A Retargetable Micro-architecture Simulator

Wai Sum Mong, Jianwen Zhu
Electrical and Computer Engineering
University of Toronto, Ontario M5S 3G4, Canada
{mong, jzhu}@eecg.toronto.edu

ABSTRACT

The capability of performing architectural exploration has become essential for embedded microprocessor design in System-On-Chip. While many retargetable instruction set (ISA) simulators have been reported, the more relevant micro-architecture simulators, which are capable of modeling the detailed machine features such as cache organization, branch prediction and out-of-order scheduler, have not be equipped with retargetability. In this paper, we propose a new methodology that can generate completed micro-architecture simulators from the abstract ISA and the application binary interface (ABI) specification. We demonstrate our methodology by the development of a tool that can automatically port the SimpleScalar toolset, the *de facto* standard for micro-architecture simulation , to any processor.

Categories and Subject Descriptors

I.6.7 [**Simulation Support Systems**]: Environments; C.0 [**General**]: Modeling of computer architecture

General Terms

Design, Languages

1. INTRODUCTION

Due to the rapid development of the embedded system in the recent decades, many new application-specific processor architectures are developed to meet the required performance and power consumption constraints. To reduce both the design time and the development cost, simulation has been widely used as a means to validate and evaluate the architecture without physically implementing the design at cost and risk.

Instruction set simulation mimics the behavior of each instruction and models the effect on the target processor state at each step. With the instruction set simulator, a new processor architecture design can be functionally verified against any real program. *Micro-architectural simulation*, in contrast, mimics the effect of a micro-architectural design to the instruction execution process. Towards the growing complexity of micro-architectural designs, modeling a *detailed* micro-architectural design in simulation becomes more challenging. Due to its micro-architectural modeling capability and extensibility, the *SimpleScalar toolset* [1] developed at University of Wisconsin emerged as the *de facto* standard of modern micro-architecture simulators. With the rich set of micro-architecture components, such as memory, cache, branch predictor and scheduler, the SimpleScalar toolset can model architectures with varying detail, ranging from the simplest unpipelined processors to the out-of-order superscalar architectures. According to the statistics, more than one half of the papers published in the last Annual International Symposium on Computer Architecture (ISCA 2002) have used SimpleScalar tools to evaluate their designs [1].

Being processor-dependent, simulators are traditionally developed *manually* to support different processor architectures. This approach is no longer efficient enough for modern embedded processors used in the system-on-chips area. Often times, the instruction set architecture (ISA) and the micro-architecture must be designed to adapt to a specific or a family of applications. To find the best solution from the design space, the system architects must perform the so-called architecture exploration, where the software development tools, among which simulators are the most important, have to be developed for each architecture configuration. This necessitates the notion of retargetable simulators, which can be automatically generated from an architecture configuration specification. While retargetable instruction set simulators have been reported extensively in the literature, no retargetable tool have be reported with capability as comprehensive as SimpleScalar. This is becoming a problem for future high-end embedded processors where the abundance of instruction-level parallelism will make the accuracy of instruction set simulation intolerable.

In this paper, we propose techniques that lead to automatic porting the SimpleScalar toolset, which includes a rich, extensible library of micro-architectural modeling components. We make the following contributions. First, we propose an architectural description language (ADL), called *Babel*, to capture not only the ISA information of the processor, but also the ABI information. Note that while ABI is essential, it has not been captured comprehensively by previous ADLs. Second, we enhance the SimpleScalar simulators with additional features such as delayed branches and registered windows. Third, we provided the added value to the popular SimpleScalar toolset with an automatic porting tool. From our own experience, manually porting the tool takes one month of study time of the open-source SimpleScalar infrastructure, which has 30K lines of C code, and one additional month of development time. With our tool and an reusable architectural specification, a retargeted simulator is close to one click away.

The remainder of this paper is organized as follows. In Section 2, we provide an anatomy of the latest SimpleScalar tool set. Sec-

DAC 2003, June 2-6, 2003, Anaheim, CA, USA.

tion 3 presents the detail of our methodology to automatically porting the SimpleScalar for a new processor architecture. We describe the implementation of the automated porting tool in Section 4 and demonstrate experimental results on the SPARC and the PISA architectures. We provide a brief overview of the related works in Section 5.

2. THE SIMPLESCALAR TOOL SET

The SimpleScalar tool [1] set is an infrastructure developed for micro-architectural modeling and simulation. The current release (version 3.0) contains seven simulators at different level of micro-architectural detail as summarized in Figure 1.

Micro-architectural detail

Simulator	Descirption	#line
`sim-fast`	Simple functional simulator	402
`sim-safe`	Speed-optimized functional simulator	307
`sim-profile`	Functional simulator with profiling	812
`sim-cache`	Hierarchical memory simulator	782
`sim-cheetah`	Single-pass multi-configuration cache simulator	479
`sim-bpred`	Customizable branch prediction simulator	513
`sim-outorder`	Detailed micro-architectural simulator with dynamic instruction scheduler and multi-memory hierarchy	4555

Figure 1: SimpleScalar Simulators.

2.1 Infrastructure

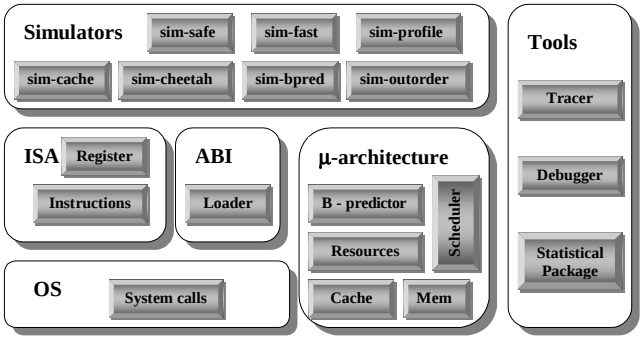

Figure 2: SimpleScalar Infrastructure.

Figure 2 shows the SimpleScalar infrastructure. The behaviors of the simulators are highly dependent on three aspects of the target processor model - ISA, ABI and micro-architecture.

Consisting of the instruction definitions and the register organization, the ISA is the fundamental required knowledge to make a simulator working properly. In SimpleScalar, register files can be modeled with the provided API. The instruction set definition has a pre-defined (or recommended) format; each of which contains the assembly format, binary opcode, execution unit component, register dependency information, instruction class and an enum opcode assigned by the infrastructure. To mimics the instruction effect on the target processor state, each instruction is also associated with a semantic action statement. Trap instructions are however exceptionally handled by the system call simulation package at the underneath OS support.

The SimpleScalar also simulates a *program loader*, which copies the instructions reading from the input executables into the simulated memory. The instruction loading task is dependent on the *binary file format* of the machine code. Generally, the SimpleScalar users use the provided `COFF` file loader although there exists another choice of using the GNU's binary file descriptor (BFD) library. Note that dynamic linking is not supported by SimpleScalar, so the instructions must have been linked and relocated statically before loading.

Micro-architectural components can be modeled through the APIs in SimpleScalar. Until recently, the tool set allows modeling of cache, memory, functional unit resource, scheduler and branch predictor. With its simple design, it is believed that more components can be added to extend the micro-architectural modeling ability of SimpleScalar, and hence serves it playing a better role of a micro-architectural simulator.

2.2 The Simulation Flow

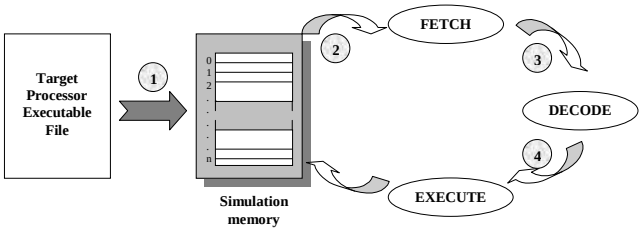

Figure 3: The Simple Simulation Process

The SimpleScalar provides an environment to implement interpretive simulation. We illustrate the simulation flow with the simplest unpipelined functional simulation in Figure 3. **(1)** Before the simulation starts, the instruction and data are loaded to the simulated memory by either the provided `COFF` loader or the `GNU BFD` loader. The program counter (PC) holding at a simulated register is initialized with the program entry point address after loading. The simulator then loops through the *fetch-decode-execute* process at each instruction simulation. **(2)** At fetch, the instruction addressed by the PC register is copied from the simulated memory to the simulated instruction register (IR). **(3)** The fetched instruction is then decoded and it comes up with the SimpleScalar-specific enum opcode of the corresponding instruction type in ISA. **(4)** By using the enum opcode obtained from the decoding process, the correct instruction semantic definition is selected and executed.

2.3 Supported Architectures

Only two target processor architectures are supported in the version 3.0, they are the Portable Instruction Set Architecture (PISA) and the Alpha instruction set architecture.

Designed by the SimpleScalar's authors, PISA is a derivative of the MIPS architecture. In fact, PISA and MIPS only have a little difference in term of their ISAs. Most of the MIPS compilation tools, especially the linker, are reusable by PISA.

Supporting only one real processor architecture (Alpha) until the recent release, the SimpleScalar team has been spending a lot of efforts to manually porting the SimpleScalar infrastructure to many other processor architectures.

3. RETARGETING SIMPLESCALAR

To enhance the ability of SimpleScalar to support modern micro-architectural design under more different ISA and ABI environment, we present an automatic porting system to relieve developers from the complex and timing-consuming porting process.

3.1 System Overview

Figure 4 is the complete design of our SimpleScalar retargeting system. By modeling the target processor architecture with *Babel*, the architectural information is captured as *architectural intellectual property (IP)* and reusable. The *Babel language compiler*

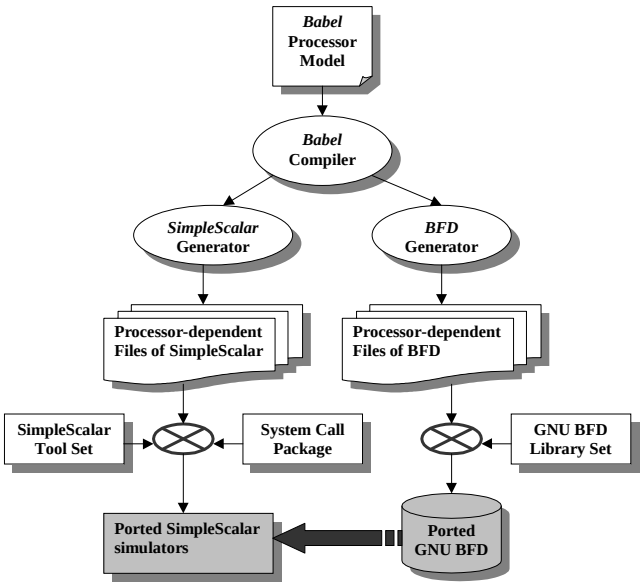

Figure 4: Retargetable SimpleScalar System.

compiles the input IP into a form of internal abstract data. The internal data can be accessed with a set of APIs. By accessing the architectural data captured by the internal data through the APIs, the *SimpleScalar generator* produces the processor-dependent portion of SimpleScalar. Without the simulation of dynamic linking, the processor-dependent portion of SimpleScalar involves little ABI information but mainly consists of the ISA information of the target architecture.

Both of the simulated loader and the instruction decoder of the simulator have access of the binary executable files. We decided to use `GNU BFD` library, the foundation library of another *de facto* binary utility standard, `GNU binutils`. It contains all software downstream development tools, such as assembler and link editor.

While `GNU BFD` and `GNU binutils` are extremely powerful, they are also very complex (1 quarter million lines of code) and therefore difficult to port. We leverage the tool we developed earlier [2] to automatically port *binutils* from the ISA and ABI specification. The API for accessing object code is then readily available.

The *Babel* processor model captures only the architectural description. In the aspect of operating system that supports the target processor, we need a system call simulation package to simulate the behavior of the trap instructions (as shown in the OS part of Figure 2). This is the only component that our system fails to port automatically, and the users will require to port this manually.

With the processor-dependent files generated for the SimpleScalar toolset, the ported `GNU BFD` library generated from our retargetable `BFD` tool, and also the user-provided system call simulation package, integrating them properly in the SimpleScalar infrastructure achieves our goal of retargeting SimpleScalar.

3.2 Architectural Modeling with Babel

To capture processor architectural IPs in *Babel*, we defined models in the views of *ISA*, *ABI* and *MICRO* (micro-architecture). Our SimpleScalar generator mainly use the information from the ISA model while the BFD generator needs both of the ISA model and the ABI model. Modeling micro-architectural features, our MICRO model is flexible and extensible. To focus on retargeting the ISA and the ABI, the MICRO model is not required by the current SimpleScalar generator and nor the BFD generator tool.

Processor	BEH view(#lines)	ISA view(#lines)	ABI view (#lines)
SPARC	196	2300	56
PISA	90	1048	45
Alpha	1144	4511	52

Table 1: Architectural IPs using Babel.

To model processor architecture *myarch* , the following three files written with *Babel* are required:

- *`myarch`*`.bbl`, which captures the behavior view of the architecture. It consists of the data types of registers and complex behavioral semantics definitions on which some instructions in the *myarch*.isa.bbl file will depend.
- *`myarch`*`.isa.bbl`, which captures the ISA view of the architecture. It includes the register file organization and the instruction set definition.
- *`myarch`*`.abi.bbl`, which captures the ABI view of the architecture. The current ABI model gives information about the calling convention in terms of the register usage; and on the other hand the information of linking such as relocation and dynamic linking rules.

Table 1 shows the complexity of *Babel* specification of three different architectures - SPARC, PISA and Alpha. For each architecture, we show the size of each file (as mentioned above) in terms of the number of lines of code.

In this paper, we focus on the ISA modeling part required by the SimpleScalar generator, and a portion of the ABI model that the generator requires will also been briefly mentioned.

3.3 ISA Modeling with Babel

Our ISA model consists of elements of *store*, *field*, *format* and *instruction*. The *store* elements give the data size and organization of the register files, while other elements model instructions.

3.3.1 Register Files

The organization of each register file is given as an element of *store* in the ISA model. As shown in Definition 1 is the *store* model defined in terms of *Babel*. A register file is characterized with `size` number of register, each of which has a bitwidth of `gran`. The other parameters are optional - `depth`, `overlap` and `pointer` models the windowed register organization by specifying the total number of physical registers, the number of overlapping registers between consecutive windows and the current window pointer respectively. At last, `cells` represent the registers that belong to the register file. Example 1 gives a simple model of a register file consisting of 8 32-bit registers.

DEFINITION 1. *A store is defined as*

```
typedef []field    CellGroup;  // a sequence of fields

class Store {
  Store( int gran, int size ); // required properties
                               // optional properties
  int         depth;           //
  int         overlap;         // overlap of windows
  field       pointer;         // window ptr
  {}CellGroup cells;           // regs members
  }
```

EXAMPLE 1. *Modeling a register file with 8 32-bit registers.*

```
  unsigned[32] g0, g1, g2, g3, g4, g5, g6, g7;

  stores = {
    sGPR = new Store( 32, 8 ) {
      maps = {
          gpr = [ g0, g1, g2, g3, g4, g5, g6, g7 ]
          }
      }
    }
```

3.3.2 Instructions

Each instruction of the target architecture is modeled with one *instruction* element in our model. The ISA *instruction* model is defined in Definition 2. An instruction is characterized by its assembly format, binary encoding format, opcodes and behavior.

DEFINITION 2. *An instruction is defined as*

```
class Instrn {
  string          asmFormat;      // assembly format
  Format          binFormat;      // instrn format
  []int           opcodes;        // opcode
  {}method        patterns;       // instrn semantics
  boolean         isDelayed;      // delayed instrn?
  }
```

To reduce the complexity of the model, we define the binary encoding format, `binFormat`, as an sequence of *fields*. A *field* is the smallest unit in the instruction. Each *field* is characterized by its role playing in the instruction, such as an opcode, a register or an immediate value. Definition 3 and Definition 4 give the definitions of the *format* and the *field* respectively.

DEFINITION 3. *An instruction format is defined as*

```
typedef []Field   Format;
```

DEFINITION 4. *A field is defined as*

```
class Field {
  /* opcode field */
  Field( int size );

  /* immediate field */
  Field( int size, boolean isSigned, int argno );

  /* register field */
  Field( int size, boolean isDest, int argno,
         CellGroup group );

  /* relocatable field */
  Field( int size, boolean isSigned, int argno,
         int reloctype );
  }
```

Figure 5 gives an example by demonstrating the modeling process of the SPARC `BNE` instruction with the corresponding *fields* and the binary encoding format. The opcodes of the instructions are not defined in the *fields* nor the *format*. To increase the reusability of the *formats*, the values of all opcode *fields* are defined in the *instruction*. The *patterns* gives the instruction behavior, which can be modeled as simple as using one pre-defined opcode of *Babel*, or using the user-defined method behavior definition, which is specified at `myarch.bbl`.

3.4 ABI Modeling with Babel

Definition 5 defines the ABI model handled by the SimpleScalar generator. Generally speaking, it defines the register usage convention as well as the stack layout. Our ABI also includes other

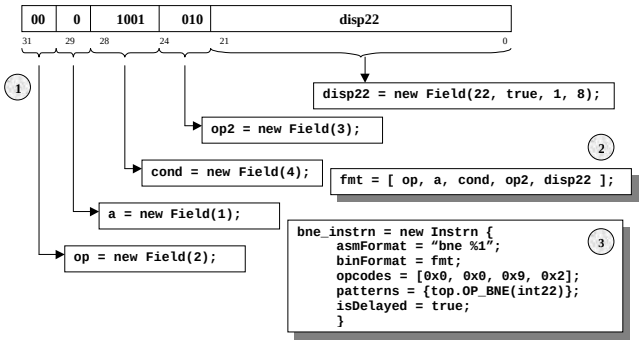

Figure 5: Modeling the SPARC `BNE` instruction.

information, including calling convention, relocation, as well as information related to dynamic linking. Since these information are not used directly by the SimpleScalar generator, we omit the discussion in this paper. However, they are all necessities to automatically port the GNU BFD library [2].

DEFINITION 5. *The ABI model for retargeting SimpleScalar is defined as*

```
class domain.ABI {
  CellGroup       zero;     // zero-valued reg
  CellGroup       itemp;    // local regs
  CellGroup       ftemp;    // local regs for float
  CellGroup       isave;    // regs saved at dispatch
  CellGroup       fsave;    // floating regs saved
  CellGroup       iarg;     // input args
  CellGroup       farg;     // input args for float
  CellGroup       ra;       // return addr
  CellGroup       iret;     // return value
  CellGroup       fret;     // floating return value
  CellGroup       pc;       // program counter
  CellGroup       npc;      // next PC
  CellGroup       sp;       // stack ptr
  CellGroup       fp;       // frame ptr
  CellGroup       gp;       // global ptr
  boolean         isBig;    // is big endian?
  int             spBase;   // stack base addr.
  int             argc;     // offset of [argc] at stack
  }
```

3.5 The SimpleScalar Generator

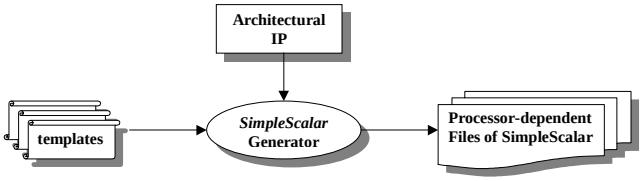

Figure 6: SimpleScalar Generator.

As shown in Figure 6, the SimpleScalar generator tool produces the processor-dependent portion of the code from the architectural IP with the assistance of a set of template files. The template files are created based on the processor-dependent files in the current release of SimpleScalar. Blanking the processor-dependent part, the template files contains only the processor-independent information. The processor-dependent part will be filled by the generator when the compiled architectural IP is available.

Table 2 lists the processor-dependent files of the SimpleScalar toolset. In addition to the files in `target-`*`arch`* directory, the

File	Description
target-*arch*/loader.c	loads program into simulation memory and initializes registers (e.g PC)
target-*arch*/symbol.c	symbol manipulation used by the debugger
target-*arch*/*arch*.h	definitions of register organization and MACROS for instruction manipulation
target-*arch*/*arch*.c	processor-dependent procedures (e.g. *decoder*)
target-*arch*/*arch*.def	instruction definitions
sim-safe.c	simple functional simulator
sim-fast.c	speed-optimized functional simulator
sim-profile.c	functional simulator with profiling
sim-cache.c	cache simulator
sim-cheetah.c	single-pass multi-configuration cache simulator
sim-bpred.c	branch prediction simulator
sim-outorder.c	cycle-accurate OOO-superscalar simulator

Table 2: Processor-dependent files.

File	# lines		
	PISA	rPISA	rSPARC
target-*arch*/loader.c	615	283	284
target-*arch*/arch.h	737	505	473
target-*arch*/arch.c	671	1014	1624
target-*arch*/arch.def	2067	1863	3059
sim-safe.c	307	279	449
sim-cache.c	782	756	910
sim-bpred.c	513	490	658
sim-outorder.c	4555	4451	4634

Table 3: Manually-made vs. Generated Files.

simulator files contain definition of register access interfaces and require porting therefore.

For example, in target-*arch*/*arch*.c, we automatically generate the instruction decoder. In order to achieve this, we first construct a decoding tree by scanning all instructions specified in Babel and analyzing its binary format. Each node in the decoding tree represents a value of a opcode field that can distinguish different instructions. We then emit a tree of C switch statements which mimic the structure of the decoding tree. Note that the generated decoder is much more powerful than the one supplied by SimpleScalar which can only decode instruction with a single opcode field. As another example, we emit C code which can simulate the behavior of each instruction according to the instruction semantics provided by Babel specification.

4. IMPLEMENTATION & EXPERIMENTS

Our retargetable SimpleScalar system (in Figure 4) was tested with the PISA and the SPARC architectures. In so far, our system supports only binary files in the *ELF* format. To make the input architectural IPs, we modeled the tested architectures with *Babel* and their complexities are as shown in Table 1.

Our implemented SimpleScalar generator has successfully generated correct processor-dependent files from the *Babel* architectural models of PISA and SPARC. In this experiment, we demonstrate the feasibility of our idea and the ability of our tool by using 4 ported simulators with different level of micro-architectural detail. Table 3 shows the processor-dependent files generated from our tool (rPISA and rSPARC) and compare them with the original ones provided by SimpleScalar (PISA). To avoid confusion, we named the ported PISA architecture as *rPISA* and the ported SPARC as *rSPARC*. The generated files, which carry the processor-dependent portion of SimpleScalar, will be put into the directory of target-rPISA and the directory of target-rSPARC respectively.

The SimpleScalar simulators generated from our tool are tested with a subset of SPEC2000 testbenches, including 3 integer benchmarks (181.mcf, 197.parser and 164.gzip) and 3 float benchmarks (183.equake, 188.ammp and 179.art). We also compare the performance of the generated simulators with the manually-generated simulators provided from SimpleScalar. All experiments are performed on a 750MHz SunBlade 1000 workstation.

Benchmark	# instructions executed	
	PISA	SPARC
181.mcf	419,091,512	497,727,974
197.parser	46,080,766,461	61,881,001,366
164.gzip	147,714,434,639	204,889,958,252
183.equake	3,277,913,476	2,927,744,817
188.ammp	25,016,922,286	16,563,751,978
179.art	28,986,818,426	36,118,444,032

Table 4: Number of instructions executed.

There are several things required to be done manually before we can activate our simulators. First, the current SimpleScalar toolset has no support of *windowed register* and *delayed branches* in the register library and the branch predictor library respectively. To support SPARC architecture, we manually correct the corresponding libraries to adapt. Second, there is no compilation tools exist to make ELF-formatted executables for PISA. To make the compilation tools, we ported the GNU binutils automatically with our retargetable binutils tool, and GNU gcc and GNU glibc manually. Porting these applications provide us assemblers and linkers, compiler and standard C library respectively. The current release of GNU compilation tools already support SPARC in ELF format on the other hand. At last, as previously mentioned, our system cannot generate the system call simulation package required by the OS part of SimpleScalar. We can re-use the system call simulation package provided by SimpleScalar for *rPISA*. For *rSPARC*, we manually made an equivalent system call package for the experiments.

Our first experiment demonstrates that our tool can successfully perform instruction set simulation. Figure 7 shows the simulation performance in terms of simulated instructions per second for the *rPISA* and *rSPARC*, compared against *PISA*. Table 4 shows the number of instructions executed for each benchmark in terms of different ISA. It can be observed that the performance of our generated instruction simulator is comparable to the original one provided by the SimpleScalar simulator.

We then demonstrate the performance of our retargeted simulator for detailed micro-architecture simulation: Figure 8 for cache simulation and Figure 9 for branch prediction simulation. Again, the performance of our generated simulators are comparable to the original ones. Figure 10 is shown to demonstrate the different simulation speed for different level of architectural detail for the same processor.

5. RELATED WORKS

Several recent research efforts focus on the retargetability issue of the instruction set simulation, where the goal is to generate a simulator automatically from a machine description language. The Insulin simulator [3], translates target machine code into a generic assembly code, which in turn is simulated by a VHDL simulator. In [4] and [5], interpretive and compiled simulators are generated from nML machine description language respectively. Similarly, the JACOB system [6], generates both interpretive and compiled simulators from the MIMOLA HDL.

For micro-architecture simulation, *FastSim* is a micro-architectural simulator optimized for out-of-order processor simulation [7]. Even

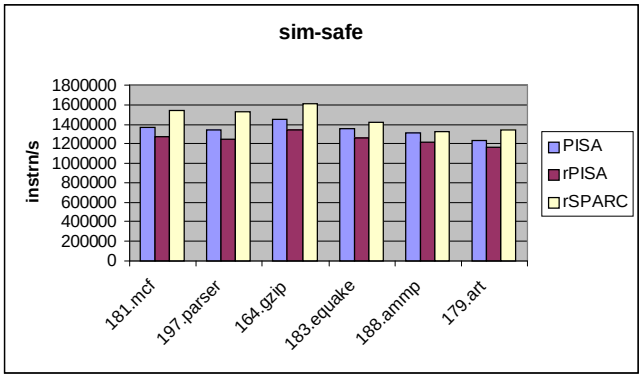

Figure 7: Performance of *sim-safe*.

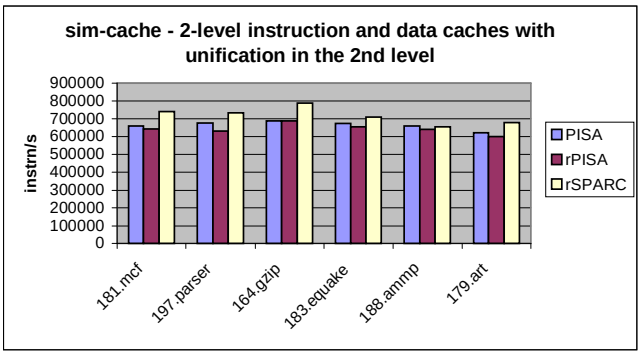

Figure 8: Performance of *sim-cache*.

though the architecture simulated by *FastSim* involves memory and branch predictor, it is not flexible and extensible enough to model new micro-architectural features in the future.

The SimpleScalar [1] toolset is open source distributed for academic purpose. In addition to its simple hence flexible infrastructure design, researchers can use SimpleScalar source code to build new tools and extend it to model more innovative micro-architectural features. For example, *Wattch* is an architecture-level power modeling framework developed on top of the SimpleScalar for power analysis and estimation [8]. The *HydraScalar* Project at the University of Virginia extends SimpleScalar with multipath execution capability [9]. Also, multi-threaded architecture modeling is achieved by enhancing the SimpleScalar infrastructure in the *SIMCA* project at University of Minnesota [10].

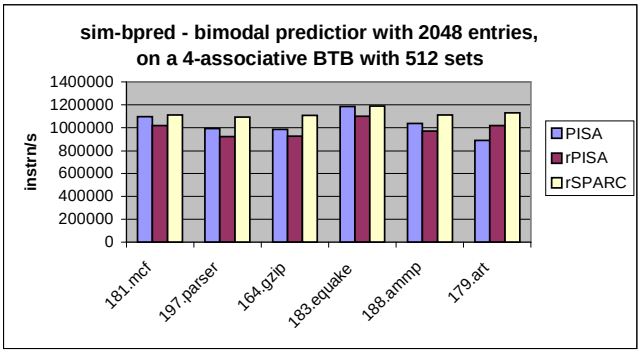

Figure 9: Performance of *sim-cache*.

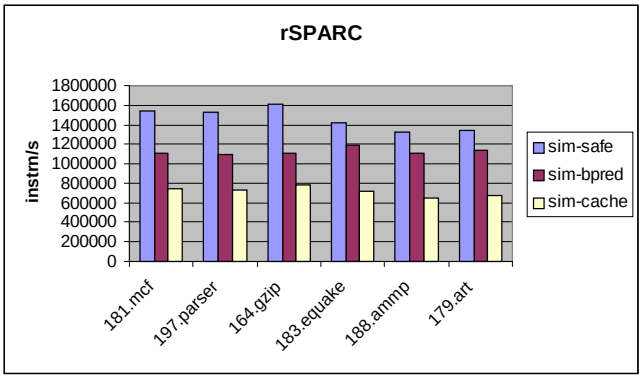

Figure 10: Performance of rSPARC.

6. CONCLUSION

In conclusion, we have argued that micro-architectural simulators are important for embedded processor design and adding retargetability to the micro-architectural simulators is urgent for modern technology. This leads to our work of automating the retargeting process of micro-architectural simulators. For its flexibility and extensibility, we selected the SimpleScalar toolset and enhance its capability with our *SimpleScalar generator*. Our experimental results prove the feasibility of this methodology.

7. REFERENCES

[1] *SimpleSclar LLC*, `http://www.simplescalar.com`.

[2] M. Abbaspour and J. Zhu, "Retargetable binary utilities," New Orleans, USA, June 2002.

[3] S. Sutarwala, P. Paulin, and Y. Kumar, "Insulin: An instruction set simulation environment," in *Proceedings of CHDL-93*, Ottawa, Canada, 1993.

[4] A. Fauth, "Beyond tool-specific machine descriptions," in *Code Generation for Embedded Processors*. 1997, Kluwer Academic Publishers.

[5] M. Hartoog, J. Rowson, P. Reddy, and et al., "Generation of software tools from processor descriptions for hardware/software codesign," in *Proceeding of the 34th Design Automation Conference*, 1997.

[6] R. Leupers, J. Elste, and B. Landwehr, "Generation of interpretive and compiled instruction set simulators," in *Proceeding of Asian-Pacific Design Automation Conference*, Hong Kong, January 1999.

[7] E. Schnarr and J. Larus, "Fast out-of-order processor simulation using memorization," in *The Eighth International Conference on Architectural Support for Programming Lanuages and Operating Systems (ASPLOS-VIII)*, San Jose, California, October 1998.

[8] V. Tiwari D. Brooks and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *The Proceedings of the 27th International Symposium on Computer Architecture*, Vancouver, BC, June 2000.

[9] K. Skadron and P. S. Ahuja, "Hydrascalar: A multipath-capable simulator," in *Newsletter of the IEEE Technical Committee on Computer Architecture*, Jan 2001.

[10] *SIMCA, the Simulator for the Superthreaded Architecture*, `http://www-mount.ee.umn.edu/~lilja/SIMCA`.