Application of Instruction Analysis/Synthesis Tools to x86's Functional Unit Allocation

Ing-Jer Huang, Ping-Huei Xie
Institute of Computer and Information Engineering
National Sun Yat-Sen University
Kaohsiung, TAIWAN 804, R. O. C.
EMAIL: ijhuang@cie.nsysu.edu.tw

Abstract

Designing a cost effective superscalar architecture for x86 compatible microprocessors is a challenging task in terms of both technical difficulty and commercial value. One of the important design issues is the measurements of the distribution of functional unit usage and the micro operation level parallelism (MLP), which together determine the proper allocation of functional units in the superscalar architecture. To obtain such measurements, an x86 instruction set CAD system x86 Workshop is developed, which consists of both instruction set analysis and optimization tools. x86 Workshop has been applied to analyze several popular Windows95 applications such as Word, Excel, Communicator, etc. The MLP and distribution of functional unit usage are measured for these applications. The measurements are used to evaluate several existing x86 superscalar processors and suggest future extension.

1. Introduction

Improving the performance of microprocessors executing Intel's x86 instruction set has been a meaningful and demanding work, because of, not only their remarkable market volume, but also their technical challenge. Of all the architectural alternatives, the *superscalar* architecture, found in Intel's Pentium [2] and Pentium Pro [3], AMD's K5 [4] and Cyrix's 6x86 [5] *etc*., has been shown as an effective solution to improve the performance of x86 instruction execution.

The general practice to speed up x86 instruction execution is a two-layered microarchitecture, as shown in Figure 1. The outer layer fetches x86 instructions from the instruction cache, and translates the x86 instructions into simpler operations (called micro operations *MOP*'s or RISC-like instructions) that are executed in the inner layer of the microarchitecture. The inner layer, the area surrounded by the dashed line, adopts a superscalar core to speed up the execution.

We are currently participating in a research project, called NSC98, to build a high performance superscalar x86 compatible microprocessor. One important design issue of the superscalar architecture is the allocation of functional units: how many integer units, load/store units, branch unit, floating point units are necessary? The keys to answer these questions are the *micro operation level parallelism* (MLP) and the distribution of functional unit usage, which should be derived from typical and real application software, such as MS-Word, MS-Excel, *etc*.

A straightforward approach to obtain such statistics is to construct a simulator for the superscalar architecture and execute the simulator with real application software. However, such approach is impractical due to its tremendous amount of overheads, especially during the *design exploration* phase of the microprocessors. First is the overhead in simulator construction. In order to make it possible for the simulator to execute real binary code, it is necessary to construct every detail of the superscalar architecture, including x86-to-MOP decoders, the reorder buffer, the reservation station, register renaming, data forwarding, functional units, memory translation, *etc*. Second is the overhead in the required simulation time. According to our primarily experiments, the typical speed of such superscalar simulator ranges from a few tens to just a little bit more than one hundred x86 instructions simulated per second. With such speed, it is almost impossible to simulate real application software. Third is the overhead in operating system environment support.

Therefore, a fast performance/cost approximation tool is highly needed to explore the design boundaries and identify feasible design choices during the early stage of the design process. Based on the above observation, we propose a time-saving approach based on our instruction set CAD system *x86 Workshop*, which consists of three tools: *x86Bench*, *State Mapper*, and *ASIA-II*. x86Bench is an x86 application analysis system which produces disassembled x86 basic blocks annotated with their execution counts. State Mapper is an automatic tool which maps x86 instructions to MOPs for the given microarchitecture. Both x86Bench and State Mapper serve as the front end of ASIA-II, which is a second generation of our instruction

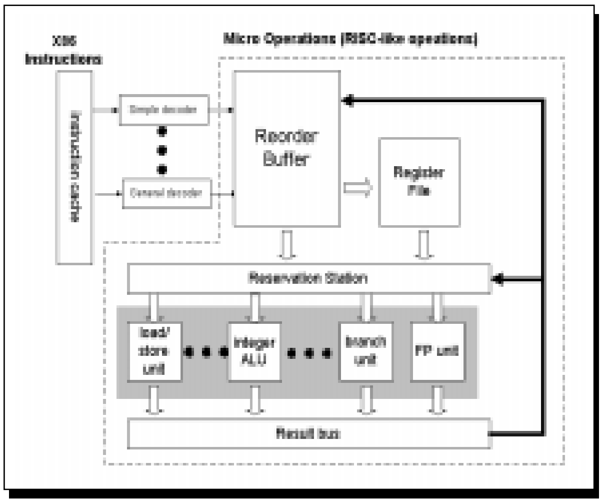


Figure 1. The superscalar architecture for x86 instruction execution

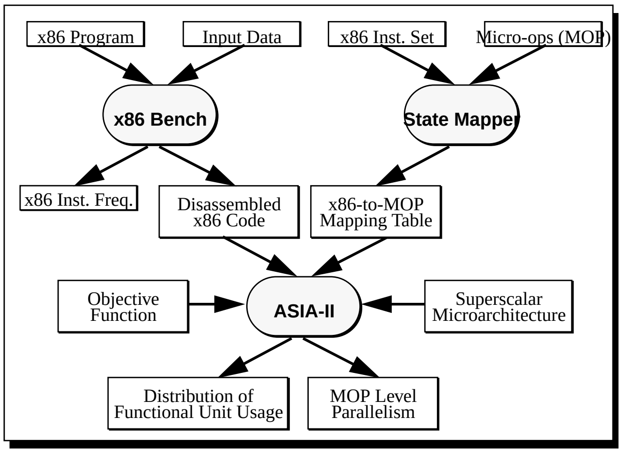


Figure 2. The x86 inst. set CAD system: x86 Workshop

synthesis tool ASIA [1]. ASIA-II enables us to investigate many interesting instruction behaviors in advanced superscalar architecture, including the distribution of functional unit usage and MLP that are interesting to our x86 study.

The rest of the paper is organized as follows. Section 2 describes the CAD framework and individual tools for the x86 instruction analysis. Section 3 presents the analysis of x86 application software and the results. Section 4 draws conclusions for this study and points out future direction.

2. The x86 Instruction Set CAD System: x86 Workshop

2.1 Framework

The framework of the x86 instruction set CAD system x86 Workshop is shown Figure 2. x86 Workshop consists of three tools: x86 Bench, State Mapper and ASIA-II. x86 Bench and State Mapper serve as the front-end of ASIA-II, which investigates superscalar features. In this paper we present the overall framework and the details of ASIA-II. Details of x86 Bench and State Mapper can be found in [6] and [7], respectively.

x86 Bench is an instruction analysis tool for the x86 instruction set, which is built around Intel's performance tuning tool VTune [10]. x86 Bench accepts an x86 program and its input data. The x86 program can be a DOS or Windows95 application. The tool can analyze x86 programs either with or without source code (high level language source code). For the given x86 program and its input data, the tool generates the x86 instruction usage frequencies and the disassembled code annotated with basic blocks' execution counts. Note that currently we are able to analyze instructions only belong to the application programs but not the operating system. To analyze instructions in the operating system we need the symbol files for Windows95's kernel which are not available [9].

State Mapper is an instruction retargeting tool. It translates a given assembly code from one instruction set to another instruction set, based on a machine state transition notation. It can be configured to solve our x86 problem, as illustrated in Figure 2. Each x86 instruction, due to its CISC nature, is considered as an assembly code, which is to be translated into a sequence of MOP's (i.e., micro sequence, or micro program). The MOP's of the target architecture is considered as the target instruction set for State Mapper. The generated micro sequences can be viewed as the entries of the x86-to-MOP mapping table.

ASIA-II reads in the disassembled x86 code generated by x86 Bench and maps the x86 code into MOP's, according to the x86-to-MOP mapping table generated by Sate Mapper. ASIA-II then schedules the MOP's into time steps, subject to constraints of their dependencies and the constraints of the given superscalar microarchitecture model. The superscalar microarchitecture model describes the supported micro-operations, operational delays and the topology of data path components. The numbers of data path resources can also be given as the resource constraints, or unspecified and let the tool to search for the best combination (w.r.t. to the given objective function). The user given objective function controls the scheduling. It can be configured to optimize for performance (as in the experiment of this paper), functional unit cost, or a combination of both. MOPs scheduled into the same time step represent MOPs that are executed in parallel in the superscalar core. From the scheduled MOP's the MLP and the distribution of functional unit usage can be obtained.

In the following sub-section, we present ASIA-II, the investigation tool for superscalar microarchitecture, in more details.

2.2 ASIA-II

ASIA-II is the second generation of our instruction set analysis/synthesis tool ASIA (Automatic Synthesis of Instruction set Architecture) [1]. ASIA analyzes and synthesizes application specific instruction sets for pipelined uni-processors. The kernel of ASIA is a micro operation scheduling engine based on a simulated annealing algorithm. When the instruction synthesis capability is desired, an instruction formation mechanism which is integrated into the scheduling engine can be turned on.

Since the internal superscalar core in Figure 1 can be regarded as an application specific RISC-based core with its sole application as an x86 instruction set emulator, ASIA can be tuned to study many design issues of x86 compatible microprocessors, such as the design of the internal RISC-based instruction set (MOP's) [8] and the functional unit allocation problem for the superscalar core, which is the focus of this paper.

To support the investigation into superscalar issues, ASIA needs the following improvements.

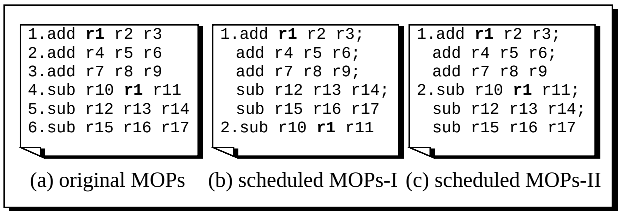
2.2.1 Distribution of functional unit usage

Using an instruction scheduling tool to investigate MLP and distribution of functional unit usage for superscalar architecture is a convenient approach but requires special cares. Otherwise, non-optimal designs may results.

For example, Figure 3 (a) shows a piece of MOP code. Figure 3 (b) (five MOPs in time step 1 and one MOP in time step 2) and Figure 3 (c) (three MOPs in time step 1 and three MOPs in time step 2) are two versions of its scheduled MOPs. Both versions take two time steps to finish. Both also have the same MLP of three (# of MOP's / # of time steps, 6/2). However, Figure 3 (b) requires five ALU's (to support the five MOP's in time step 1) to sustain such parallelism while Figure 3 (c) requires only three. This observation suggests that a scheduling algorithm which also tries to balance the resource usage while optimizing for performance is suitable to explore superscalar design space.

In addition, in a superscalar core, the relative order of operations is usually preserved during execution unless there are dependencies or there are some operations which take much more cycles than others to finish. For example, it is very unlikely that the sixth MOP (*sub r15 r16 r17*) in Figure 3 (a) is executed in time step 1 while the second MOP (*add r4 r5 r6*) being executed in time step 2, although the dependency relationship allows so. Therefore, the scheduling algorithm should also try to preserve the relative order while optimizing for performance.

To take care of the above two issues, ASIA-II adopts a scheduling algorithm based on local compaction with a simulated annealing approach. During the simulated



```
1.add r1 r2 r3
2.add r4 r5 r6
3.add r7 r8 r9
4.sub r10 r1 r11
5.sub r12 r13 r14
6.sub r15 r16 r17
```

(a) original MOPs

```
1.add r1 r2 r3;
  add r4 r5 r6;
  add r7 r8 r9;
  sub r12 r13 r14;
  sub r15 r16 r17
2.sub r10 r1 r11
```

(b) scheduled MOPs-I

```
1.add r1 r2 r3;
  add r4 r5 r6;
  add r7 r8 r9
2.sub r10 r1 r11;
  sub r12 r13 r14;
  sub r15 r16 r17
```

(c) scheduled MOPs-II

Figure 3. MOP schedules and distribution of functional unit usage

annealing process, MOPs are randomly selected and displaced to other time steps. The displacement is subject to dependence constraints and the displacement distance is limited. Therefore, a MOP will stay close to its original position as long as it does not lengthen the schedule. It is possible to displace a MOP further away from its original location through many iterations of displacements. However, ASIA-II performs such long range displacement only when performance can be improved. As a result, *ASIA-II produces optimized schedules with better distribution of functional unit usage as in the case of Figure 3 (c). In addition, the operation patterns in the schedules are more realistic and achievable in a superscalar core.*

2.2.2 Register renaming

The superscalar core in Figure 1 supports the register renaming mechanism in order to boost parallelism. Register renaming eliminates *anti* (write after read) and *output* (write after write) dependencies between a pair of operations by redirecting the write operation (the later operation in the dependent pair) to a different location. Later operations that read the write result are also redirected to the new location.

The register renaming feature is supported in ASIA-II by ignoring the anti and output dependencies among the MOP's during scheduling.

2.2.3 Hardware branch prediction

Branch instructions impede the instruction fetcher's capability to supply instructions at a sufficient rate to keep functional units busy. When the outcome of a branch instruction is not known, the instruction fetcher has to stall or incorrect instructions are fetched. A stalled instruction fetcher or incorrectly fetched instructions decrease the number of instructions ready to execute in parallel.

Hardware branch prediction aims to reduce the branch penalty by predicting, with hardware support, the direction of the current branch instruction based on its previous branch outcomes before the result of the current branch is known. If the branches are successfully predicted, the instruction fetcher is stalled for less times, and less number of incorrect instructions are fetched. The functional units in the data path would see more instructions ready for parallel execution. Therefore, branch predication reclaims potential parallelism which is undermined by branch instructions.

The effect of branch prediction is equivalent to the enlargement of basic blocks' sizes. The larger the basic block, the more chances the superscalar core can find instructions to execute in parallel.

ASIA-II adopts the basic block enlargement approach to account for the hardware branch prediction effect. Figure 4 illustrates this approach with a real example taken from a Window95 archives application Winrar. Figure 4 (a) shows a flow graph with basic blocks A through H. The number next to the basic block is its execution count. Basic block A executes for 11314 times, out of which 10594 times jumps to basic block B and 720 times jumps to basic block C. This flow graph can be differentiated into four versions of enlarged basic blocks (called *Eblocks*), as shown in Figure 4 (b). The Eblocks are derived by tracing down every path in the flow graph. While constructing the Eblocks, ASIA-II also derives their execution counts. For example, Eblock 1 is obtained by concatenating basic blocks A, B and F. Eblock 1 is executed for 10594 times. To model the variation in superscalar execution when entering the same basic block from different preceding blocks, different versions of Eblocks can be constructed for the same basic block.

In its implementation, ASIA-II provides a user defined parameter which controls how far ASIA-II should travel to construct Eblocks. In the experiment conducted in this paper, it is set to eight instructions; i.e., for a given basic block, ASIA-II searches for the next eight instructions from the flow graph and appends the eight instructions to the end of the given basic block to form an Eblock. There may be many versions of the eight-instruction pattern. Each version produces one Eblock.

2.2.4 Schedules, distribution and parallelism

Eblocks are then mapped into MOP's and the MOP's are scheduled into time steps, as shown in Figure 4 (c). The Eblocks are optimally scheduled into 5, 5, 5 and 4 time steps, respectively. Note that in the second time step of Eblock 2, two MOP's loading into the same register `ecx` are scheduled into the same time step. It's a legal schedule since one of the load will be renamed by hardware. There is a similar case in Eblock 3.

From the MOP patterns in the time steps, the distribution of functional unit usage can be derived. For example, the MOP's in the first time step of Eblock 1 need three load/store units (0A3M0B0F), which accounts for 19% (10594/(5*10594+5*250+5*158+4*312)) of executed time steps. Figure 5 shows the distribution of functional unit usage of all Eblocks. Note that the most significant functional unit usage patterns (3A0M1B0F, 2A0M1B0F, 1A1M0B0F, 0A3M0B0F and 0A1M1B0F) are contributed

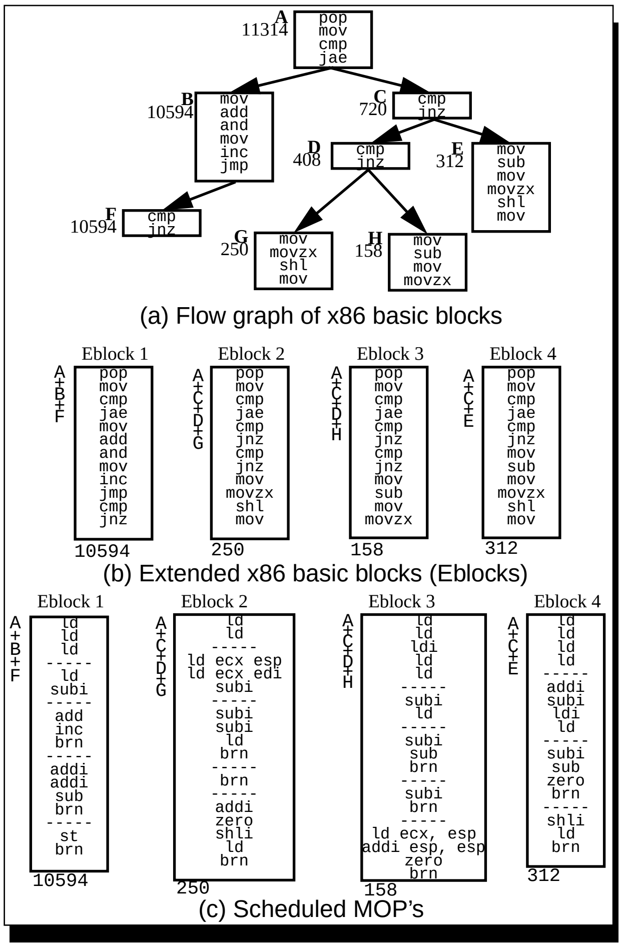


Figure 4. ASIA-II's approach to support branch prediction

by Eblock 1 since its execution count is much higher than others. The MLP for this example is 2.83 which is calculated with the following equation EQ 1. The weights in the equation specify the relative frequencies of the programs executed in a typical workload environment. In this example, the weight is set to one since there is only one program (the flow graph in Figure 4 (a)).

$$\text{MOP Parallelism} = \frac{\sum_{p}^{\text{Programs}} \sum_{e}^{\text{Eblocks}} \text{Weight}_p \times \text{Execution Count}_e \times \text{number of MOPs}_e}{\sum_{p}^{\text{Programs}} \sum_{e}^{\text{Eblocks}} \text{Weight}_p \times \text{Execution Count}_e \times \text{number of time steps}_e}$$

EQ 1

3. Analysis of x86 Application Software

In this section, we apply x86 Workshop to measure the potential MLP and the distribution of functional unit usage of several commercial Windows95 applications. Table 1 lists the Windows95 applications used in this experiment, including Microsoft's Word and Excel, Netscape's Communicator 4.03, Winzip, Winrar and Turbo95. These appli-

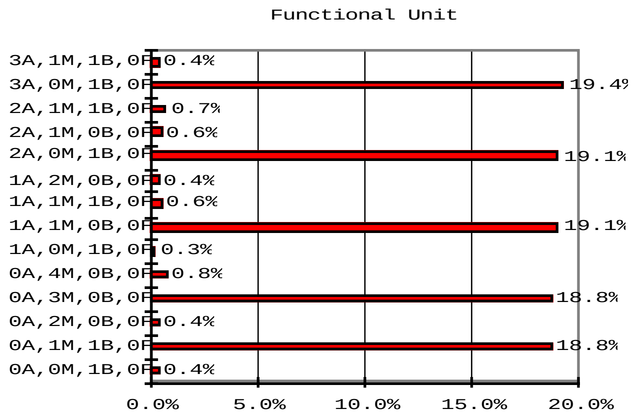


Figure 5. Distribution of functional unit usage in the schedules of Figure 4 (c)

Notation: A: integer unit, M: memory, B: branch unit, F: floating unit

cations are typical programs used by graduate students of computer engineering. In the table we list the numbers of executed instructions of the programs and the weights of programs which represent the relative frequencies of these applications used in a typical work environment. Note that Word and Excel execute less instructions than other applications, because they are interactive programs which spend most of the time in idle and waiting for users' inputs. The total number of instructions executed is over 445 million.

Program	Executed Inst[a].	Weight
MS-Excel 7.0	3,249,987	10
MS-Word 7.0	8,222,279	20
Netscape Communicator 4.03	70,573,959	15
Winzip 6.3	97,064,114	2
Winrar 2.02	79,824,168	1
Turbo95	187,063,542	1
TOTAL	445,998,049	

Table 1: Description of Windows95 applications under experiment

a. User instructions only, not including the service of operating system

The superscalar core under measurement is based on the superscalar model in Figure 1 with the following assumptions in order to obtain the maximal available MLP.

1. The cache is 100% hit; i.e., there is no delay cycles caused by cache miss.
2. The branch prediction is 100% accurate.
3. The instruction fetcher and decoders are fast enough to provide and decode sufficient instructions, in order to sustain the maximal MLP.
4. All the functional units are pipelined and the execution latency of functional units is one cycle, except the load/store unit which requires two cycles (the first cycle computing the effective address while the second cycle accessing the cache).
5. The reservation station is large enough to accommodate all ready MOPs and perform all necessary register renaming.

Table 2 lists the MLP's for the given programs. Note Winzip and Winrar have higher MLP's than Excel, Word and Communicator because the former are computation intensive jobs while the latter are interactive jobs which require more condition handling. Turbo95 is a PC performance measurement tool. It requires less number of function calls. The average size of basic blocks is larger than other programs. Therefore, its MLP is significantly larger than others. The average MLP for all the programs is 2.97 which is based on the equation EQ 1 and with the weights given in Table 1.

Program	MLP
MS-Excel 7.0	2.58
MS-Word 7.0	2.62
Netscape Communicator 4.03	2.76
Winzip 6.3	3.34
Winrar 2.02	3.73
Turbo95	4.96
AVERAGE	2.97

Table 2: Measured MLP

Figure 6 shows the overall distribution of functional unit usage of the MOP's in the time steps for these programs. Each bar represents the frequency of the corresponding pattern of functional unit usage in the programs. To save space, distributions of insignificant functional unit usage patterns (< 1.8%) are lumped together and listed under the label "others." The most frequent patterns are 2A0M0B0F (24.5%) and 1A0M0B0F (14%). The next frequent patterns uses two to three integer units and one to two load/store units. The result suggests that integer units and load/store (memory) units are the most critical functional units for efficient x86 execution. Note that these programs are mainly integer applications. The use of floating point units and MMX are very insignificant. Therefore, we will not discuss floating point and MMX units in the rest of the discussion.

A functional unit pattern can cover some other patterns. For example, the time steps of the pattern 1A0M0B0F can also be accommodated by the pattern 2A1M1B1F. Therefore, we need to calculate the accumulated coverage of time steps for possible allocations of functional units that we are considering. Figure 7 shows the accumulated coverage of several functional unit allocations found in existing x86 compatible microprocessors such as Pentium Pro, K5,

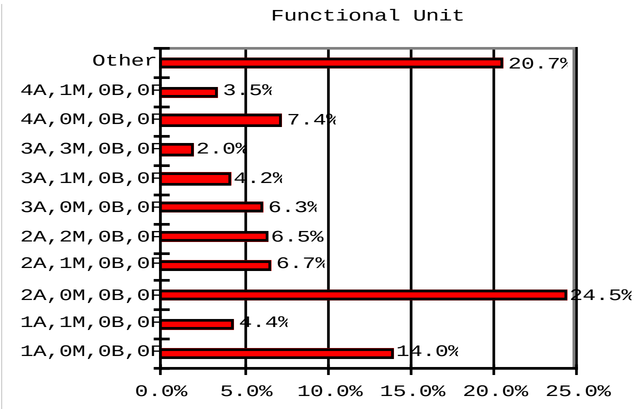


Figure 6. Distribution of functional unit usage

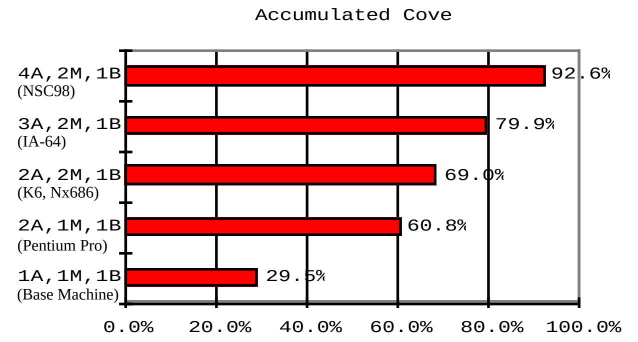


Figure 7. Accumulated coverage of functional unit patterns

Nx686, *etc*. The 1A1M1B stands for a base machine with one resource for each functional unit. It covers about 30% of available parallelism. With an additional integer unit, Pentium Pro is able to cover 61% of the available parallelism. Adding an extra load/store unit to Pentium Pro, as in K6 and Nx686, can increase the coverage up to 70%. Finally, by adding two integer units and one load/store units to K6, 93% of the available parallelism can be covered. Therefore, the 4A2M1B configuration is adopted in the planned NSC98 microprocessor.

The experiment of ASIA-II takes about 24 hours of computing time on four UltraSparc CPU's (one at 143MHz, two at 200MHz, and one at 270MHz).

4. Conclusions

We have developed an x86 instruction set CAD system x86 Workshop to measure the distribution of functional unit usage and the micro operation level parallelism (MLP), which together determine the proper allocation of functional units in the x86 compatible superscalar architecture. x86 Workshop consists of three tools: x86 Bench, State Mapper and ASIA-II. x86 Bench is an instruction analysis tool for the x86 instruction set. It produces disassembled code for a given Windows95 program and its input data, annotated with execution counts for the basic blocks in the disassembled code. State Mapper automatically generates the mapping table to map the x86 instruction set to MOP's for a given superscalar architecture.

ASIA-II reads in the disassembled x86 code generated by x86 Bench and maps the x86 code into MOP's, according to the x86-to-MOP mapping table generated by Sate Mapper. ASIA-II then schedules the MOP's into time steps. From the scheduled MOP's, the distribution of functional unit usage and average MLP can be measured. We have presented the necessary mechanisms in ASIA-II in order to support several superscalar features, such as register renaming, branch prediction, *etc*.

x86 Workshop has been successfully applied to analyze several popular Windows95 applications such as Word, Excel, Communicator, *etc*. The MLP and distribution of functional unit usage are measured for these applications. The measurements are used to evaluate the allocation of functional units in several existing x86 superscalar processors and suggest the improvement to our planned NSC98 x86 compatible superscalar microprocessor.

In the future, we'd like to conduct experiments to cover a much wider spectrum of Windows95 applications. In addition, we'd like to extend the capability of x86 Workshop to address other superscalar design issues, such as instruction pairing (instruction folding), elimination of short conditional branches, instruction decoder allocation, branch prediction depth, *etc*.

Acknowledgment: This paper is supported by NSC under 85-2262-E-009-010R and 86-2262-E-009-009.

References

[1] I. J. Huang and A. Despain, "Synthesis of Application Specific Instruction Sets," *IEEE Trans. on CAD*, June, 1995

[2] Dick Pountain, "Pentium: More RISC Than CISC," Byte Magazine, Sept. 1993.

[3] Linley Gwennap. "Intel's P6 Uses Decouple Superscalar Design," *Microprocessor Report* Vol. 9, No. 2, February 16, 1995.

[4] AMD's K5, http://www.amd.com/products/cpg/k5

[5] Cyrix's 6x86 (M1), http://www.cyrix.com/process/prodinfo/legacy

[6] I. J. Huang and T. C. Peng, "Analysis of x86 Instruction Set Usage for DOS/Windows Applications and Its Implication on Superscalar Design," submitted to ICCD 1998.

[7] I. J. Huang and W. F. Kao, "Instruction Retargeting Based on the State Pair Notation," *Proc. of the Asia-Pacific Conference on Hardware Description Languages*, Aug., 1997.

[8] I. J. Huang and J. M. Shiu, "Design of RISC-based Instructions for Efficient x86 Emulation," *Proc. of National Computer Symposium*, Taiwan, Dec. 1997

[9] Private communication with an engineer of Intel's VTUNE team, 1997

[10] Mark Atkins and Ramesh Subramaniam. *PC Software Performance Tuning.* IEEE Computer, August 1996