超純量多重處理中之平行度開發
Parallelism Exploitation in Superscalar Multiprocessing

盧能彬　　鍾崇斌
Neng-Pin Lu and Chung-Ping Chung

國立交通大學資訊工程系
Department of Computer Science and Information Engineering
National Chiao Tung University
{nplu, cpchung}@csie.nctu.edu.tw

摘要

藉由開發程式中所有的平行度，超純量多重處理機系統已成爲高速計算機系統設計的趨勢。爲評估超純量多重處理機系統的效能，我們發展了一個能夠模擬超純量處理及多重處理的架構模擬器。藉由此架構模擬器，我們模擬了四個選自 SPLASH-2 的標竿程式來探討超純量多重處理中程式平行度的開發。模擬結果顯示，在理想的 PRAM 記憶體模式下，中度的超純量處理配合上高度的多重處理就足以完全開發程式中的平行度；相對於一個 1-issue 處理器，一個具有 32 個 8-issue 處理器的超純量多重處理機系統，大約可加快 200 倍的執行速度。

關鍵字：超純量處理，多重處理，平行度

Abstract

To exploit more parallelism in programs, superscalar multiprocessor systems have been the trend in designing high speed computing systems. In this research, we developed a simulator for evaluating superscalar multiprocessor systems. This simulator models both superscalar processor that can exploit instruction-level parallelism and shared-memory multiprocessor system that can exploit task-level parallelism. We used this simulator to run four applications chosen from SPLASH-2 benchmark suites, and collected some performance data to investigate the parallelism exploitation capability of the superscalar multiprocessor systems. We observed that the instruction-level and task-level parallelism in programs can be fully exploited by a moderate degree of superscalar processing and a high degree of multiprocessing when the memory system is the perfect PRAM model. For example, the speedup of 32-way multiprocessor with 8-issue processors can be over 200 relative to a single-issue uniprocessor.

Keywords: superscalar processing, multiprocessing, parallelism

1. Introduction

To exploit more parallelism in programs, superscalar multiprocessor systems have been the trend in designing high speed computing systems. Example of such systems include Cray SuperServer 6400 [8], Cray T3D System [9], Kendall Square Research KSR-1 [12], and Sun SparcCenter 2000 [6]. While superscalar processing exploits the instruction-level parallelism (ILP) within a processor, multiprocessing exploits task-level parallelism between processors. Superscalar multiprocessor systems provide enormous computing power by exploiting both fine-grained and coarse-grained parallelism in programs.

When designing a multiprocessor system, performance projection is an important study to the multiprocessor architecture decisions. There are three methods to verify a multiprocessor system: *prototyping*, *analytical modeling*, and *simulation*. Prototyping can predict most accurately the behavior 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 modeling uses the simplified parameter set or probability distribution to model a system. However, due to the complexity of real systems, modeling 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 desired detail. Simulation also allows the study of the behaviors of many design alternatives in a very short turn-around time and at a relatively low cost, reducing the effort and developing time of the system.

Currently, there are many multiprocessor simulators available. Examples are the Proteus [5], RPPT [7], and TangoLite [10]. 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 accesses. In this research, we developed a simulator for performance evaluation of superscalar multiprocessor systems. We also used this simulator to run a number of benchmark programs to investigate the parallelism exploitation capability of the superscalar multiprocessor systems.

The rest of this paper is organized as follows. Section 2 describes our simulator that models superscalar multiprocessor systems. Section 3 describes the benchmark programs. Section 4 presents and discusses the simulation results of parallelism exploitation capability of the superscalar multiprocessor systems. Section 5 concludes this paper and addresses our future work.

2. Superscalar Multiprocessor Simulator

2.1 MINT--A RISC Multiprocessor Simulator

Our superscalar multiprocessor simulator is based on MINT [15], a RISC multiprocessor simulator that supports MIPS R3000 instruction set. The major characteristics of the original MINT is briefly described as follows. MINT is a program-driven simulator as shown in Figure 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. A program-driven simulator can be partitioned into two main parts: a memory reference generator (also called the "front end"), and a target system simulator (also called the "back end"). The 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. Currently, MINT only supports MIPS R3000 RISC core. By being linked to proper back-end that describes the memory hierarchy, MINT can simulate the multiprocessor system with single-issue RISC processors.

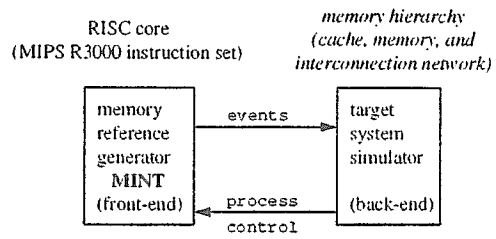

Fig. 1 MINT simulator.

2.2 SMINT--A Superscalar Multiprocessor Simulator

To enable accurate simulation of multiprocessor systems using superscalar processors, we modified MINT to support superscalar processing in the processor core. We call the modified simulator SMINT (superscalar MINT). SMINT uses the same MIPS R3000 instruction set of MINT and supports a variety of ILP features of contemporary superscalar microprocessors Figure 2 shows the SMINT simulator. SMINT has the following features:

-- Superscalar execution--multiple instructions can be issued per cycle

-- Dynamic instruction scheduling (out-of-order execution)

-- Register renaming

-- Dynamic branch prediction and speculative execution

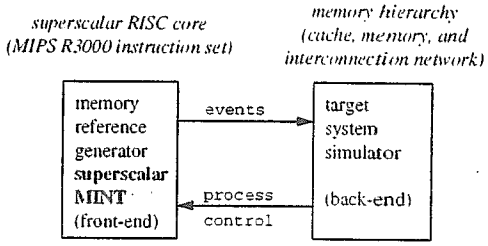

Fig. 2 The SMINT simulator.

By being linked to a proper back-end that describes the memory hierarchy, SMINT can simulate the multiprocessor system with superscalar processors. Figure 3 shows an example of superscalar multiprocessor system that SMINT can simulate. In this system, an interconnection network connects all of the processing elements (PEs). A processing element is composed of a superscalar processor, an instruction cache, a data cache, a shared memory module, and a network interface.

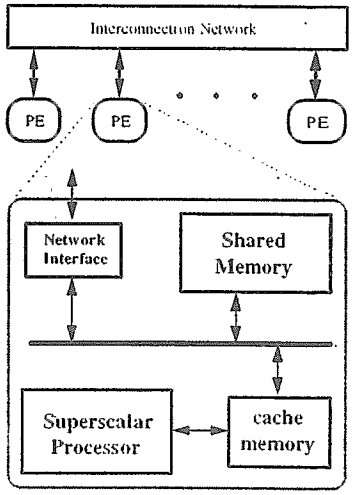

Fig. 3 A superscalar multiprocessor system.

Figure 4 illustrates the processor microarchitecture that can be modeled by SMINT. The instruction control unit is the central controller of the superscalar processor. It handles instruction address generation, instruction fetching, interrupts, and so forth. 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 corresponding 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 2-bit saturation counters is used to perform conditional branch prediction and support speculative execution.

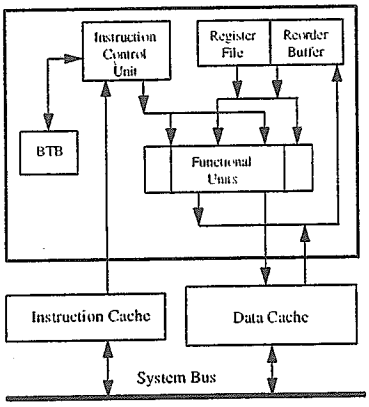

Fig. 4 The superscalar processor model.

To guarantee correct 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 lookahead 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 results. In the superscalar processor model, the execution time for all instructions is one cycle. And 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.

3. Benchmark Programs

In this section, we describe the benchmark suite we used--the SPLASH-2 [14], which is also widely accepted in studying centralized and distributed shared-address-space multiprocessors. The original SPLASH-2 programs were annotated by ANL macros [3] for multiprocessor execution. After expanding the macros into c codes by UNIX utility m4, 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. Figure 5 outlines our simulation flow.

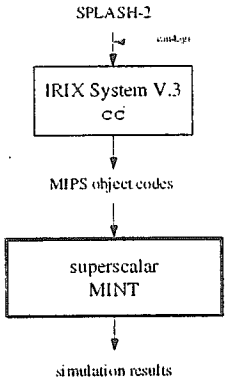

Fig. 5 Execution flow of the simulation.

SPLASH-2 consists of a mixture of complete applications and computational kernels. It currently has 8 complete applications and 4 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 descried in [1], which is optimized to minimize 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 organized 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 require all-to-all interprocessor communication. Assume p is total number of processors, every processor transposes a contiguous submatrix of $\left(\sqrt{n}/p\right) \times \left(\sqrt{n}/p\right)$ from every other processors, and transposes one submatrix locally. The transposes 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 [13]. 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, and (iii) it uses a red-black Gauss-Seidel multigrid equation solver [4], rather than an SOR solver.

3.4 Radix

The integer radix sort kernel is based on the method described in [2]. 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 generated 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 provides the input problem sizes, of the benchmarks that we used.

Table 1. Input problem sizes of benchmarks

Code	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 about parallelism exploitation collected by SMINT. To avoid the performance impact caused by memory system, we assume that the memory system is perfect (PRAM model [11]), so that all memory references complete in a single cycle. To exploit parallelism at different levels, we simulated the systems in multiprocessing, superscalar processing, and superscalar multiprocessing configurations as follows.

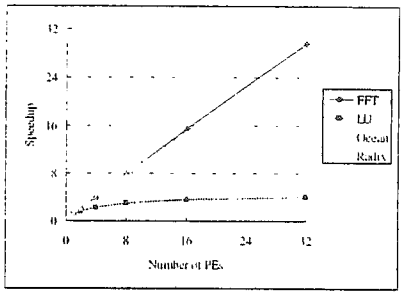

Fig. 6 Speedup due to multiprocessing.

4.1 Multiprocessing

Figure 6 shows the speedup of multiprocessor systems with single-issue RISC processors. (Only the parallelizable 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. The reason why LU fails to achieve a linear speedup is that LU spends much time in blocking due to synchronization. For all the other benchmarks, the blocking time also restrains them from having a perfect linear speedup. Among these benchmarks, FFT obtains the maximum speedup due to the least blocking time.

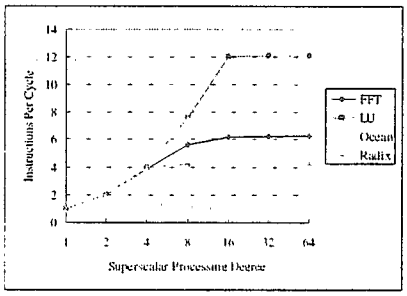

Fig. 7 Instruction-level parallelism of the four benchmarks on a superscalar processor with perfect branch prediction and infinite instruction window size.

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. Figure 7 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), 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 efficiency and limited instruction window size on the instruction-level parallelism.

4.2.2 Branch Prediction Affecting Instruction-Level Parallelism

SMINT models BTB with 2-bit saturation counter. Based on simulation results, we found that 256 BTB entries with 2-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, 256 2-way set associative BTB entries are assumed in the following simulations. Figure 8 shows the impact of real 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.

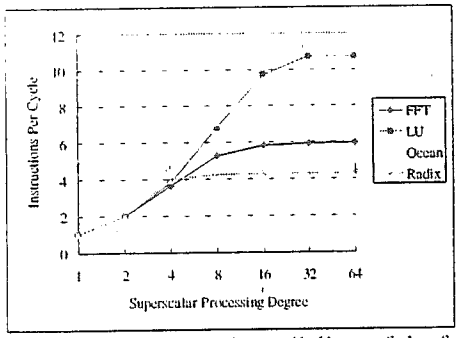

Fig. 8 Instruction-level parallelism of the four benchmarks on a superscalar with 256-entry, 2-way set associative, 2-bit saturation counter BTB prediction.

4.2.3 Instruction Window Size Affecting Instruction-Level Parallelism

Figure 9 shows the effect of instruction window size on the exploitable instruction-level parallelism. The instruction window size is the maximum number of instructions that can be scheduled dynamically in the processor. 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 exploitable instruction-level parallelism is very insignificant after the instruction window size grows above a certain threshold. From Figure 9, 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.

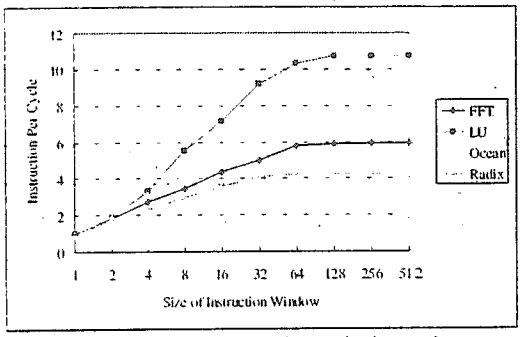

Fig. 9 Effect of instruction window size on instruction-level parallelism.

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 which has n homogeneous functional units, with a 2-way set associative BTB of 256 entries, and an 128-entry instruction window.

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

$$\frac{\text{Execution time of } n\text{ - issue uniprocessor}}{\text{Execution time of } m\text{ - way multiprocessor with } n\text{ - issue processors}}$$

and the *overall speedup* as

$$\frac{\text{Execution time of 1 - issue uniprocessor}}{\text{Execution time of } m\text{ - way multiprocessor with } n\text{ - issue processors}}$$

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

Figures 10 shows the multiprocessing speedup of the benchmarks. For the FFT, the multiprocessing speedup is independent from the superscalar processing degree. This is because FFT spends little time in being blocked, so that as the processor element speed increases, the multiprocessing speedup can still be maintained. However, the multiprocessing speedups of LU and Ocean are lowered as the processor element speed increases, indicating that the systems with high speed processor elements will cause 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 and near 100% branch prediction accuracy, so that the instruction window is often full in a high-issue degree processor. Therefore, as the number of processors increases, more number of instruction windows in the processors will be able to exploit more instruction-level parallelism, so that the multiprocessing speedup increases.

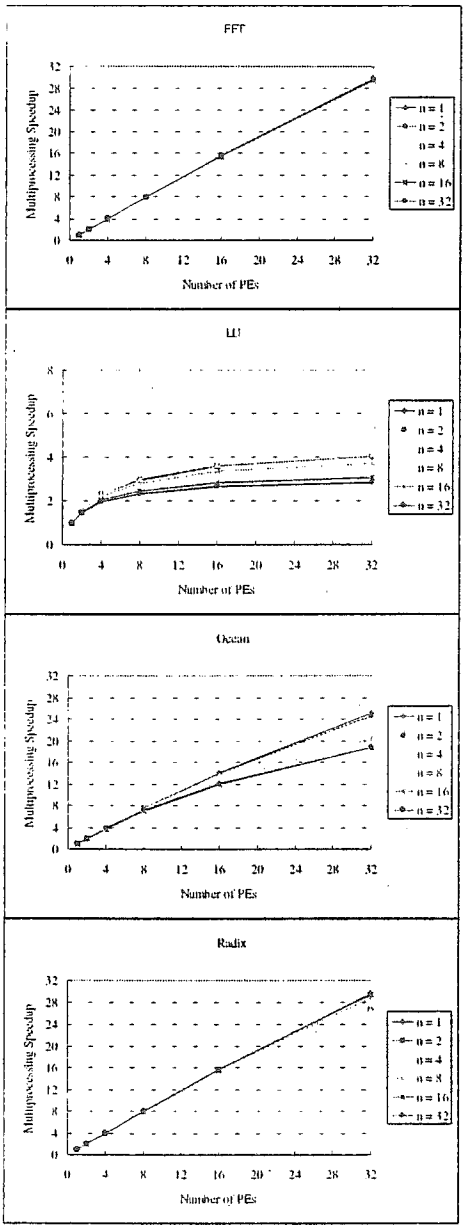

Fig. 10 Multiprocessing speedups of the four benchmarks in superscalar multiprocessing.

Figure 11 shows the overall speedups of the benchmarks. It is observed that the parallelism--both instruction-level and task-level--is fully exploited in superscalar multiprocessing. When the total number of PEs is 32, 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.

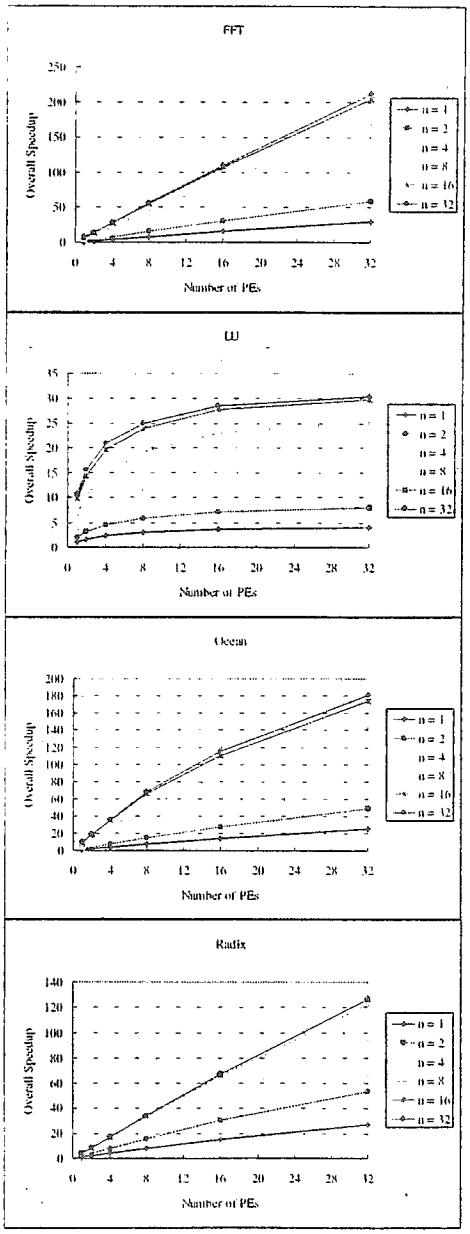

Fig. 11 Overall speedups of the four benchmarks in superscalar multiprocessing.

4.3 Discussion

Superscalar multiprocessing systems exploit both instruction-level and task-level parallelism in programs. From the above simulation results, it is noticed that

exploiting task-level parallelism via multiprocessing is much more useful than exploiting instruction-level parallelism via superscalar processing. In the *m*-way multiprocessing, the parallelization speedup can be approaching perfectly linear. In contrast, the inherent instruction-level parallelism of benchmarks ranges only from 4.29 to 12.08. Thus, increasing the multiprocessing power gains more performance than increasing the superscalar processing power in general. From the simulation results, we suggest that a system with a moderate degree of superscalar processing and a high degree of multiprocessing can exploit the most instruction-level and task-level parallelism in programs. For example, a 32-way multiprocessor with 8-issue processor elements can speed up the FFT by over 200 times relative to a single-issue uniprocessor (as shown in Figure 11).

5. Conclusion and Future Work

In this paper, we investigated the parallelism exploitation in the superscalar multiprocessor systems. To enable accurate simulation of superscalar multiprocessor systems behavior, we developed a simulator, called SMINT, for superscalar multiprocessor systems. The SMINT models both superscalar processor that can exploit instruction-level parallelism and shared-memory multiprocessor system that can exploit task-level parallelism. With this simulator, we ran a number of applications chosen from SPLASH-2 benchmark suite to examine the parallelism exploitation in the systems of multiprocessing, superscalar processing, and superscalar multiprocessing. We found that the parallelism in programs can be best exploited by a moderate degree of superscalar processing and a high degree of multiprocessing. For example, the speedup of a 32-way multiprocessor with 8-issue processor elements can be over 200 times 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 designs of superscalar multiprocessor systems, such as single-chip multiprocessor and multiprocessor clusters, to exploit all the possible parallelism and locality in programs. The research topics about 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.

References

[1] David H. Bailey, "FFT's in External or Hierarchical Memory," *Journal of Supercomputing*, 4(10):23-35, March 1990.

[2] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha, "A Comparison of Sorting Algorithms for the Connection Machine CM-2," in *Proceedings of the Symposium on Parallel Algorithms and Architectures*, pp. 3-16, July 1991.

[3] J. Boyle, R. Butler, T. Disz, B. Blickfeld, E. Lusk, and R. Overbeek, *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, 1987.

[4] Achi Brandt, "Multi-Level Adaptive Solutions to Boundary-Value Problems," *Mathematics of Computation*, 31(138):333-390, 1977.

[5] E.A. Brewer, C.N. Dellarocas, A. Colbrook, and W.E. Weihl, "Proteus: A High-Performance Parallel Architecture Simulator," *Technical Report MIT/LCS 516*, Massachusetts Institute of Technology, September 1991.

[6] M. Cekleov, et al. SPARCcenter 2000: Multiprocessing for the 90's!" in *Proc. Compcon Spring 93*, pp. 345-353, Feb. 1993.

[7] R.C. Covington, S. Madala, V. Mehta, J.R. Jump, and J.B. Sinclair, "The Rice Parallel Processing Testbed," in *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 4-11, 1988.

[8] *Cray Superserver CS6400 Product*, Cray Research, Inc., Eagan, MN, 1993.

[9] *Cray/T3D Technical Summary*, Cray Research, Inc., Eagan, MN, October 1993.

[10] H. Davis, S.R. Goldschmidt, and J. Hennessy, "Multiprocessor Simulation and Tracing Using Tango," in *Proc. 1991 Int'l Conf. Parallel Processing*, Vol. II, pp. 99-107, 1991

[11] S. Fortune and J. Wyllie, "Parallelism in Random Access Machines," in *Proceedings of the Tenth ACM Symposium on Theory of Computing*, May 1978.

[12] *KSR Technical Summary*, Kendall Square Research, 1993.

[13] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta, "SPLASH: Stanford Parallel Applications for Shared Memory," *Computer Architecture News*, 20(1):5-44, March 1992.

[14] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceeding of the 22nd Annual International Symposium on Computer Architecture*, pp. 24-36, June 1995.

[15] J.E. Veenstra and R.J. Fowler, "MINT Tutorial and User Manual," *Technical Report 452*, University of Rochester, June 1993.