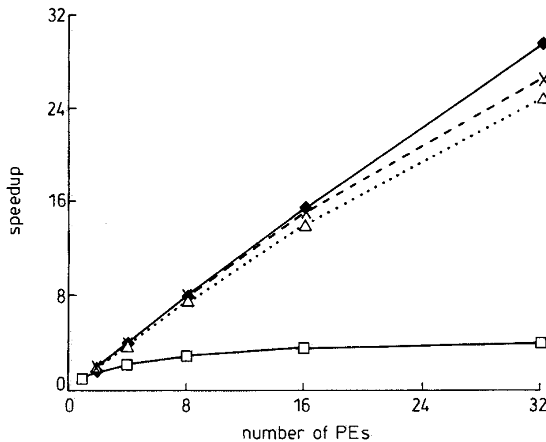

Fig. 8 *Speedup due to multiprocessing*
◆ FFT
□ LU
△ Ocean
× Radix

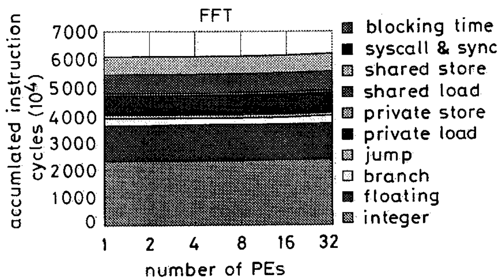

Fig. 9 *Dynamic instruction mixes for FFT*

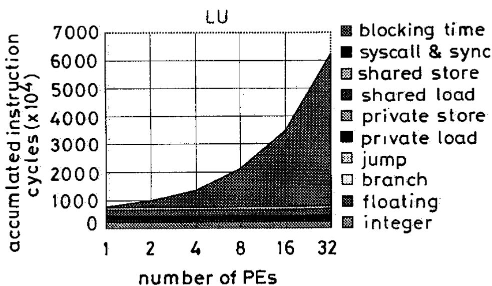

Fig. 10 *Dynamic instruction mixes for LU*

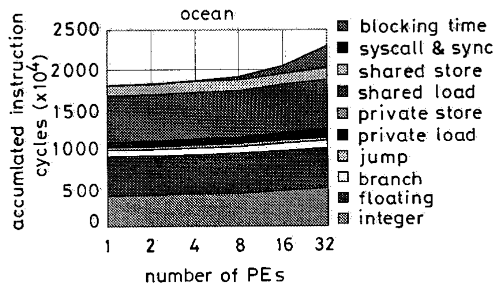

Fig. 11 *Dynamic instruction mixes for Ocean*

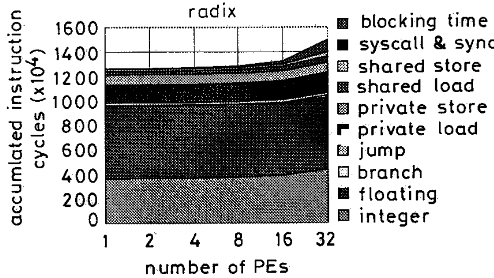

Fig. 12 *Dynamic instruction mixes for Radix*

4.2 Superscalar processing

4.2.1 Instruction-level parallelism:

To exploit the ultimate instruction-level parallelism of the benchmarks, we assume an ideal superscalar processor that has perfect branch prediction and infinite instruction window size. Fig. 13 shows the achievable IPCs (instructions per cycle) of the benchmarks on the assumed superscalar processor with varied superscalar processing degrees. It is observed that the sustained IPCs of the benchmarks range from 4.29 (Radix), 6.25 (FFT) and 10.48 (Ocean) to 12.08 (LU), and further gain in IPC is little when the superscalar processing degree is greater than 16. In the following, we study the impact of branch prediction accuracy and limited instruction window size on the instruction-level parallelism.

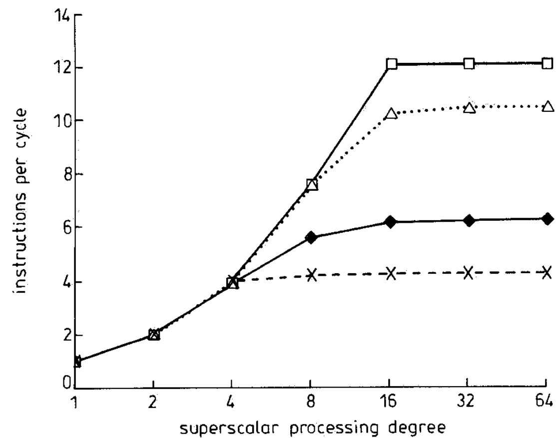

Fig. 13 *Instruction-level parallelism of four benchmarks on a superscalar processor with perfect branch prediction and infinite instruction window size*
◆ FFT; □ LU; △ Ocean; × Radix

4.2.2 Branch prediction affecting instruction-level parallelism:

SMINT models BTB with 2-bit saturation counters. Figs. 14–21 show the performance

of the various BTB designs. It is observed that 256 BTB entries with two-way set associativity are sufficient to reduce the BTB miss ratio to zero and achieve the maximum prediction accuracy for all the benchmarks. The maximum prediction accuracies are 90.4%, 90.9%, 96.5% and 99.9% for FFT, LU, Ocean and Radix, respectively. (Thus, a two-way set associative BTB with 256 entries is assumed in the following simulations.) Fig. 22 shows the impact of branch prediction on the instruction-level parallelism. The sustained IPCs of FFT, LU and Ocean drop to 5.96, 10.74 and 9.63 due to imperfect branch prediction, respectively. However, Radix retains the same IPC (4.29) due to its 99.9% prediction accuracy.

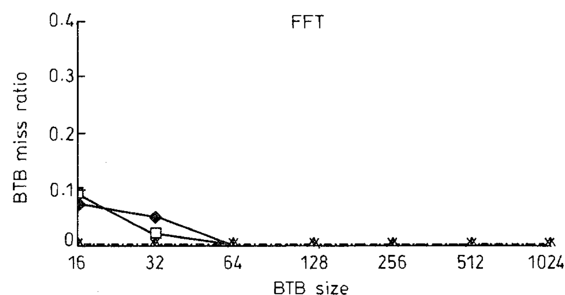

Fig. 14 *BTB miss ratio against BTB size and organisation for FFT*
◆ one-way; □ two-way; △ four-way; × eight-way

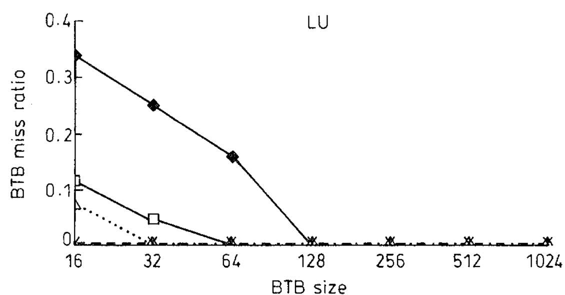

Fig. 15 *BTB miss ratio against BTB size and organisation for LU*
◆ one-way; □ two-way; △ four-way; × eight-way

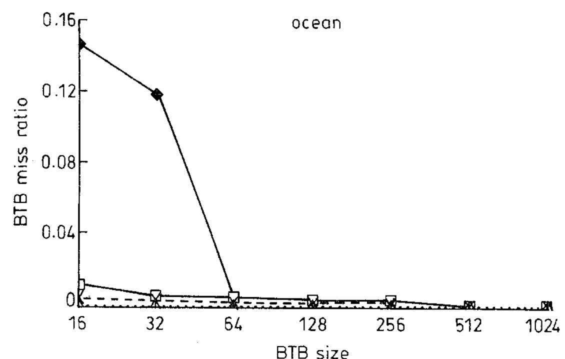

Fig. 16 *BTB miss ratio against BTB size and organisation for Ocean*
◆ one-way; □ two-way; △ four-way; × eight-way

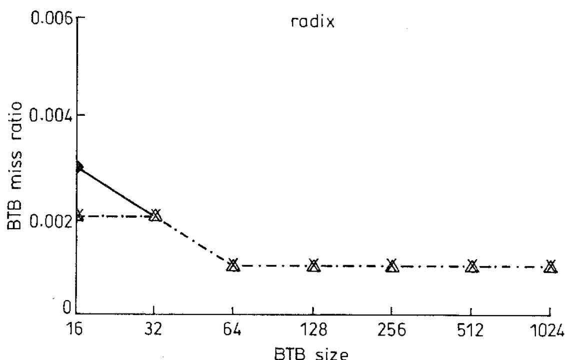

Fig. 17 *BTB miss ratio against BTB size and organisation for Radix*
◆ one-way; □ two-way; △ four-way; × eight-way

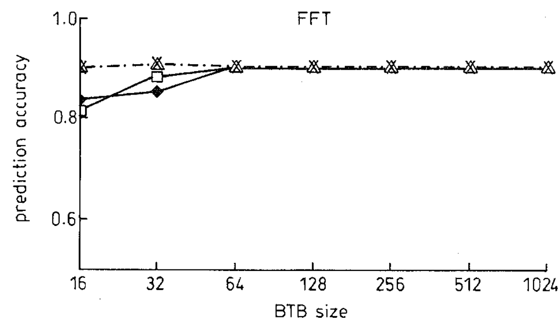

Fig. 18 *Prediction accuracy against BTB size and organisation for FFT*
◆ one-way; □ two-way; △ four-way; × eight-way

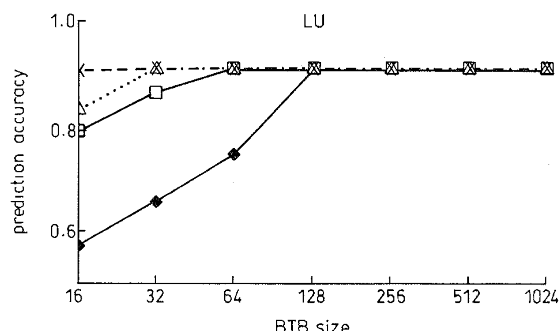

Fig. 19 *Prediction accuracy against BTB size and organisation for LU*
◆ one-way; □ two-way; △ four-way; × eight-way

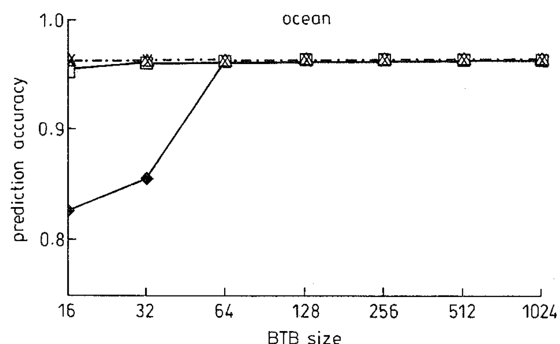

Fig. 20 *Prediction accuracy against BTB size and organisation for Ocean*
◆ one-way; □ two-way; △ four-way; × eight-way

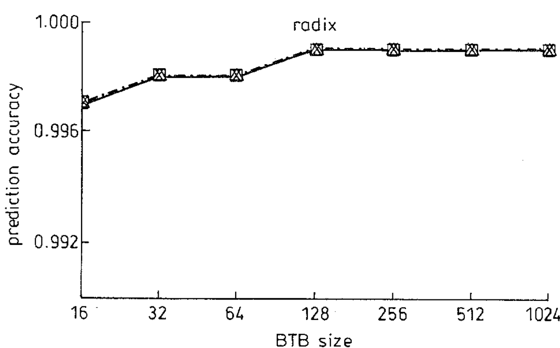

Fig. 21 *Prediction accuracy against BTB size and organisation for Radix*
◆ one-way; □ two-way; △ four-way; × eight-way

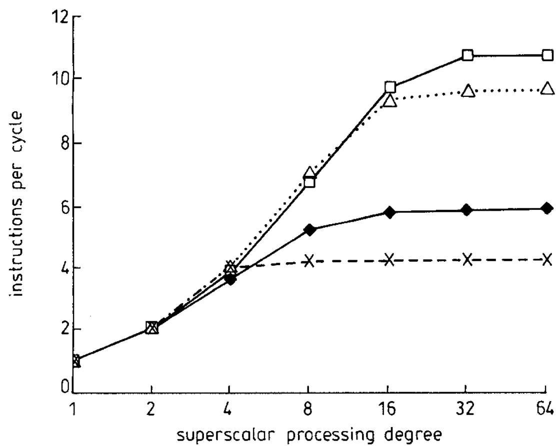

Fig. 22 *Instruction-level parallelism of the four benchmarks on a superscalar processor with 256-entry, two-way set associative and two-bit saturation counter BTB prediction*
◆ FFT; □ LU; △ Ocean; × Radix

4.2.3 Instruction window size affecting instruction-level parallelism: Fig. 23 shows the effect of instruction window size on the exploitable instruction-level parallelism. In general, the larger the instruction window, the more the exploitable instruction-level parallelism. However, the instruction window is very costly, and its circuit complexity grows tremendously as its size increases. Furthermore, the increased exploitable instruction-level parallelism is very insignificant after the instruction window size grows above a certain threshold. From Fig. 23, it is observed that an instruction window size of 128 is sufficient to exploit the most instruction-level parallelism. For this reason, in the following simulations we assume an instruction window size of 128.

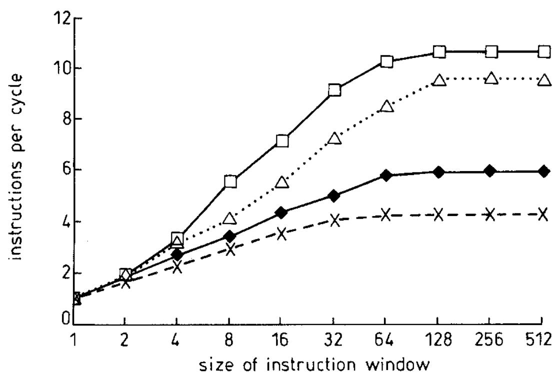

Fig. 23 *Effect of instruction window size on instruction-level parallelism*
◆ FFT; □ LU; △ Ocean; × Radix

4.3 Superscalar multiprocessing

In this Subsection, we present the simulation results about parallelism exploitation in the superscalar multiprocessor systems. Based on the simulation results of Section 4.2, we define the superscalar processor for constructing superscalar multiprocessing systems as follows: It is an n-issue superscalar processor with n homogeneous functional units, a two-way set associative BTB of 256 entries and a 128-entry instruction window.

Assume that the execution time of the sequential portions of all benchmarks are excluded in the following speedup calculations. Let m be the multiprocessing degree and n be the superscalar processing degree. We define the multiprocessing speedup as

$$Speedup_{multiprocessing}(m,n) = \frac{\text{execution time of } n\text{-issue uniprocessor}}{\text{execution time of } m\text{-way multiprocessor with } n\text{-issue processors}} \quad (1)$$

$$Speedup_{superscalar\ processing}(m,n) = \frac{\text{execution time of } m\text{-way multiprocessor with one-issue processors}}{\text{execution time of } m\text{-way multiprocessor with } n\text{-issue processors}} \quad (2)$$

$$Speedup_{overall}(m,n) = \frac{\text{execution time of one-issue uniprocessor}}{\text{execution time of } m\text{-way multiprocessor with } n\text{-issue processors}} \quad (3)$$

The idealised multiprocessing, superscalar processing and overall speedups can be m, n and $m * n$, respectively. However, these ideal speedups are hardly achievable due to synchronisation blocking, load imbalance in multiprocessing, limited instruction-level parallelism, branch misprediction, etc.

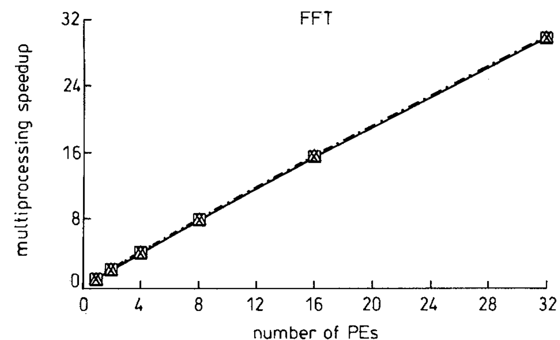

Fig. 24 *Multiprocessing speedups of FFT in superscalar multiprocessing*
◆ n = 1; □ n = 2; △ n = 4; × n = 8; –×– n = 16; –·×·– n = 32

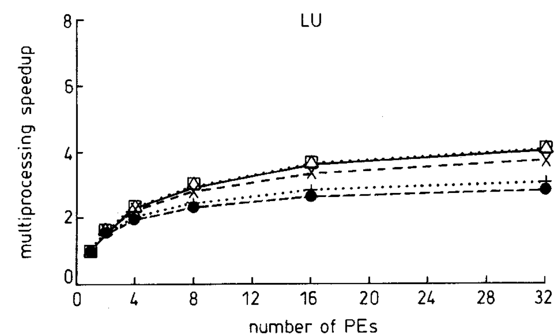

Fig. 25 *Multiprocessing speedups of LU in superscalar multiprocessing*
◆ n = 1; □ n = 2; △ n = 4; × n = 8; + n = 16; ● n = 32

Figs. 24–27 show the multiprocessing speedups of the benchmarks. For the FFT, the multiprocessing speedup remains an invariant as the superscalar processing degree increases. This is because FFT spends little time in being blocked (as shown in Figs. 9–12), so that as the processor element becomes more capable of exploiting instruction-level parallelism, the multiprocessing speedup can still be maintained. However, the multiprocessing speedups of LU and Ocean are lowered as

the processor element becomes more capable of exploiting instruction-level parallelism, indicating that the systems with such processor elements will spend more time in blocking relatively. As for the Radix, the multiprocessing speedup increases slightly as the superscalar processing degree increases. This phenomenon can be explained as follows. The Radix has less instruction-level parallelism (see Fig. 13) and near 100% branch prediction accuracy (see Fig. 21), so that the instruction window is often full in a high-degree superscalar processor. Therefore, as the number of processors increases, the more instruction windows in the processors will be able to exploit more instruction-level parallelism, so that the multiprocessing speedup increases.

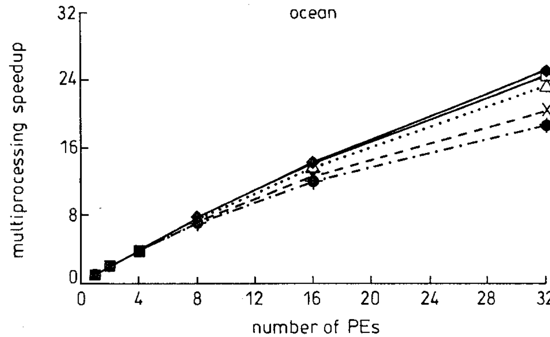

Fig.26 *Multiprocessing speedups of Ocean in superscalar multiprocessing*
◆ $n = 1$; □ $n = 2$; △ $n = 4$; × $n = 8$; + $n = 16$; ● $n = 32$

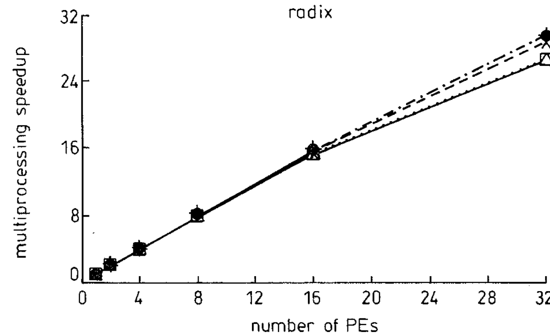

Fig.27 *Multiprocessing speedups of Radix in superscalar multiprocessing*
◆ $n = 1$; □ $n = 2$; △ $n = 4$; × $n = 8$; + $n = 16$; ● $n = 32$

Figs. 28–31 show the superscalar processing speedup of the benchmarks. The superscalar processing speedup of FFT remains an invariant as the multiprocessing degree varies. This result shows that in FFT the exploitation of instruction-level parallelism is independent of the exploitation of task-level parallelism. However, the superscalar processing speedups of LU and Ocean degrade as the number of processor elements increases. This is because blocking time constrains not only task-level parallelism but also instruction-level parallelism. As for the Radix, the superscalar processing speedup of the system with high-issue processors ($n \geq 8$) increases slightly as the multiprocessing degree increases. The reason for this phenomenon is the same as that stated in the last paragraph. The greater the number of instruction windows, the greater the exploitable instruction-level parallelism.

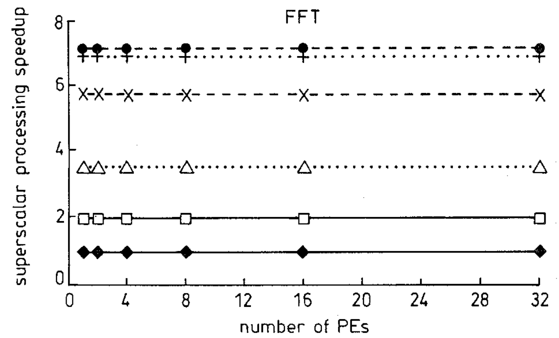

Fig.28 *Superscalar processing speedups of FFT in superscalar multiprocessing*
◆ $n = 1$; □ $n = 2$; △ $n = 4$; × $n = 8$; + $n = 16$; ● $n = 32$

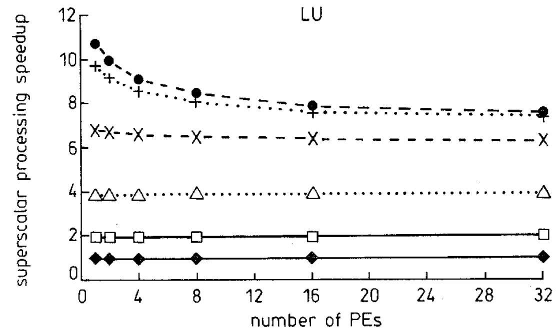

Fig.29 *Superscalar processing speedups of LU in superscalar multiprocessing*
◆ $n = 1$; □ $n = 2$; △ $n = 4$; × $n = 8$; + $n = 16$; ● $n = 32$

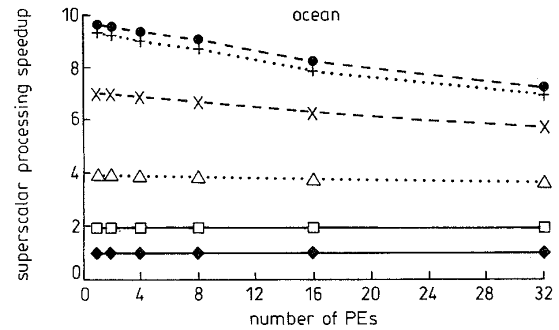

Fig.30 *Superscalar processing speedups of Ocean in superscalar multiprocessing*
◆ $n = 1$; □ $n = 2$; △ $n = 4$; × $n = 8$; + $n = 16$; ● $n = 32$

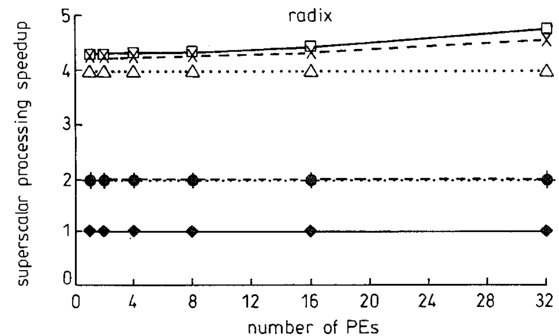

Fig.31 *Superscalar processing speedups of Radix in superscalar multiprocessing*
◆ $n = 1$; □ $n = 2$; △ $n = 4$; × $n = 8$; + $n = 16$; ● $n = 32$

Figs. 32–35 show the overall speedups of the benchmarks. It is observed that the parallelism, both instruction-level and task-level, is fully exploited in

superscalar multiprocessing. The sustained overall speedups of the FFT, LU, Ocean and Radix are 211.5, 30.4, 180.5 and 126.4, respectively. In summary, the FFT has the highest task-level parallelism and a substantial instruction-level parallelism so that it achieves the highest overall speedup. Although the LU has the highest instruction-level parallelism, it shows the lowest overall speedup due to insufficient task-level parallelism.

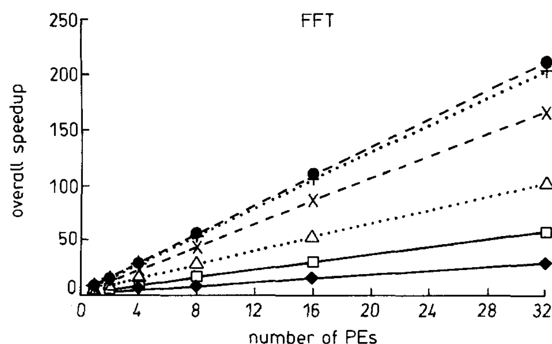

Fig. 32 *Overall speedups of FFT in superscalar multiprocessing*
◆ n = 1; □ n = 2; △ n = 4; × n = 8; + n = 16; ● n = 32

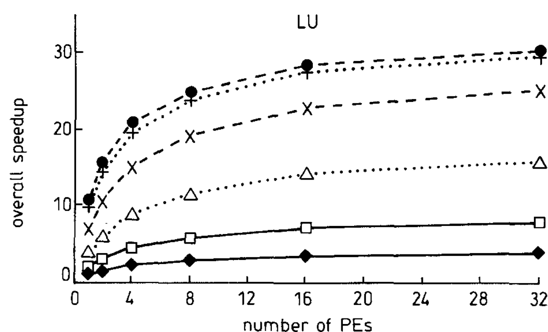

Fig. 33 *Overall speedups of LU in superscalar multiprocessing*
◆ n = 1; □ n = 2; △ n = 4; × n = 8; + n = 16; ● n = 32

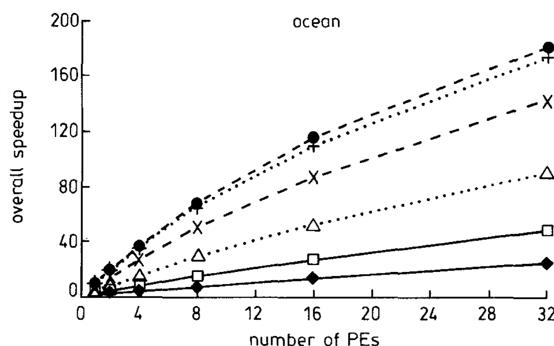

Fig. 34 *Overall speedups of Ocean in superscalar multiprocessing*
◆ n = 1; □ n = 2; △ n = 4; × n = 8; + n = 16; ● n = 32

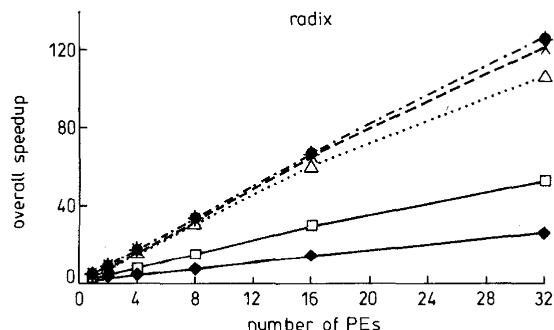

Fig. 35 *Overall speedups of Radix in superscalar multiprocessing*
◆ n = 1; □ n = 2; △ n = 4; × n = 8; + n = 16; ● n = 32

4.4 Discussion

To obtain an insight into parallelism exploitation in the benchmark programs individually, we identify the two types of parallelism of the benchmark programs in Table 3. With high task-level parallelism and moderate instruction-level parallelism, the FFT achieves the highest overall speedup. For this type of program, we suggest that a system with a moderate degree of superscalar processing and a high degree of multiprocessing be used to exploit the most instruction-level and task-level parallelism in programs. For example, a 32-way multiprocessor with eight-issue processor elements can speed up the FFT by over 200 times relative to a single-issue uniprocessor (as shown in Fig. 32).

Table 3: Parallelism classification of the benchmarks

Benchmark	Task-level parallelism	Instruction-level parallelism
FFT	high	moderate
LU	low	high
Ocean	moderate	high
Radix	moderate	low

The LU has the lowest task-level parallelism (its sustained multiprocessing speedup is only about four). However, its inherent instruction-level parallelism is nearly 12. For the LU, superscalar processing is more beneficial than multiprocessing. From Fig. 33, it can be observed that the overall speedup of a four-issue uniprocessor can easily outperform a 32-way multiprocessor with one-issue processors. For this type of program, we suggest that superscalar processing is more appropriate than multiprocessing in exploiting parallelism in programs.

Both the Ocean and the Radix have moderate task-level parallelism. Yet the Ocean has high instruction-level parallelism, whereas the Radix has low instruction-level parallelism. If two programs have the same task-level parallelism, their instruction-level parallelism determines the overall speedup. As a result, the Ocean achieves higher overall speedup than the Radix does (see Figs. 34 and 35). However, the superscalar processing speedup of the Ocean degrades more sharply than that of the Radix as the multiprocessing degree increases (see Figs. 30 and 31). So, we conclude that a system with a moderate degree of superscalar processing and a high degree of multiprocessing is appropriate for programs with moderate task-level parallelism.

In summary, we observed that exploiting task-level parallelism via multiprocessing is much more efficient than exploiting instruction-level parallelism via superscalar processing for the set of benchmark programs that we chose. In the m-way multiprocessing, the speedup can be almost linear for most of the benchmarks programs. In contrast, the inherent instruction-level parallelism of benchmarks ranges from only 4.29 to 12.08. Obviously, increasing the multiprocessing power gains more performance than increasing the superscalar processing power in general. Furthermore, instruction-level parallelism is also constrained by task-level parallelism in superscalar multiprocessing. As a superscalar

multiprocessor system design guideline, we suggest that the parallelism in programs can best be exploited by a moderate degree of superscalar processing and a high degree of multiprocessing.

5 Conclusions and future work

In this paper, we investigated the parallelism exploitation in superscalar multiprocessor systems. To enable accurate simulation of superscalar multiprocessor systems behaviour, we developed a simulator, called SMINT, for superscalar multiprocessor systems. The SMINT models both a superscalar processor that can exploit instruction-level parallelism, and a shared-memory multiprocessor system that can exploit task-level parallelism. With this simulator, we ran four applications chosen from the SPLASH-2 benchmark suite to examine the parallelism exploitation capabilities in the systems of multiprocessing, superscalar processing, and superscalar multiprocessing. We found that the parallelism in programs can best be exploited by a moderate degree of superscalar processing and a high degree of multiprocessing. For example, the speedup of a 32-way multiprocessor with eight-issue processor elements can be over 200 relative to a single-issue uniprocessor.

In this paper, we assumed a perfect memory system (the PRAM model). We will study the impact of memory system design on superscalar multiprocessor systems in the future. Furthermore, we intend to study a variety of architectural alternatives of superscalar multiprocessor systems, such as a single-chip multiprocessor and multiprocessor clusters, to further identify the different levels of parallelism and locality in programs. These research topics concerning superscalar multiprocessor systems will also include the tradeoffs of parallelism exploitation and locality management, the impact of ILP processors on memory consistency models and the implementation of speculative memory access techniques.

6 References

1 Cray superserver CS6400 product (Cray Research, Inc., Eagan, MN, USA, 1993)

2 Cray/T3D technical summary (Cray Research Inc., Eagan, MN, USA, October 1993

3 KSR technical summary (Kendall Square, Research, 1993)

4 CEKLEOV, M.: 'SPARCcenter 2000: Multiprocessing for the 90's!'. Proceedings of Compcon Spring 93, San Fransisco, CA, USA, February 1993, pp. 345–353

5 BREWER, E.A., DELLAROCAS, C.N., COLBROOK, A., and WEIHL, W.E.: 'Proteus: A high-performance parallel architecture simulator'. Technical Report MIT/LCS 516, Massachusetts Institute of Technology, Cambridge, MA, USA, September 1991

6 COVINGTON, R.C., MADALA, S., MEHTA, V., JUMP, J.R., and SINCLAIR, J.B.: 'The Rice parallel processing testbed',ACM SIGMETRICS international conference on *Measurement and modeling of computer systems*, Sante Fe, NM, USA, 1988, pp. 4–11

7 DAVIS, H., GOLDSCHMIDT, S.R., and HENNESSY, J.: 'Multiprocessor simulation and tracing using Tango'. Proceedings of 1991 international conference on *Parallel processing*, Austin, TX, USA, 1991, Vol. II, pp. 99–107

8 VEENSTRA, J.E., and FOWLER, R.J.: 'MINT tutorial and user manual'. Technical report 452, University of Rochester, Rochester, New York, USA, June 1993

9 WOO, S.C., OHARA, M., TORRIE, E., SINGH, J.P., and GUPTA, A.: 'The SPLASH-2 programs: Characterization and methodological considerations'. Proceedings of the 22nd annual international symposium on *Computer architecture*, Santa Margherita, Ligure, Italy, June 1995, pp. 24–36

10 BOYLE, J., BUTLER, R., DISZ, T., BLICKFELD, B., LUSK, E., and OVERBEEK, R.: 'Portable programs for parallel processors' (Holt, Rinehart and Winston, 1987)

11 BAILEY, D.H.: 'FFT's in external or hierarchical memory', *J. Supercomput.*, 1990, **4**, (10), pp. 23–35

12 SINGH, J.P., WEBER, W.-D., and GUPTA, A.: 'SPLASH: Stanford parallel applications for shared memory', *Computer Archit. News*, 1992, **20**, (1), pp. 5–44

13 BRANDT, A.: 'Multi-level adaptive solutions to boundary-value problems', *Math. Comput.*, 1977, **31**, (138), pp. 333–390

14 BLELLOCH, G.E., LEISERSON, C.E., MAGGS, B.M., PLAXTON, C.G., SMITH, S.J., and ZAGHA, M.: 'A comparison of sorting algorithms for the connection machine CM-2'. Proceedings of the symposium on *Parallel algorithms and architectures*, Hilton Head, South Carolina, USA, July 1991, pp. 3–16

15 FORTUNE, S., and WYLLIE, J.: 'Parallelism in random access machines'. Proceedings of the 10th ACM symposium on *Theory of computing*, San Diego, CA, USA, May 1978, pp. 114–118