A SIMULATION TOOLKIT FOR x86-COMpatible
PROCESSORS — Xsim*

HUNG-CHANG HSIAO, CHUNG-TA KING
WEI-KUO CHEN and HSIAN-HSIUNG LIN
Department of Computer Science, National Tsing Hua University
Hsinchu, Taiwan 300, R.O.C.
e-mail: king@cs.nthu.edu.tw

CHIEN-CHAO TSENG
Department of Computer Science and Information Engineering
National Chiao Tung University, Hsinchu, Taiwan 300, R.O.C.

Received August 1999

ABSTRACT
Functional-level simulators have become an indispensable tool in designing today's processors. They help to exploit the design space and evaluate various design options so as to derive a suitable processor microarchitecture. Although Intel's x86 series processors are the most popular CPU in the computers, there are only a few simulation tools available for studying these processors. This paper introduces such a simulation tool. Internally it simulates a decoupled de-code/encode architecture and has a RISC core. It is trace-driven and thus has a tracing system, a trace sampling system, and a processor simulator. We will describe the internal workings of the simulation tool and demonstrate how it can be used in evaluating a specific x86-compatible processor.

1. Introduction. The demand for the processors' computing power is continuously rising. Many schemes have been devised to improve the performance of modern processors [1,11,12,17,25,29]. These schemes may be easily implemented on processors with a reduced instruction-set architecture at a high clock rate. But for processors with a complex instruction-set architecture, the effort of adopting these techniques may be very high and the design process may be very complex. Unfortunately the most widely used processors in today's computers, the x86-compatible processors, are based on the complex instruction-set architecture. To help reduce the design

*This work was supported in part by National Science Council under Grant NSC 85-2622-E-009-010R, 86-2622-E-009-009 and 88-2622-E-009-004.
complexity, recent designs of x86-compatible processors have adopted the *decoupled decode/encode architecture* [5] and relied on simulation tools to sift through various design options.

In the decoupled decode/encode architecture, a complex instruction is decoded and translated into simple, RISC-like operations. The succeeding pipeline stages can then process these RISC-like operations efficiently. Such a processor can be viewed as consisting of a RISC core coupled with an intelligent fetcher/decoder. In this way, the resultant processor can maintain an x86 instruction-set architecture while taking advantage of RISC architectures.

Even though the decoupled decode/encode architecture can reduce the design complexity and allow easy incorporation of many advanced techniques to boost performance, the whole design process is still a very complex undertaking due to huge design space — growing with each new technique introduced. Functional-level simulators [9] are thus introduced to prune the design space and they are unlike circuit-level or logic-level simulators, which allow for rapid prototyping, even if the design is not physically fabricated. Rapid prototyping allow architects to study the feasibility of a high level design before actually carrying out the tedious and low-level details such as placement and routing in circuit-level or logic-level simulators. In this paper, we introduce a functional-level simulation tool for evaluating the performance of x86-compatible processors adopting the decoupled decode/encode architecture.

The construction of a simulator should consider two important factors: preciseness and speed. We chose to build our simulator, called *Xsim*, using the trace-driven approach, because trace-driven simulation allows fast simulation with little loss of preciseness. In addition, techniques such as *sampling* can be used with traces if simulation speed is a concern [8,16,21]. Our simulation tool includes a tracing system, a trace sampling system, and a processor simulator. With these tools we have exploited the design space and evaluated different alternatives in designing a new x86-compatible processor.

The remainder of the paper is organized as follows. Section 2 reviews recent x86-based systems and available simulation tools. Section 3 describes our simulation tool for x86-compatible processors. Use of the simulation tool in designing a specific x86-compatible processor is demonstrated in Section 4. Finally, conclusions and possible future works are given in Section 5.

**2. Background.** In this section, we first review some x86-compatible processors and then examine several simulation tools, which were developed to help the design and evaluation of processors.

**2.1. The x86-compatible processors.** Due to the dominance of x86 processors in the computer world, many manufacturers have entered the
market with their own x86-compatible processors. Examples include Intel’s Pentium [10], Cyrix’s M1 [24], AMD’s K6 [5], and Rise’s mP6 [13]. The majority of these processors adopt the decoupled decode/encode architecture by translating a complex instruction into a number of simple RISC-like instructions. For example, in the Intel Pentium Pro processor, an x86 instruction is translated into a number of micro operations (uOP). In the AMD K6 processor, the RISC-like instructions are called the RISC86 operations (ROP).

Once an x86 instruction is translated to RISC-like operations, many advanced techniques for boosting processor performance can be easily applied [11,17]. For example, the Pentium Pro processor implements a 12-stage superpipeline with a superscalar microarchitecture, supporting two integer execution units (EU), one floating point EU, one branch prediction EU and two address generation units (AGU). It adopts the Tomasulo’s algorithm [27] to support dynamic execution and allow three x86 instructions to complete in one machine cycle. It also employs a variant of the Yeh’s algorithm [31] for dynamic branch prediction. Centralized reservation stations are used for renaming registers, which are accessible by the execution units.

2.2. Processor simulation tools. There are three different strategies to implement a processor simulator: synthetic workload-driven, trace-driven and execution-driven. We review these design strategies first and then list some representative tools.

- **Synthetic workload-driven:** Synthetic workload-driven simulation uses probability distributions or synthesized kernels to abstract the behavior of the workload. For example, Dally [6] abstracted behavior of several applications into several kernel programs, with which optimization of micro instructions of floating-point operations can be investigated. The abstracted model or the kernel program is then used as the input to the simulator. Using this abstract workload, the behavior of the input workload can be controlled with a number of simple parameters. In addition, the simple models allow fast simulation.

- **Trace-driven:** Trace-driven simulation uses traces as input for simulation. The traces record instructions in the dynamic execution path of a program and are collected from real or simulated machines. Only instructions of interest, e.g. load/store instructions or branch instructions, are collected. The Pixie tool [19], for example, can be used to annotate an object file so that traces can be collected when the annotated object file is executed.

- **Execution-driven:** Execution-driven simulation performs simulation as the benchmarking program runs. Thus the simulator is linked together with the benchmarking programs. Usually, it needs an interpreter to interpret the instructions just executed by the
benchmarking program and feed the interested instructions into the simulator. Alternatively it can annotate the benchmarking programs with simulation code and then runs the simulation without any interpretation. For example, MINT [28] uses the former strategy and Augmint [20] follows the latter for studying memory hierarchy.

Although there are many processor simulation tools available today, a majority of them are not designed for x86-based processors. Two well-known tools are Shade [26] from the Sun Microsystems and ATOM [7] from the Digital Equipment Corporation. They are execution-driven, but are designed to run on specific platforms — Shade for SPARC processors and ATOM for Compaq Alpha processors.

SimpleScalar [2] represents another attractive approach to processor simulation. It can support functional simulation for the x86 instruction set, with a proper translation of the x86 instructions to the SimpleScalar instructions. However, different x86-compatible processors have different RISC-like operations and have different microarchitectures. It is thus necessary to rewrite the RISC-like instructions when using SimpleScalar to study the processor at hand. This is quite complex and time consuming.

Recently the Etch [23] tool were developed for the Win32 execution environment. It is basically an execution-driven simulation that instrumentation is necessary to augment the binaries. The instrumentation may even include DLLs in the Win32 system. It can be used to trace the application bottleneck and help application developers to interact with architects for better application execution. Unfortunately the complete tool set is currently not available for the public.

3. The Simulation Toolkit — Xsim. In this section, we describe our simulation toolkit — Xsim. Its general organization is introduced first, followed by detailed descriptions of each major component. The tool has been used in a project to design a new generation x86-compatible microprocessor.

3.1. The basic structure. The structure of the simulation tool is shown in Fig. 1. It consists of five major parts: the specification of respective micro operations, the tracing system, the sampling system, the processor simulator and the machine configuration files. The specification of respective micro operations allows to define a group of RISC-like operations which correspond to an x86 instruction. The tracing system extracts traces of a running program. Currently it can collect traces under a UNIX or a MS Windows system. It interprets the instructions for the running processes and saves the interpreted instructions on files. Thus, Xsim does not require any source-level program code information, especially for MS Windows' applications. Our Xsim also allows to adopt the approach which is similar to
the execution-driven simulation by simulating instructions while interpreting them during their runtime without saving traces on the storage. The sampling system is used to analyze a full trace and helps to determine a suitable sampling configuration to increase the simulation speed while maintaining preciseness. The processor simulator models an x86-compatible processor with a configurable RISC-like core. The machine model can be defined by the parameter files. The simulation is cycle-based and the behavior of the target processor at each pipeline stage is modeled and simulated in detail. To avoid the input buffer problem, the simulation proceeds backward from the last stage to the first stage. We discuss the simulation of a processor core in the next section.

The whole simulation toolkit was developed with a modular structure. Each major architecture module, e.g. register file, reorder buffer, reservation table, etc., corresponds to a simulator module. This modularized structure has a number of advantages:

- **Match hardware modules and simplify simulator integration:** Functions of the simulator modules can be identified easily according to the functions of the hardware modules. Interface of each module can also be defined accordingly. From the view point of software engineering, modulization makes the functions of the modules clear and simulator integration simple. This in turn reduces the development efforts and allows quick debugging.

- **Simplify modification:** A simulator module may have many different ways of internal implementation. When the simulator design is forced to change, modularization gave us the freedom of internal modification without having to worry about other modules.

**Fig. 1. The structure of the Xsim simulation toolkit.**
• **Allow reconfiguration:** The simulator can be adapted to simulate new processors easily with modularization. All that we have to do is to change the internal implementation of the affected modules, and add/remove modules.

Besides, our Xsim toolkit allows to study different topics while developing a microprocessor. They include design of RISC-like instructions by rewriting the specification of primitive micro operations, sampling techniques by analyzing the trace for a given application and developing new sampling algorithms, and processor architectures by adding/removing architectural modules or rewriting their internal structures.

3.2. The tracing system. The tracing system in our simulation tool can collect traces of running programs under UNIX or MS Windows. The trace records several information about an instruction, including the registers used, the instruction type, its address, length, and the instruction itself. The tracing techniques for UNIX and Windows are different and thus they are discussed separately below.

- **UNIX:** On a UNIX system, e.g. Linux, we use the `ptrace()` system call. The tracing system first forks a child process to execute the program to be traced. It then uses the `ptrace()` system call with the parameter `PTRACE_SINGLESTEP` to interrupt the traced child process after each instruction executed. The instruction which has just been executed can then be retrieved through the parameters `PTRACE_PEEKTEXT` and `PTRACE_PEEKUSR`. This process is repeated until the child process exits, a condition that the parent process can check through the system call `WIFSTOPPED()`.

- **Microsoft Windows:** In Microsoft Windows (Windows 95/98 or Windows NT), the Win32 APIs [22] provides a set of DEBUG APIs to catch the events for debugging purpose. Using the APIs, we can read/write arbitrary data from/to the context of a traced process. The tracing procedure is then very similar to that in UNIX. The parent process uses `WaitForDebugEvent` [22] to claim its intention to catch all the debugging events which occurred in the child process, especially those “exception debug events”. When a child process is loaded, the parent process receives a breakpoint exception from the operating system, where it can insert some breakpoints in the child process. After the child process has executed one basic block, the parent process is wakened up with a signal. It can then examine the set of instructions encapsulated in the basic block just executed. Before it returns control back to the child process, it replaces the last instruction of the next basic block with a 0xCC breakpoint instruction. Also it needs to restore the last 0xCC instruction in the
current basic block to the original instruction. In this way the child process will be interrupted again after it executes the next basic block. Using this mechanism, we can collect traces under the Windows environment without changing the execution path.

3.3. The sampling system. The sampling system is used to obtain a representative, sampled trace out of a full trace. It helps to increase the simulation speed and to reduce the storage capacity required to store the traces. There are several sampling techniques proposed previously [8,16,18,21]:

- **One large sample**: A single and large sample from the trace is chosen.
- **Multiple periodic samples**: A fixed-size chunk of instructions are samples, followed by a fixed-size chunk of instructions skipped, and so on. The number of chunks collected can be determined according to a scaling factor.
- **Multiple random samples**: Samples are distributed randomly over the full trace.
- **Profile-driven samples**: The traced application is instrumented first to collect information on its execution, e.g. execution frequency of a basic block. The full trace is sampled according to the profiled information according to the desired scaling.

Currently our sampling system implemented the first three techniques. Effectiveness of sampling will be studied in the next section.

3.4. The processor simulator. In this section we present the processor simulator involved in the toolkit and then is followed by the validating models provided for the processor simulator.

3.4.1. The architectural modules. The processor simulator models a default RISC core as shown in Fig. 2(a). Note that it is a decoupled decode/encode architecture. Thus a complex x86 instruction needs to be first translated into a number of RISC-like micro-operations (RMO) [14]. Note also that the modularized design of the simulator allows this default configuration to be changed easily.

The default RISC core supports multiple issues, out-of-order execution, and in-order completion. The simulator itself consists of several major modules: instruction cache (ICache), data cache (DCache), fetcher, decoder, functional units, reservation stations (RS), out-of-order execution buffer (OOB), and register file (Reg). Each corresponds to one hardware module in the RISC core.

The simulator modules are written according to the object-oriented principle. For example, the basic data structure used inside a functional unit
Fig. 2. The default (a) RISC core and (b) pipeline stage simulated by the processor simulator.
module includes the tag of the RMO being executed, and a count recoding the cycles that the RMO is to occupy the unit. Different functional units then inherent the above data structure and extend it to include their own specific information. When a RMO is dispatched to the corresponding functional unit, its latency is loaded into the counter. The counter is decremented by one in every cycle until it becomes zero, which means the execution is completed.

The whole simulation is driven by a single clock. In each clock cycle, the simulator program examines the pipeline stages (Fig. 2(b)) from the last toward the first, to avoid the input buffer problem. For each pipeline stage, the simulator invokes related simulator modules and updates corresponding data structures. The behavior of the target processor at each pipeline stage is modeled and simulated in detail. We show the algorithm in Fig. 3.

3.4.2. The validating models. The validating models provided by Xsim adopt the incremental approach. That is the validating models are classified into four aspects — the resource dependency model, the data dependency model, the control dependency model and the memory model. These models are used to verify the correctness of the simulator implementation.

- Resource dependency model (RD): In this model, the simulator models limited resources, e.g. entries of the instruction queue, reservation stations, ALU units, decoders and etc. Data and control dependency will not be checked in this model. Cache memory is also not modeled and ideal memory access time (1 cycle delay) is assumed. When the simulator is configured as a RD model, the architects can concentrate on validating the correctness of instructions flowing in the pipeline. It detects possible race conditions due to a
given non-proper design and examines possible mismatching pairs for resource allocation/deallocation. Also, it provides the architects the initial performance metrics and design spectrum for evaluating a given application since maximal instruction-level parallelism can be achieved under this model.

- **Data dependency model (DD):** The DD model is based on the RD model by augmenting data dependence checking during execution instructions. In this model, an architect can focus on techniques employed for resolving data dependency, e.g., reservation stations. It guarantees that an instruction will be stalled unless the operands it required are ready.

- **Control dependency model (CD):** The CD model is based on the DD model and thus it is stricter model. In this model, all control dependency due to branch instructions are considered. It allows an architect to develop and verify the most efficient branch prediction methods. All penalty of miss predictions are considered. A limitation of our approach is that the trace-driven simulation cannot capture the instructions in the execution paths due to miss prediction. Thus it may effect the accuracy of simulation of ICache since these instructions are not available in the ICache.

- **Memory model (MEM):** The MEM model is the strictest model, which is based on the CD model. It details the cache memory sub-system in a microprocessor. The latency to access cache memory and the number of ports to serve a memory operation are defined. One can use the interface provided by the cache memory module and change its internal organization arbitrarily. Thus, an architect can emphasize the design of cache memory even with write buffers.

We show the architectural-level restrictions in Fig. 4. The most inner circular shape denotes the resource dependency model and the most outer shape is for the memory model. Obviously, the resulting performance metrics, such as instructions completed per cycles (IPC), approaches the one measured in an ideal machine under the resource dependency model. When more architectural restrictions are considered, the resulting performance metrics in a more realistic model, e.g. the memory model, should be degraded while comparing with a theoretical model, e.g. the resource dependency model.

4. **Design Space Investigations using Xsim.** In this section, we demonstrate how Xsim was used to evaluate the design of an x86-compatible processor. To characterize the workloads of applications, e.g., MS Windows’ applications, these are shown first. Then, a general organization of the processor is introduced next, which is used to evaluate the design alternatives of processors and sampling techniques. It is followed by the results of our study.
4.1. Characterizing workloads of applications. The characterizations of MS Windows’ applications — word viewer and media play 2 are shown in Table 1. In this experiment, we collect the traces on a machine running Windows NT server 4.0. Further information about top 6 DLL libraries calls are shown in Table 2. We only show the number of instructions executed and the number of basic blocks for these two applications.

From Table 1, we can see that the average number of instructions in a basic block for the two characterized applications are about 4 or 5. For media play 2, the maximal number of instructions in a basic block are about 500, which is twice as many to about 250 in the word viewer. Table 2 shows that both applications invoke ntdll.dll and kernel32.dll libraries.

**Table 1**

Instructions and basic blocks characterized in the two MS Windows’ applications — word viewer and media player 2.

<table>
<thead>
<tr>
<th>Applications</th>
<th>Num. of instruction</th>
<th>Num. of basic blocks</th>
<th>Average num. of instructions/ basic blocks</th>
<th>Maximal num. of instructions/ basic block</th>
<th>Maximal num. of basic blocks executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word viewer</td>
<td>195179435</td>
<td>38136209</td>
<td>5.12</td>
<td>226</td>
<td>780493</td>
</tr>
<tr>
<td>Media Player 2</td>
<td>3022222227</td>
<td>68328872</td>
<td>4.42</td>
<td>488</td>
<td>1098552</td>
</tr>
</tbody>
</table>
438  Hsiao et al.

Table 2

Top 6 DLL libraries used in the (a) word viewer and (b) media player 2.

(a)

<table>
<thead>
<tr>
<th>DLL library</th>
<th>Num. of instructions</th>
<th>Num. of basic blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wordviewer.exe</td>
<td>162396565</td>
<td>33506869</td>
</tr>
<tr>
<td>Ntdll.dll</td>
<td>20819545</td>
<td>1907700</td>
</tr>
<tr>
<td>Mso97v.dll</td>
<td>5815240</td>
<td>1326770</td>
</tr>
<tr>
<td>GDI32.dll</td>
<td>076524</td>
<td>460491</td>
</tr>
<tr>
<td>Kernel32.dll</td>
<td>1290610</td>
<td>324009</td>
</tr>
<tr>
<td>Rpcrt4.dll</td>
<td>937573</td>
<td>197420</td>
</tr>
<tr>
<td>Percentage (%)</td>
<td>99.06</td>
<td>98.92</td>
</tr>
</tbody>
</table>

(b)

<table>
<thead>
<tr>
<th>DLL library</th>
<th>Num. of instructions</th>
<th>Num. of basic blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quartz.dll</td>
<td>100350853</td>
<td>17468725</td>
</tr>
<tr>
<td>Msdxm.dll</td>
<td>78493582</td>
<td>18309271</td>
</tr>
<tr>
<td>User32.dll</td>
<td>72086476</td>
<td>19442623</td>
</tr>
<tr>
<td>Ntdll.dll</td>
<td>23157041</td>
<td>4710337</td>
</tr>
<tr>
<td>Kernel32.dll</td>
<td>12862101</td>
<td>3176121</td>
</tr>
<tr>
<td>Mplayer 2.exe</td>
<td>10322152</td>
<td>4115872</td>
</tr>
<tr>
<td>Percentage (%)</td>
<td>98.36</td>
<td>98.38</td>
</tr>
</tbody>
</table>

Optimizing these libraries may have a significant boost on Windows NT's performance.

4.2. Organization of the target processor. The organization of the target processor is similar to that shown in Fig. 2. The processor is x86-compatible and issues eight RISC-like instructions per cycle. The default numbers of ALU, branch units (BU), and load/store units (LSU) are four, one and two, respectively. The number of OOB entries is 64. Since it is also necessary to preserve the execution sequence of load and store operations, we also modeled a memory out-of-order execution buffer (MOOB) [15] inside the LSUs. The number of MOOB entries is 12. The number of ports in OOB and the register files is eight. For simplicity, we assume that ICache, Dcache, instruction TLB (ITLB), and data TLB are all infinite in size.

The processor has eight pipeline stages: ICache fetch, decode, register renaming, operand access, RS dispatch, execution, write back, and retirement. During the ICache fetch stage, the fetcher first fetches the x86
instructions pointed to by the program counter from the instruction cache (ICache). There is an instruction prefetch queue in the fetcher [30] to store the instructions coexisting in one cache block. While fetching instructions, branch prediction is also performed to determine the content of the program counter for the next cycle.

In the decode stage, the decoder retrieves and decodes an x86 instruction ready in the prefetch queue. Since an x86 instruction is quite complex, the target processor employs a default of eight decoders [4] with varying decoding capability. During the register renaming stage, the resultant RMOs are dispatched to the reservation stations (RS) and are registered into the out-of-order execution buffer (OOB), which is used to ensure in-order completion and help speculative execution. After registration, if the destination operand is a register, then that register will be renamed as the tag of the RMO. In the operand access stage, the registered RMOs perform operand access if any of the source operands is a register. Source operands may be provided by the register file or OOB, and the RMO may receive the source operands’ value or tag. In the RS dispatch stage, each reservation station checks its RMOs to see if they have already obtained their source operands. If so, that RMO is despatched to an available functional unit. As shown in the figure, the reservation stations in the default RISC core are distributed to the functional units.

When the execution of a RMO is completed, its results are written back to the RS and the OOB through the result buses in the write back stage. At the retirement stage, OOB retires RMOs in their original order. However, only when all the RMOs correspond to x86 instructions executed can the x86 instruction be retired. On retirement, the values updated by the x86 instruction are written back to the registers.

Note that we have validate our processor simulator through the four machine models — the RD, DD, CD and MEM models. In this paper we only show the results by employing the CD model.

4.3. Studying processor design. We first study how the simulator can help us to exploit different design parameters. In this set of experiments, we used full traces from integer programs Dhrystone, Hanoi, LU and Quicksort. Although these kernel programs are tiny, they are sufficient to demonstrate our simulation results. The performance metric used for comparison is IPC.

4.3.1. Effects of varying number of entries in OOB. We first study the effects of varying the number of entries in OOB. Table 3 shows

\[\text{Since x86 instructions have variable lengths, an instruction may sit on the boundary of a cache block. A cache miss is thus needed to be issued to the memory bus to fetch the next cache block storing the remaining part of the instruction. Our simulator simulates this operation.}\]
Table 3

Effects of varying the number of entries in OOB.

<table>
<thead>
<tr>
<th>Num. of OOB entries</th>
<th>40</th>
<th>48</th>
<th>56</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dhrystone</td>
<td>2.077</td>
<td>2.093</td>
<td>2.098</td>
<td>2.098</td>
</tr>
<tr>
<td>Hanoi</td>
<td>1.891</td>
<td>1.894</td>
<td>1.894</td>
<td>1.895</td>
</tr>
<tr>
<td>LU</td>
<td>1.665</td>
<td>1.704</td>
<td>1.718</td>
<td>1.720</td>
</tr>
<tr>
<td>Quicksort</td>
<td>2.337</td>
<td>2.388</td>
<td>2.409</td>
<td>2.417</td>
</tr>
</tbody>
</table>

Table 4

Effects of varying the number of ports of register files.

<table>
<thead>
<tr>
<th>Num. of ports of register files</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dhrystone</td>
<td>1.140</td>
<td>2.096</td>
<td>2.096</td>
<td>2.096</td>
</tr>
<tr>
<td>Hanoi</td>
<td>1.070</td>
<td>1.896</td>
<td>1.896</td>
<td>1.896</td>
</tr>
<tr>
<td>LU</td>
<td>0.934</td>
<td>1.720</td>
<td>1.720</td>
<td>1.720</td>
</tr>
<tr>
<td>Quicksort</td>
<td>1.302</td>
<td>2.417</td>
<td>2.417</td>
<td>2.417</td>
</tr>
</tbody>
</table>

the resultant IPC with the number of entries varying from 40 to 64. We can see that the impact of OOB entries is modest. Quicksort has a small drop in IPC when the number of entries is down to 40.

4.3.2. Effects of varying number of ports of register files. We next study the impact of the number of ports available to access the register files. Since x86 architecture has only a few registers, this parameter is quite important to determine how much registers can be accessed concurrently. Table 4 shows the results, where the number of ports varies from 2 to 16. We can see that four ports are sufficient for the processor studied.

4.3.3. Effects of varying number of result buses. Result buses are used to distribute output data from a functional unit to the reservation stations, OOB, and register file. If the number of buses is too small, then although functional units have produced their results, they cannot pass the data out and thus cannot carry out new computations. In Table 5, we see that the resultant IPC is very sensitive to the buses available. More buses would achieve a higher instruction throughput for the programs studied. On the other hand, increasing the number of buses means more chip areas and high complexity in the controlling logic. Thus there is a tradeoff in choosing this parameter.

4.3.4. Effects of varying number of entries in reservation stations. Finally, we study the impact of entries in the reservation stations for
Table 5

Effects of varying the number of resulting buses.

<table>
<thead>
<tr>
<th>Num. of resulting buses</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dhrystone</td>
<td>0.570</td>
<td>1.140</td>
<td>2.101</td>
<td>3.041</td>
</tr>
<tr>
<td>Hanoi</td>
<td>0.535</td>
<td>1.070</td>
<td>1.887</td>
<td>3.127</td>
</tr>
<tr>
<td>LU</td>
<td>0.467</td>
<td>0.934</td>
<td>1.720</td>
<td>2.282</td>
</tr>
<tr>
<td>Quicksort</td>
<td>0.651</td>
<td>1.302</td>
<td>2.417</td>
<td>3.216</td>
</tr>
</tbody>
</table>

Table 6

Effects of varying the number of entries in reservation stations.

<table>
<thead>
<tr>
<th>Num. of entries in reservation stations</th>
<th>6</th>
<th>10</th>
<th>14</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dhrystone</td>
<td>2.007</td>
<td>2.096</td>
<td>2.101</td>
<td>2.101</td>
</tr>
<tr>
<td>Hanoi</td>
<td>1.887</td>
<td>1.887</td>
<td>1.887</td>
<td>1.870</td>
</tr>
<tr>
<td>LU</td>
<td>1.439</td>
<td>1.721</td>
<td>1.721</td>
<td>1.722</td>
</tr>
<tr>
<td>Quicksort</td>
<td>2.171</td>
<td>2.417</td>
<td>2.419</td>
<td>2.424</td>
</tr>
</tbody>
</table>

Table 7

IPC of the programs using trace sampling.

<table>
<thead>
<tr>
<th>Program</th>
<th>Full trace</th>
<th>Sampling (Chunk size = 2000)</th>
<th>Sampling (Chunk size = 5000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compress</td>
<td>0.954061</td>
<td>0.954051</td>
<td>0.954053</td>
</tr>
<tr>
<td>Go</td>
<td>0.933775</td>
<td>0.933495</td>
<td>0.933641</td>
</tr>
<tr>
<td>Perl</td>
<td>0.947305</td>
<td>0.947280</td>
<td>0.947533</td>
</tr>
<tr>
<td>Vortex</td>
<td>0.904558</td>
<td>0.904475</td>
<td>0.904584</td>
</tr>
</tbody>
</table>

ALUs. The number of entries varies from 6 to 18. Table 6 shows that it is sufficient to use 10 entries in the target processor.

4.4. Study on trace sampling. Next we study the effects of trace sampling on the preciseness of the evaluation results. We adopted the multiple periodic sampling technique and used SPECint95 programs: Compress, Go, Perl and Vortex [3]. The scaling factor was set to be 1/10 of the full trace, and the sampling chunk was chosen to be 2000 and 5000 instructions respectively. The primary metric for comparison is instruction per cycle (IPC).

Table 7 lists the IPC of the tested programs with and without trace sampling. From the table we see that for the evaluated programs trace sampling is as accurate as using the full trace, considering only IPC. Note however
that it does not mean that the statistics of other parameters, e.g. resource utilization, branch misprediction rate, cache miss rate, and so on, will be as accurate. Nevertheless, trace sampling does provide a quick way of evaluating processor performance and help to exclude many unplausible options.

Fig. 5. Variation of IPC along time for Dhrystone using random sample and a scaling factor of (a) 1/10000, (b) 1/1000, and (c) 1/100.
We also inspected the variation of IPC over time. For demonstration purpose, we chose a relatively small workload and fed the simulator with two small programs: Dhrystone and Hanoi respectively. In this experiment we used the random sampling technique with a scaling factor of 1/10000, 1/1000 and 1/100 of the full trace. From Figs. 5 and 6 we can see that the random
sampling technique can also produce very accurate results considering IPC only. With a larger scaling factor, the execution converges slower to the final IPC value and thus requires a longer simulation time.

5. Conclusions. In this paper we have introduced a trace-driven simulation tool for evaluating and developing x86-compatible processors. The tool consists of three major components: tracing system, sampling system, and processor simulator. It supports a decoupled decode/encode architecture. We have also demonstrated how it can be used to study the design of a specific x86-compatible processor.

The tool is currently still under development. New features will be added and more efforts will be put to make the tool more stable. Although the trace-driven strategy was chosen early in the development due to its simplicity, we found that it has many limitations. For example, traces only contain instructions from the execution path, but not those fetched, despatched, but discarded later. As a result, it becomes difficult to simulate accurately the behavior of the branch predictor. Another problem with a functional-level simulator is that it does not actually calculate values. Thus many runtime information, e.g. the contents of given memory addresses, are not known and this also hurt the accuracy in the simulation. These are the issues which need to be addressed in the next version of the simulation tool.
REFERENCES


