An Ultra-fast Instruction Set Simulator

Jianwen Zhu, *Member, IEEE*, Daniel D. Gajski, *Fellow, IEEE*

Abstract—

In this paper, we present new techniques which further improve the static compilation-based instruction set architecture (ISA) simulation by the aggressive utilization of the host machine resources. Such utilization is achieved by defining a low level code generation interface specialized for ISA simulation, rather than the traditional approaches which use C as a code generation interface. We are able to perform the simulation at a speed of up to 10^2 millions of simulated instructions per second (MIPS) on a 270 MHz Ultra-5 workstation. This result is only on average 1.6 times slower than the native execution on the host machine, the fastest to the best of our knowledge.

Keywords—**Computing, High-performance, Logic-simulation, System-level**

I. Introduction

An *instruction set simulator* is a tool that runs on a workstation, called the *host machine*, to mimic the behavior of, or *simulate* a program running on another machine, called the *target machine*, which either does not yet exist, or is not available. Typically, instruction set simulation allows the user to examine the internal state of the target machine, such as the values of processor registers, during the execution of each instruction.

Instruction set simulators are indispensable tools in the development of conventional computer systems. They help to *validate* the processor design, the compiler design, as well as *evaluate* architectural design decisions such as cache sizes. Instruction set simulators play an even more important role in the development of modern *embedded systems*, which typically integrate one or more processors, acceleration hardware, and sometimes analog front-ends, on one chip to implement one specific application, such as cellular phone and personal communication systems. *Hardware/software co-simulation* [1], of which instruction set simulation is one of the most important parts, must be performed in order to validate and evaluate not only architectural decisions, but also implementation decisions such as how the functionality of the application is partitioned into hardware and software before any such systems are built. Such capability of *virtual prototyping* is essential to the success of a product.

It is obvious that the most important quality metric of an ISA simulator is its *simulation speed*, which is especially relevant to the development of high performance systems, where being able to perform simulation in *real time* is desired. Hardware *emulation*, despite its cost, has to be used when real time simulation is impossible. Other quality metrics include *compilation speed*, which has to do with how fast the simulator can bring an application into a simulatable state; *traceability*, which has to do with how flexible the simulator can collect useful statistics such as instruction profiling; *portability*, which has to do with how easy the tool can be ported to new platforms; *retargetability*, which has to do with how easy the tool can be extended to handle new target machines; *interoperatability*, which has to do with its capability to integrate with other tools such as debuggers, hardware simulators, etc.

Due to their importance, numerous ISA simulators have been developed, which can be categorized into three types (Section II), namely, *interpretation-based*, *static compilation-based* and *dynamic compilation-based*.

In this paper, we present the design of a simulator for pure ISA simulation. Our simulator, however, does not handle complete machine simulation including additional peripheral devices such as co-processors and UARTs at the current stage. Our tool falls into the category of static compilation-based simulators. In addition to the advantages inherited, our tool makes several contributions, which lead to its superior performance. First, we propose to use a RISC like *virtual machine*, which has a predefined instruction set and an unlimited number of virtual registers, to serve as the intermediate form to which the target instructions get translated, and from which the host instructions are generated. This is in contrast to the dynamic compilation-based approaches which usually directly emit host instructions, where portability has to be sacrificed; and the traditional static approaches which emit C, where the direct manipulation of host machine resources is impossible.

Second, we use an aggressive, yet extremely simple *register allocator*, which is tailored for the purpose of ISA simulation. Effectively, this allows the direct mapping of target machine registers to host machine registers, while retaining portability. Such effect is hard, if not impossible to achieve in the traditional C-emitting approach, even when sophisticated optimizations are used.

In addition, the proposed low level interface allows us to bypass the host machine calling conventions (Section IV-F), which effectively expose more registers for the register allocator to manipulate on host machine architectures with register windows, such as SPARC. In combination, we are able to simulate the benchmarks only 1.1-2.5 times slower than the execution of their counterparts directly compiled on the host machine when tracing is off. This result is on average 2 times faster than that can be achieved by the approaches used in state-of-the-art tools [2] [3] [4].

The remainder of this paper is organized as follows. Section II gives more detailed description on the various approaches and compares their trade-offs. Section III and Section IV present the details of our simulator. Section V

Jianwen Zhu is with Electrical and Computer Engineering, University of Toronto, 10 King's College Road, Toronto, Ontaria M5S 3G4, Canada.

Daniel D. Gajski is with Information and Computer Science, University of California, Irvine, CA 92717-3425, USA.

discusses its extensions and limitations. Section VI describes the experiment setup and gives an analysis of our experimental results on the chosen benchmarks. We would also like to acknowledge that the preliminary results of this work were presented at [5].

II. Related Works

A. Interpretation-Based Simulation

Interpretation-based simulation builds in memory a data structure representing the state of the target processor. It then enters a loop, the body of which executes the sequence of actions as shown in Figure 1: *fetch*, which reads an instruction word from memory; *decode*, which analyzes the instruction and extracts the opcode field of the instruction; *dispatch*, which uses a switch statement to jump to the appropriate code to handle a particular instruction; and *execute*, which updates the processor state according to the semantics of the instruction.

```
for( ; ; ) {
    instruction = fetch( pc );
    opcode = decode( instruction );
    switch( opcode ) {
        ....
    case ADD    :
        ....
        break;
        }
    }
```

Fig. 1. Simulation loop of interpretative simulator.

A representative, widely used interpretation-based simulator for the MIPS processor is described in [6]. Almost all commercially available simulators are interpretative. Despite ease of implementation and flexibility, interpretation-based simulators suffer performance problems, mainly due to the tremendous overhead spent on instruction fetching, decoding and dispatching, which, from the simulation point of view, is *unproductive*. The simulator [6] reports a 25 times slowdown of the native execution. [4] reported that it takes DSP simulators provided by vendors 6.4 hours to simulate G.726 speech transcoder for 13 seconds of speech signals, in contrast to the 7 seconds of native execution time.

B. Compilation-Based Simulation

Compilation-based approaches reduce the runtime overhead by translating each target machine instruction directly to a series of host machine instructions which manipulate the simulated machine state. Typically, the simulated machine state is maintained in a global memory space of the host machine. For example, the MIPS code in Example 1 is translated to the SPARC code in Example 2 for simulation. Here, `sp__sim` is the memory location which holds the value of the simulated `sp` register.

Example 1: Target code.

```
addu $sp,$sp,-80
```

□

Example 2: Simulation code.

```
sethi %hi(sp__sim), %l0
ld [%lo(sp__sim)+%l0], %l1
add %l1, -80, %l2
sethi %hi(sp__sim), %l3
st [%lo(sp__sim)+%l3], %l2
```

□

Such translation can be done either at compile time, as in the case of statically compiled simulation, where the translation overhead is completely eliminated; or at load time, as in the case of dynamically compiled simulation, where the overhead is amortized over the loops which repeatedly execute the same code.

Static compilation-based simulation, as shown in Figure 2, usually translates the target program into C code, and then uses an optimizing C compiler (e.g., gcc with option -O3) to translate the C code into host machine instructions. In [4], such simulators are developed for DSP processors. The authors reported a 200-640 faster speed than the corresponding interpretative simulator. However, the simulation speed still ranges from 0.8 MIPS to 2.5 MIPS, which we believe is slower than [3] and [2] for the following reasons: First, simulation of DSP instructions is usually more complex than RISC instructions, especially when bit true simulation is required. Second, when the input is a binary executable in which the symbolic information is missing, the simulator has to assume that every instruction is a target for branching. The resultant C code is very difficult for the compiler to optimize.

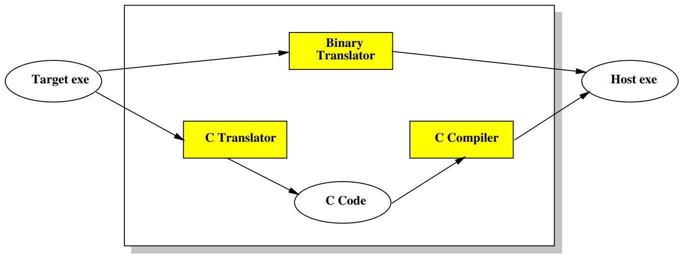

Fig. 2. Static compilation-based simulator.

Dynamic compilation-based simulation, as shown in Figure 3, translates the target program into host machine code on the fly. More specifically, chunks of translated host machine codes, called *translations*, are kept in the so-called *translation cache* (TC), which is in turn addressed by the *translation lookaside buffer* (TLB). The translation usually consists of a prologue, which typically consists of instructions that load simulated target machine state from the memory into the host machine registers; the body, which manipulates the target machine state in the host machine registers; and the epilogue, which dumps the content of the host machine registers back to the memory. The simulator proceeds by first looking up the TLB with the current target program counter value. If there is a hit, that is, the corresponding translation has been performed before, the TLB will return a host machine address in the translation cache to which the simulator can jump immediately.

Otherwise, a chunk of target machine instructions starting from the target PC address will be translated and the TLB and TC are updated accordingly.

The dynamic compilation-based approach is pioneered by the shade simulator [3], where the SPARC V8, V9 and MIPS instruction set can be simulated within 3-10 times native time. Inspired by [3], the Embra simulator [2] performs complete machine simulation with similar performance.

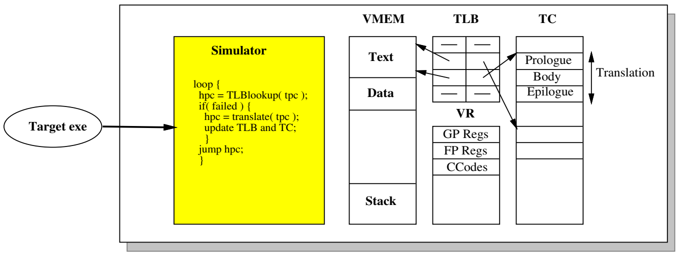

Fig. 3. Dynamic compilation-based simulator.

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 [7], translates target machine code into a generic assembly code, which in turn is simulated by a VHDL simulator. In [8] and [9], interpretive and compiled simulators are generated from nML machine description language respectively. Similarly, the JACOB system [10], generates both interpretive and compiled simulators from the MIMOLA HDL.

To compare these efforts with ours, [4], [9], [10] ignore register allocation and leave everything to the C compiler. [3], [2] do limited register allocation within the boundary of so called *translation*, a unit of code which can be roughly considered a basic block. In addition, portability issues are not addressed. In contrast, our approach allows the register allocation spanning the *entire* target program, which provides the additional performance improvement that will be illustrated later. In this paper, we focus mainly on the performance issue of instruction set simulation. The retargetability issue, however, is not addressed, although we cannot envision fundamental reasons that can prevent us from combining our techniques with those in [7], [8], [9], [10]. We will conduct detailed study on traceability and cycle accuracy in separate works.

The techniques discussed in this paper are not limited to embedded system design. It is also closely related to the field of binary translation [11], [12], [13], [14], [15], which promises to emulate the software of one platform, for example, a Microsoft Windows application, on another platform, for example, a Sun workstation. It is obvious that our technique can be used for the purpose of binary translation. However, we would like to point out that the reverse is not true. The reason is that while binary translation only needs the translated executable to produce the same result as the original, the simulation also needs to correctly maintain the target machine state at every simulated machine cycle. Given such freedom, binary translation can potentially achieve better performance than compilation-based simulation. One extreme case is to reverse engineer the control-dataflow graph from the target machine code, and aggressive compiler optimization can then be applied to obtain a program which can be potentially faster than the original —- this certainly cannot be the case for instruction set simulation.

III. A New Approach

As shown in Figure 4, our simulator looks like, and in fact is integrated into, a *retargetable compiler*. A retargetable compiler is able to cross compile a source program into binary code for a number of targets. Typically, it has a *backend code generator* for each target it is able to support. Our tool adds a corresponding number of backends for simulation purpose. Instead of generating the target code, our simulation backends generate host machine code to simulate target code instead. As illustrated in Figure 4, for each target, a software component called the *target translator* is responsible for emitting a series of virtual machine instructions through a *simulation code generation API*, for each target machine instruction to be simulated. For example, in Figure 4, the *MIPS target translator* translates MIPS machine code into virtual machine code. The abstract simulation code generation API is in turn implemented by a *host translator*, which translates each virtual machine instruction into a form that can be converted into host machine instructions. There can be many host translator implementations depending on what platform the simulation is to be performed. For example, in Figure 4, the *host SPARC translator* translates virtual machine instructions directly into SPARC instructions. The *C translator* translates virtual machine instructions into C code, which can be compiled into any host machine code using a standard C compiler. In order to achieve good simulation performance, the host translators might need to manage the mapping between virtual machine registers to host machine registers via the *register allocation API*, which can be implemented by a *register allocator* using an algorithm independent of any host machine platforms.

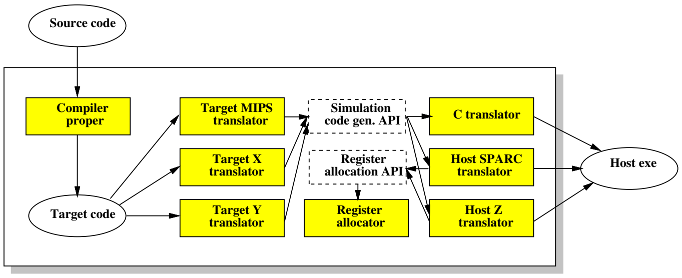

Fig. 4. Our simulator.

A. Simulation Code Generation API

Figure 5 defines the API that every host translator has to implement. The API essentially provides the routines to emit code of a virtual machine, whose instruction set

```
public enum SegKind {
  SEG_CODE = 1, SEG_BSS, SEG_DATA, SEG_LIT
  }

public interface IHost {
  void begin();
  void end();
  void beginFunction( String name );
  void endFunction( String name );
  void segment( SegKind seg );

  void exportSymbol( String symbol );
  void importSymbol( String name, int size );
  void declSymbol( String name, boolean isstatic );

  void emitConstantValue( Type type,
    Object value );
  void emitAddressValue( String name );
  void emitStringValue( int n, String name );
  void emitSpace( int size );
  void emitAlign( int align );
  void emitInstrn(
    Opcode opcode, Type type,
    TargetExpr dest,
    TargetExpr op1, TargetExpr op2
    );

  int declGlobal( String name );
  int declLocal();
  void undeclAllLocals();

  void    emitFetch( IRegAlloc ra, int vreg );
  void    emitFlush( IRegAlloc ra, int vreg );
  }
```

Fig. 5. Simulation code generation API.

```
public enum Opcode {
  OP_ADD, OP_SUB, OP_MUL, OP_DIV, OP_MOD,
  OP_AND, OP_OR, OP_XOR, OP_SHL, OP_SHR,
  OP_COMP, OP_NOT, OP_NEG, OP_MOV, OP_SET,
  OP_CNVI, OP_CNVU, ...,
  OP_LD, OP_ST,
  OP_RET, OP_J, OP_JAL,
  OP_BLT, OP_BLE, OP_BGT, OP_BGE,
  OP_BEQ, OP_BNE,
  OP_NOP
  }

public enum Type {
  TYPE_C, TYPE_UC, TYPE_S, TYPE_US,
  TYPE_I, TYPE_U, TYPE_L, TYPE_UL,
  TYPE_F, TYPE_D, TYPE_P, TYPE_V,
  }
```

Fig. 6. Virtual machine opcode and data types.

resembles that of a RISC machine, and contains an unlimited number of virtual registers. Hence in many ways, the API looks like one that helps to emit assembly code. Conceptually, the API is a procedural interface to help emit either data or instructions into different *memory segments* at the current *program location* of the current virtual machine code *module*.

The virtual machine that we define has an instruction set that resembles [16], which in turn is derived from the intermediate representation of [17]. Each instruction is represented as a value tuple of opcode, type, destination and operands. The opcodes include arithmetic/logical operations, type conversion operations, load/store operations and control transfer operations. The types further constrain the operations to work on a byte (signed or unsigned), halfword, word, long, single and double precision floating point and pointer value. They are defined in Figure 6.

The operands can be either an immediate value, that is, a constant, a symbol, an expression which manipulates constants and symbols, or a virtual register. A symbol is nothing but a symbolic name for address. The destination is always a virtual register.

To briefly describe the API, `begin` and `end` give the host translator an opportunity to initialize and finalize its internal data structure to emit code for a module. Likewise, `beginFunction` and `endFunction` signal the beginning and the end of a function. `segment` announces that the target translator will start emitting either instructions in the text segment `SEG_CODE`, or uninitialized data in segment `SEG_BSS`, or initialized data in segment `SEG_DATA`, or constant data in segment `SEG_LIT`.

The next few functions manage symbols. A symbol can be either a function name, a global variable name, or a label to which control flow can merge. Function `exportSymbol` announces that the provided symbol can be exported to other modules. Function `importSymbol` announces that the provided symbol should be imported from other modules. Function `declSymbol` declares a symbol in the current module.

The next few functions emit code. Function `emitConstantValue` emits constant numerical data in segment `SEG_LIT`. Function `emitAddressValue` emits address value, typically simply using a symbolic name, in segment `SEG_DATA`. Function `emitSpace` emits uninitialized data in segment `SEG_BSS`. The function `emitAlign` emits space so that the current program location is aligned to word or double word boundary. Function `emitInstrn` emits a virtual instruction given the opcode, type, destination and operands.

Our virtual machine has an unlimited number of virtual registers. The virtual registers are categorized into *global* registers, which are alive during the entire program execution; and *local* registers, whose values only last a short time, typically one simulated instruction. For ease of presentation, here and in the text afterwards we do not distinguish an integer register and a floating-point register. Function `declGlobal` can be used to allocate a global virtual register. Function `declLocal` can be used to allocate a local virtual register. Function `undeclAllLocals` can be used to release all the local virtual registers. In addition, `emitFetch` and `emitFlush` are helper functions for the register allocator.

The improvement of portability of our simulator over dynamic compilation-based simulators attributes to the fact that the host translators are completely decoupled from the target translators thanks to the code generation API. Therefore if we have M targets and N hosts, we need only to implement $M+N$ software components, namely M target translators and N host translators. In fact, if we always use the *C translator* in Figure 4, we can reduce the number to $M + 1$. This is in contrast to approaches used by [3], [2], where $M * N$ components have to be implemented. We envision that if our simulator is to be extended with the similar capability of retargetability as in [7], [8], [9], [10], we can specify the instruction semantics of the target machine in the architectural description language in terms of virtual instructions. This is needed anyway to help implement the instruction selector of the compiler. But now the same information can be used to automate the generation of target translators.

B. Target Translator

A target translator uses the code generation API to emit simulation code. It first allocates a set of global virtual registers to simulate the target machine state. Typically, they correspond to the target machine registers. It then emits a set of virtual instructions for every target instruction, while making sure that they have the same semantics. Note that usually one virtual instruction is enough for a target instruction. Otherwise, local virtual registers have to be allocated for temporary storage. At the end of a simulated target instruction, all the allocated virtual registers should be released. For example, the MIPS instruction in Example 1 is mapped to the virtual instruction `add_i vsp, -80, vsp`, where `vsp` is a virtual register allocated for the target `sp` register. The target translator calls other interface functions to emit data and other assembly directives.

C. Register Allocation API

Most virtual instructions apply certain operations on some source virtual registers and write the result to the destination virtual registers. Each virtual register has a memory location in the simulation code to hold its value. For efficiency, the virtual registers should be cached in the host machine registers, called the *hard* registers. The policy towards how the virtual registers are cached comprises the job of the register allocator.

In Figure 7, we define that a hard register can be marked as free (`REG_FREE`), which means that it can be allocated to any virtual register; and fixed (`REG_FIXED`), which means that once it is allocated to a virtual register, the binding remains permanently; and spillable (`REG_SPILLABLE`), which means that its content can be flushed to the memory and thereby be reallocated to another virtual register; and dirty (`REG_DIRTY`), which means that it has been written after its content was fetched from the memory.

The API includes functions `start` and `end`, which initializes and finalizes the register allocator internal data structure respectively. Function `declHard` declares a hard register with its initial marking. It should always be marked as free, unspillable, not dirty, and either fixed or unfixed. Functions `declGlobal`, `declLocal` and `undeclAllLocals` implement the corresponding simulation code generation API functions. Function `getName` returns the name of a virtual register given its integer identifier. Function `ask` performs the mapping of a given virtual register `vreg` to the hard register and returns its name as the result. It will emit fetching and flushing instructions as needed. Function

```
public enum RegMark {
    REG_FREE        =   0x01,
    REG_FIXED       =   0x02,
    REG_SPILLABLE   =   0x04,
    REG_DIRTY       =   0x08
    }

public interface IRegAlloc {
    void    start();
    void    end();
    void    declHard( String name, RegMark mark );
    int     declGlobal( String name );
    int     declLocal();
    void    undeclAllLocals();
    String  getName( int vreg );
    String  ask( IHost host, int vreg, boolean isFetch );
    void    kill( int r );
    }
```

Fig. 7. Register allocation API.

`kill` releases the binding between a virtual register and a hard register.

D. Host Translator

A host translator implements the API defined in Section III-A. The majority of the development effort is usually devoted to the implementation of every virtual instruction using host machine instructions. For example, to emit the virtual instruction `add_i dest, src1, src2`, the *host SPARC translator* in Figure 4 will execute the following code sequence, assuming `ra` is the register allocator:

Example 3: VM instruction implementation for SPARC

```
    ...
    String nmsrc1 = ra.ask( src1, true );
    String nmsrc2 = ra.ask( src2, true );
    String nmdest = ra.ask( dest, false );
    System.out.println(
        "add" + nmsrc1 + ", "
        + nmsrc2 + ", " + nmdest
        );
    ra.kill( src1 );
    ra.kill( src2 );
    ra.kill( dest );
    ...
```

□

As mentioned earlier, the allocation of virtual registers is delegated to the register allocation API in Figure III-C. The data emission and other bookkeeping tasks, such as symbol management, are trivial.

SPARC implementation of the `emitFetch` and `emitFlush` is shown in Example 4.

Example 4: Fetching and flushing implementation for SPARC

```
    ...
    void    emitFetch( IRegAlloc ra, int vreg ) {
        String  name = ra.getName( vreg );
        String  hard = ra.ask( vreg, 0 );
        System.out.println( "set" + name + ", %g1" );
        System.out.println( "ld [%g1], %" + hard );
        }
    void    emitFlush( IRegAlloc ra, int vreg ) {
        String  name = ra.getName( vreg );
        String  hard = ra.ask( vreg, 0 );
        System.out.println( "set " + name + ", %g1" );
        System.out.println( "st " + hard + ", [%g1]" );
```

```
        }
    ...
```

□

IV. Implementation Strategies

Up to this point, we have introduced the software architecture of our tool. In this section, we will focus on the register allocation algorithm and discuss various implementation strategies.

A. Register Allocation Algorithm

Figure 8 shows our implementation of the register allocation API defined in Figure 7. Note that:

- In function `declGlobal`, the allocated global virtual register is associated with a hard register whenever one marked as fixed is available.
- Function `ask` will emit nothing if the virtual register is already assigned a hard register; otherwise it will call `alloc` to find a free hard register. In case of failure, it will call `spill`, thereby select a virtual register to give up its occupancy of the corresponding hard register, by first flushing its value if it is marked as "dirty" or its value is inconsistent with that stored in the memory. Once it gets a hard register, it will emit fetching instructions as needed.
- Function `alloc` has a complexity of $O(n)$, where n is the number of hard registers.
- Function `spill` has a complexity of $O(m)$, where m is the number of declared virtual registers.

B. Greedy Allocation

Equipped with the register allocator, the host translator can employ different strategies to manage the mapping of virtual registers to hard registers. A straightforward strategy for implementing a virtual instruction would fetch the source virtual register values from the memory to the hard registers, compute, and then store the result immediately back to the memory. An example of such a strategy is shown in Example 2.

To implement this strategy, the host translator will first add all hard registers with a marking of unfixed. For each virtual instruction, after emitting host instructions (Example 3), it will have to call `emitFlush` to flush all the virtual registers.

The obvious overhead of this strategy is the read of operands of the instruction from memory and the write of destination of the instruction to the memory. Each read costs one cycle, for the best case of cache hit. Each write costs at least one cycle too, regardless of the fact that the host machine might have a write-through or write-back cache policy.

C. Lazy Allocation

A better policy is to perform lazy fetching, that is, virtual register values need not be loaded from the memory if they have not recently been written after the last read from the same basic block; and lazy flushing, that is, virtual registers need not be written to the memory until the end of a basic block. Here, the basic block refers to a piece

```
public class RegAlloc implements IRegAlloc {
    void declHard( String name, RegMark mark ) {
        add a hard register with its name and mark;
        }
    int declGlobal( String name ) {
        int rtn = add a global virtual register;
        record name for rtn;

        forall hard registers hreg {
            if( hreg is marked as free and fixed ) {
                unmark REG_FREE of hreg;
                bind hreg to rtn;
                break;
                }
            }
        return rtn;
        }
    int declLocal( String name ) {
        int rtn = add a local virtual register;
        record name for rtn; return rtn;
        }
    void undeclAllLocals() { delete all locals; }
    String  getName( int vreg ) {
        return the recorded name of vreg;
        }
    String ask( IHost host, int vreg, boolean isFetch ) {
        hreg = hard register assigned to vreg;
        if( hreg != NULL ) {
            if( !isFetch )
                mark REG_DIRTY of hreg;
            return name of hreg;
            }
        rtn = alloc( vreg );
        if( rtn == NULL ) {
            spill( host );
            rtn = alloc( vreg );
            }
        if( vreg is global ) {
            if( isFetch )
                host.emitFetch( host, vreg );
            else
                mark REG_DIRTY of assigned hard register;
            }
        return rtn;
        }
    void kill( int vreg ) {
        preg = hard register assigned to vreg;
        mark REG_SPILLABLE of preg;
        }
    private String alloc( int vreg ) {
        forall hard registers hreg  {
            if( hreg is free ) {
                unmark REG_FREE of hreg;
                assign hreg to vreg;
                return name of hreg;
                }
            }
        }
    private void spill( IHost host ) {
        forall virtual registers vreg {
            hreg = hard register assigned to vreg;
            if( hreg != NULL && hreg is spillable ) {
                host.emitFlush( host, vreg );
                unmark REG_DIRTY of hreg;
                mark REG_FREE of hreg;
                return;
                }
            }
        }
}
```

Fig. 8. Register allocation algorithm.

of code which contains a single entry and does not contain control transfer instructions except the last one.

To implement this strategy, the host translator will first add all hard registers with a marking of unfixed. It will flush all the virtual registers at the end of a basic block.

The overhead of lazy allocation lies in the fetching code for the first use of virtual registers in the basic block, the spilling code which flushes virtual register, and an epilogue which flushes all the "dirty" virtual registers, for every basic block. This overhead is needed because the mapping between virtual registers and hard registers are different across different basic blocks.

D. Fixed Allocation

An observation is that if the mapping is consistent across the entire program, then these overheads can be eliminated. This is of course not always feasible since there might not be enough hard registers to hold all the virtual registers. But still, some virtual registers, such as those which correspond to the stack pointer, program counter, and target scratch registers, are so frequently used that they deserve to have one fixed hard register allocated whenever possible.

E. Hybrid Approach

This leads to a hybrid approach in which the hard registers are partitioned into two sets: one is the *fixed* register set, the member of which is assigned to a global virtual register throughout the entire program execution; the other is the *temporary* register set.

This strategy is adopted by our simulator, where a global virtual register is assigned a fixed hard register on a first-come-first-served basis. Those globals that fail to obtain a fixed hard register are mapped to the temporary registers together with the locals according to the lazy allocation mechanism.

To implement this strategy, the host translator will first add all fixed hard registers with a fixed marking, and all temporary hard registers with an unfixed marking. There is no need to explicitly flush any virtual register.

Note that our algorithm is of linear complexity in terms of number of virtual or hard registers. This is in contrast to standard approaches based on liveness analysis and graph coloring, which is (1) an overkill for allocation of locals since their lifetimes only last one simulated instruction; (2) unable to handle globals like ours without expensive inter-procedural analysis and execution profiling.

F. Calling Convention Bypass

One might argue that the high level C code generation interface can still be used, since some compiler-specific extensions of C are able to direct the compiler to map global variables to hard registers. In fact, the popular gcc compiler can accept statement `register int sp__sim asm( "%g4" )` to map global variable `sp__sim` to hard register `g4`.

This approach is certainly not portable. Furthermore, there is one fundamental reason that this proposal is not feasible.

One important family of host workstations, namely the Sun machines, use the SPARC architecture [18], which contains register windows to reduce the cost of function calls. If a standard C compiler is used, the compiler will generate code to shift the register window whenever a procedure is called. This essentially causes most hard registers to become physically different hard registers residing in a different register window. Hence these registers cannot be partitioned into the fixed register set. Thus, on SPARC, only `g4` through `g7` are available, and the performance improvement is greatly reduced.

On the other hand, by using a low level code generation interface, our approach can bypass the standard calling convention by suppressing the instructions for register window shifting. Therefore, almost all the hard registers are available for us to enable an efficient register mapping.

V. Extensions and Limitations

We have presented a "bare" simulator whose only utility is to *run* the simulated application. However, it can be extended to meet other requirements.

A. Tracing and Profiling

It is sometimes helpful to collect tracing information during the simulation. For example, the number of total instructions executed. This can be easily achieved by allocating a global `itotal` and emitting the virtual instruction

```
add_i, itotal, 1, itotal
```

before every simulated instruction.

Similarly, the number of executions of every type of target instruction can be kept track of by allocating a global `icount`, which points to the beginning of a table, and emitting virtual instructions

```
ld_i, [icount+offset], tmp
add_i, tmp, 1, tmp
st_i, [icount+offset], tmp
```

before each simulated instruction. Here `offset` is the offset into the table where the tracing information is stored, and `tmp` is a local.

It is also possible to emit instructions to call a user defined routine. For example, whenever a load or store instruction is encountered, a user provided cache simulation routine is called. Note that in order to achieve this, extra care has to be exercised so that the user routine, which uses the host machine's calling convention, does not corrupt the data maintained in the hard registers. We address this problem and study the effect of tracing in a separate study.

Note that tracing will inevitably slow down the simulation performance. But being able to directly map frequently used variables such as `itotal` and `icount` is certainly helpful.

B. Cycle-True Simulation

When cycle-true simulation is required, the program state includes the values of not only all registers, but also the registers between the pipeline stages. It is easy to see that our technique can be very useful to map the frequently used pipeline registers directly into host machine registers. However, the existence of branches, especially indirect branches, complicates the static compilation-based simulation. This problem has been pointed out by [4]. We address this issue in a separate study.

C. Source Level Debugging

Support for source level debugging can be achieved by simply enhancing the code generation interface presented with functions that emit debug information, such as `stabline`, which emits source line number information, `stabsym`, which emits symbol information, and `stabtype`, which emits type information.

D. Limitations

There are limitations to the static compiled approach in general. Simulators that fall into this category cannot handle self-modifying code and code with dynamically linked libraries. Our tool is not immune to these limitations. Fortunately, these cases are rare in embedded systems.

There are also limitations specific to our tool. First, our tool works best on high performance host machines with large register sets. When the host has a limited number of registers, the performance will degrade, however, not to the level worse than those without register allocation. Second, the difference on byte order assumed by the target machine and the host machine is ignored. Third, currently the code generation from target machine to virtual machine is directly built on a retargetable compiler, rather than a separate one which accepts assembly or binary as input. While the replacement of additional parsing with a direct function call could certainly speed up the compilation, it also ties our tool with a specific compiler. Fortunately, one can build a "binary translation" version of our tool fairly easily.

VI. Experimental Results

A. Experiment Setup

The efficiency of the proposed techniques can only be verified by extensive experiments. However, the following factors contribute adversely to the fairness of direct comparison of our results with others reported in the literature:

- Most previous works on static compilation-based simulation have few results available. For example, [4] has results on only one benchmark.
- Previous works may use different target/host combinations. For example, [3] reports results on simulating SPARC V9 instruction set on SPARC V8 machine.
- Simulators in previous works vary with accuracy. For example, [4] is cycle accurate, and [10] performs bit-true simulation.
- Dynamic compilation-based approaches [3], [2] have dynamic compilation overhead not present in our simulator.

It is hence desirable to implement *all* simulators according to the techniques that they reported for the same target/host combination, and with the same accuracy. Furthermore, dynamic compilation overhead should not be included in the comparison. In this way, we can focus on the effect of register allocation on simulation performance, which is the major contribution of this work.

We manage to do that thanks to the clean simulation code generation API defined in Section III-A. We can implement the virtual machine by emitting C code using a C translator, thus effectively implement a simulator equivalent to [4], [9], [10]. By turning on the optimization switch of the C compiler (the gcc compiler in our study), we argue that this configuration is also equivalent to [3], [2] without dynamic compilation overhead, since with trivial alias analysis, the C compiler is able to perform register allocation at the basic block level. We also argue that the potential performance reduction due to the introduction of virtual machine is eliminated, since the instruction selector of the C compiler can recognize virtual instruction patterns that can be efficiently implemented by one host machine instruction. We are hence confident that conclusions derived from comparing the simulation performance of our proposed simulator with the described mock-up of the previous works are reasonably fair.

We also choose to measure the simulation performance against native execution on the host machine, rather than the target machine. We believe it offers a better measurement on the performance of the simulator since the performance difference between the host machine and target machine is factored out. Also worthy of mention is that for a given benchmark, we use the same retargetable compiler to compile it into target code for simulation and host code for native execution. In other words, they undergo the same frontend analysis and machine-independent optimization. In this way, we are confident that the "code quality" of both are roughly the same, thus making our metric of simulation performance against native execution more reasonable.

We select a set of benchmarks to evaluate our simulator. *COUNT* consists of a loop which simply increments a counter. *IDCT* is the inverse discrete cosine transform algorithm extracted from JPEG/MPEG. *Viterbi* is a popular channel coding algorithm. *FIR* and *LD* (Levison-Durbin) are signal processing algorithms extracted from ITU speech coding standard g.723. LM* are the *Livermore Kernels Benchmark* that were used historically to rate the strengths and weaknesses of vector supercomputers.

We chose the MIPS 3000 as our target machine due to its wide acceptance. We chose an Ultra-5 Sun workstation with 270Mhz UltraSparc CPU and 64M-byte memory as our host machine due to its wide availability in research environments.

B. Simulation Performance

We performed the simulation of the benchmark set, with the total instruction count traced, using both our proposed approach and the C-emitting approach, and compared them against the native execution on the host machine. The results are summarized in Table I, where each row corresponds to a benchmark. The first column (*icount*) records the number of thousand (K) instructions executed for each run of the benchmark. The remaining columns

Benchmark	icount	native		hybrid		C-emitting		lazy		greedy	
	K	MS	MIPS	MS	MIPS	MS	MIPS	MS	MIPS	MS	MIPS
COUNT	30000	11	272	14	272	52	75	100	30	100	30
IDCT	1670	6	278	9	209	32	52	34	49	69	24
Viterbi	23638	116	203	142	185	430	54	450	53	948	25
FIR	8169	27	302	78	122	178	45	167	49	254	32
LD	18447	69	266	198	105	336	54	435	42	632	29
LM1	325	1	325	3	108	6	54	8	40	14	23
LM2	511	4	127	6	85	13	39	13	39	22	23
LM3	198	0.9	220	1	198	3	66	5	39	8	24
LM4	1067	5	213	8	133	20	53	27	40	44	24
LM5	328	1	328	3	109	6	55	8	41	14	23
LM6	7853	25	314	41	191	136	58	193	41	316	24
LM7	274	2.3	119	3.5	78	7.5	37	5.4	51	13	21.2
LM8	1128	6	188	9.6	118	27	41	26	44	47	23
LM9	2127	9.3	228	17	124	44	47	40	53	91	23
LM10	202	0.9	224	2	101	3.7	55	5.3	38	8.2	25
LM11	140	0.7	200	1.7	82	2.6	54	3.5	40	5.7	25
LM12	14247	73.5	193	111	128	350	41	218	65	641	22
LM13	1085	7	155	9.5	114	19	57	15.7	69	50	22
LM14	521	3.2	162	5.4	96	10	52	10	52	20.5	25
LM15	2114	9.4	224	17	124	44	48	36	59	90	23
LM16	321	2.5	128	3.8	84	7.4	43	6.9	47	14	23
LM17	2047	8.5	240	15	136	38	53	26.4	78	90	23
LM18	231	1.2	192	2.3	100	4.5	51	7.2	32	8.9	26
average slowdown				1.59		4.25		4.99		8.50	

TABLE I

COMPARISON OF SIMULATION PERFORMANCE.

record the performance in millisecond (MS) and millions of simulated instruction per second (MIPS) respectively, of different implementation strategies. The column *native* corresponds to compiling and running the benchmark directly on the host machine; the column *hybrid* corresponds to our proposed approach using the hybrid register allocation strategy; and the column *C-emitting* corresponds to the C-emitting approach described before, where the executable is generated by gcc with optimization (with option -O3 turned on). To show the penalty of not performing proper register allocation, we also include simulation performance of our simulator in the last two columns when the lazy and greedy register allocation strategies are employed.

Our results show that our approach simulates the benchmarks at an average speed only 1.59 times slower than native execution, whereas the C-emitting approach, which serves as the mockup of previous efforts, simulates 4.25 times slower than native execution. Our approach is hence about 2.67 times better.

C. Result Analysis

It is interesting to analyze the factors that contribute to the performance difference between our simulator and the native execution to appreciate the simulation results.

- **architecture difference between target and host**: For example, the MIPS has a flat register file architecture, whereas the SPARC architecture has a register window architecture. Hence the target code spends more time saving registers for calls, and so does our simulator. Another example is that the target machine contains instructions which are not directly implemented on the host machine. For our study, the target and host instruction set are very similar. Hence this overhead is not large. Obviously, such overhead can be never avoided.
- **target machine state**: The simulator has to maintain the target machine state. Depending on different register allocation strategies, this overhead might vary. The performance difference between the greedy allocation approach, the lazy approach, and our proposed hybrid allocation approach, quantifies this effect.
- **virtual machine abstraction:** We add one level of indirection, that is, the virtual machine, between the target and host. This will introduce overhead. For example, it might happen that both the target and host have a similar instruction, but the virtual machine does not have an equivalent one. Without a host machine instruction selector implemented, our tool has to use a sequence of host machine instructions to simulate. This overhead can be quantified by the performance difference between the C-emitting approach and the lazy allocation approach for our benchmark set: As mentioned earlier, with proper optimization switch turned on, the C compiler effectively enables basic block level register allocation, hence the overhead on maintaining target machine state should be roughly equivalent to the lazy allocation approach. However, the instruction selector of the C compiler can help to eliminate the overhead of virtual machine abstraction, while lazy approach cannot. This explains their performance difference.

VII. CONCLUSION

In conclusion, we have described a technique which uses a virtual machine code generation interface for the static compiled ISA simulation. We argue that such a low level interface is more efficient than the high level C interface.

Our future work will extend this methodology to perform cycle accurate instruction set simulation, and hardware/software co-simulation, which present more challenges.

References

[1] J. Rowson, "Hardware/software co-simulation," in *Proceeding of the 31st Design Automation Conference*, 1994.

[2] E. Witchel and M. Rosenblum, "Embra: Fast and flexible machine simulation," in *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1996.

[3] R. F. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution profiling," in *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1994.

[4] V. Zivojnvic, S. Tjiang, and H. Meyr, "Compiled simulation of programmable DSP architectures," in *Proceedings of the 1995 IEEE Workshop on VLSI Signal Processing*, Sakai, Japan, 1995.

[5] J. Zhu and D.D. Gajski, "A retargetable, ultra-fast instruction set simulator," in *Proceedings of the Design Automation and Test Conference in Europe*, Munich, Germany, March 1999.

[6] J. Hennessy and D. Patterson, *Computer Organization and Design: The Hardware-Software Interface (Appendix A, by James R. Larus)*, Morgan Kaufman, 1993.

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

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

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

[10] 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.

[11] K. Andrews and D. Sand, "Migrating a CISC computer family onto RISC via object translation," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, P. Marwedel, Ed., October 1992.

[12] R.L. Site et al., "Binary translation," *Communication of the ACM*, Feberary 1993.

[13] C. Cifuentes, "Partial automation of integrated reverse engineering environment of binary code," in *Proceedings Third Working Conference on Reverse Engineering*. November 1996, pp. 50–56, IEEE-CS Press.

[14] W. F. Kao and I. J. Huang, "Instruction retargeting based on the state pair notation," in *Asia Pacific Conference on Hardware Description Languages*, 1997, pp. 114–120.

[15] A. Ghernoff et al., "A profile-directed binary translator," *IEEE Micro*, pp. 56–64, March/April 1998.

[16] D. R. Engler, "VCODE: A portable, very fast dynamic code generation system," in *SIGPLAN Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996.

[17] C.W. Fraser and D.R. Hanson, "A code generation interface for ANSI C," *Software-Practice and Experience*, vol. 21, no. 9, pp. 963–988, September 1991.

[18] D. L. Weaver and T. Germond, *The SPARC Architectural Manual: Version 8*, Prentice Hall, 1992.

Jianwen Zhu (M'00) received the B.S degree in electrical engineering from the Tsinghua University, Beijing, China, the M.S and Ph.D. degree in computer science from the University of California, Irvine, USA, in 1993, 1996 and 1999 respectively.
He is currently an assistant professor in the Department of Electrical and Computer Engineering, University of Toronto, Canada. He is the coauthor of the book SpecC: Specification Language and Methodology (Kluwer-Academic, 2000). His research interest includes hardware/software codesign, high-level synthesis for high-performance circuits and retargetable compilation.

Daniel D. Gajski (M'77 SM'83 F'94) received the Dipl.Ing. and M.S. degrees in electrical engineering from the University of Zagreb, Croatia, and the Ph.D. degree in computer and information sciences from the University of Pennsylvania, Philadelphia.
After ten years of industrial experience in digital circuits, switching systems, supercomputer design, and VLSI structures, he spent ten years in academia with the Department of Computer Science at the University of Illinois, Urbana-Champaign. Presently, he is a Professor in the Department of Information and Computer Sciences at the University of California, Irvine. His interests are in multiprocessor architectures and science of design. He is the coauthor of the books High-Level Synthesis: An Introduction to Chip and System Design (New York: Kluwer-Academic, 1992), Specification and Design of Embedded Systems (Englewood Cliffs, NJ: Prentice-Hall, 1994), SpecC: Specification Language and Methodology (Kluwer-Academic, 2000) and the author of Principles of Digital Design (Englewood Cliffs, NJ: Prentice-Hall, 1985).

```
for( ; ; ) {
    instruction = fetch( pc );
    opcode = decode( instruction );
    switch( opcode ) {
        ....
    case ADD    :
        ....
        break;
        }
    }
```

Fig. 1. Simulation loop of interpretative simulator.

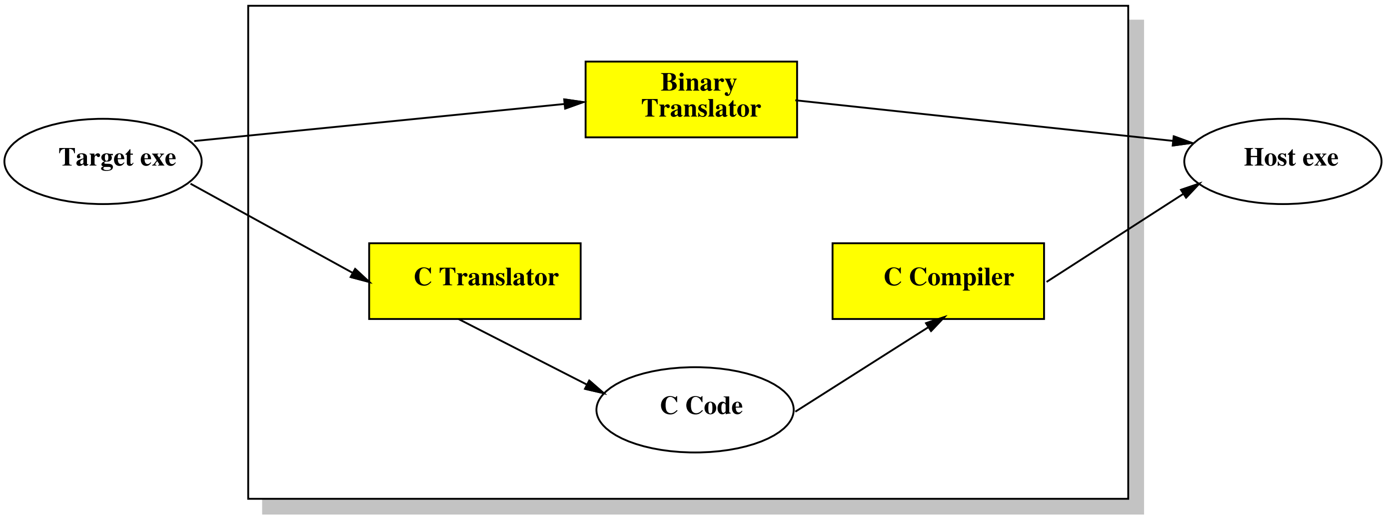

Fig. 2. Static compilation-based simulator.

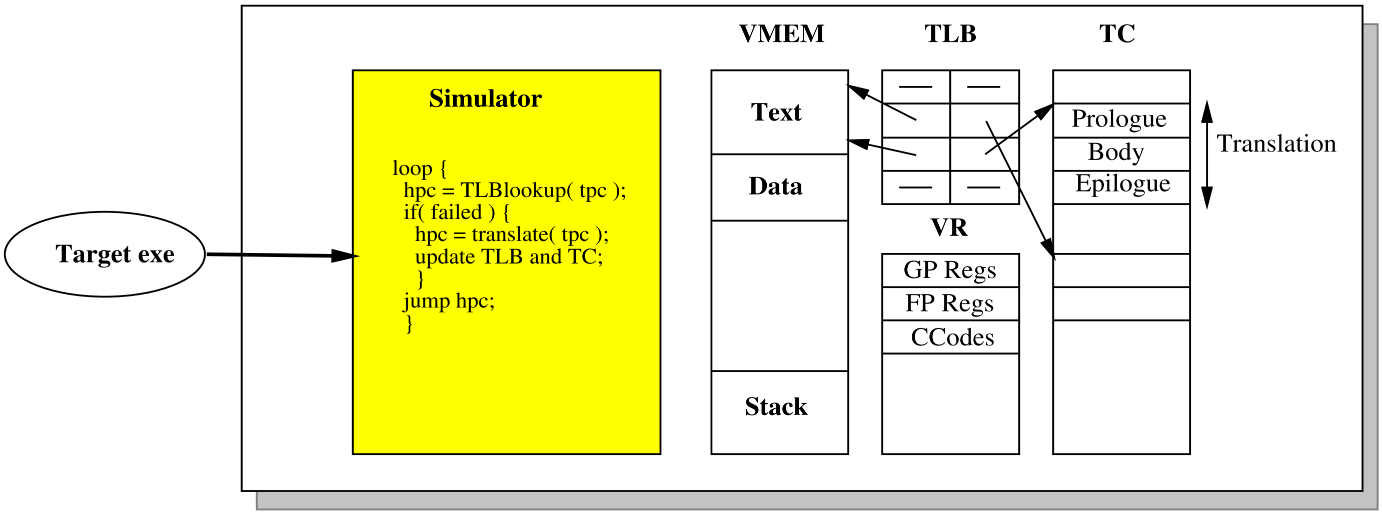

Fig. 3. Dynamic compilation-based simulator.

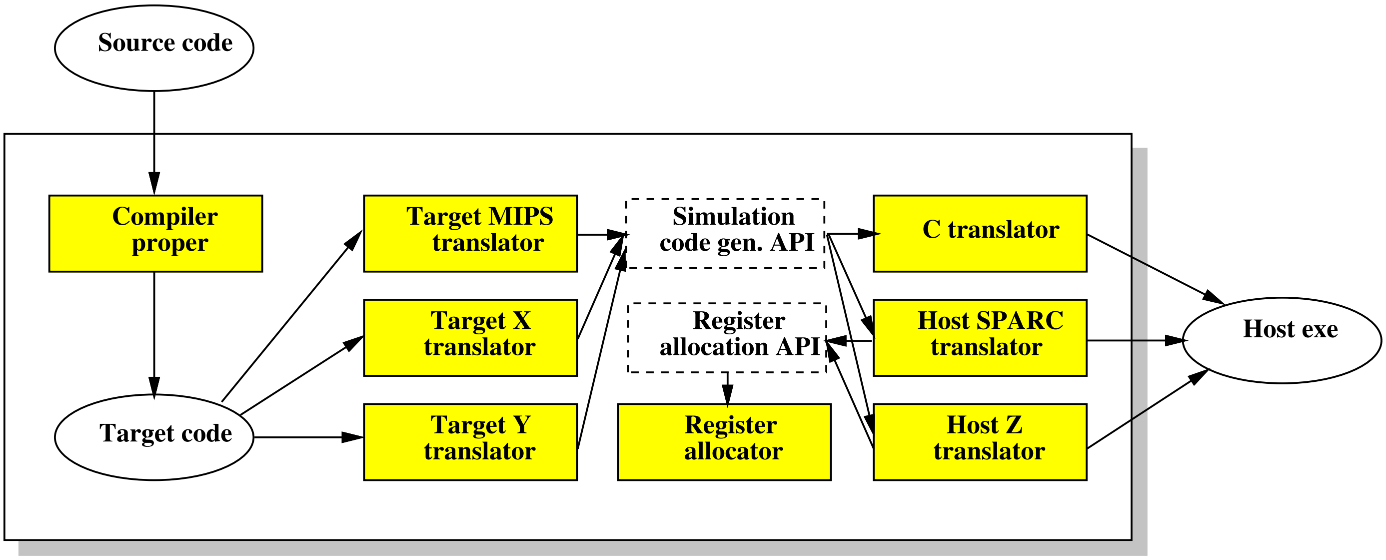

Fig. 4. Our simulator.

```
public enum SegKind {
  SEG_CODE = 1, SEG_BSS, SEG_DATA, SEG_LIT
  }

public interface IHost {
  void begin();
  void end();
  void beginFunction( String name );
  void endFunction( String name );
  void segment( SegKind seg );

  void exportSymbol( String symbol );
  void importSymbol( String name, int size );
  void declSymbol( String name, boolean isstatic );

  void emitConstantValue( Type type,
    Object value );
  void emitAddressValue( String name );
  void emitStringValue( int n, String name );
  void emitSpace( int size );
  void emitAlign( int align );
  void emitInstrn(
    Opcode opcode, Type type,
    TargetExpr dest,
    TargetExpr op1, TargetExpr op2
    );

  int declGlobal( String name );
  int declLocal();
  void undeclAllLocals();

  void    emitFetch( IRegAlloc ra, int vreg );
  void    emitFlush( IRegAlloc ra, int vreg );
  }
```

Fig. 5. Simulation code generation API.

```
public enum Opcode {
  OP_ADD, OP_SUB, OP_MUL, OP_DIV, OP_MOD,
  OP_AND, OP_OR, OP_XOR, OP_SHL, OP_SHR,
  OP_COMP, OP_NOT, OP_NEG, OP_MOV, OP_SET,
  OP_CNVI, OP_CNVU, ...,
  OP_LD, OP_ST,
  OP_RET, OP_J, OP_JAL,
  OP_BLT, OP_BLE, OP_BGT, OP_BGE,
  OP_BEQ, OP_BNE,
  OP_NOP
  }

public enum Type {
  TYPE_C, TYPE_UC, TYPE_S, TYPE_US,
  TYPE_I, TYPE_U, TYPE_L, TYPE_UL,
  TYPE_F, TYPE_D, TYPE_P, TYPE_V,
  }
```

Fig. 6. Virtual machine opcode and data types.

```
public enum RegMark {
    REG_FREE        =   0x01,
    REG_FIXED       =   0x02,
    REG_SPILLABLE   =   0x04,
    REG_DIRTY       =   0x08
    }

public interface IRegAlloc {
    void    start();
    void    end();
    void    declHard( String name, RegMark mark );
    int     declGlobal( String name );
    int     declLocal();
    void    undeclAllLocals();
    String  getName( int vreg );
    String  ask( IHost host, int vreg, boolean isFetch );
    void    kill( int r );
    }
```

Fig. 7. Register allocation API.

```
public class RegAlloc implements IRegAlloc {
    void declHard( String name, RegMark mark ) {
        add a hard register with its name and mark;
        }
    int declGlobal( String name ) {
        int rtn = add a global virtual register;
        record name for rtn;

        forall hard registers hreg {
            if( hreg is marked as free and fixed ) {
                unmark REG_FREE of hreg;
                bind hreg to rtn;
                break;
                }
            }
        return rtn;
        }
    int declLocal( String name ) {
        int rtn = add a local virtual register;
        record name for rtn; return rtn;
        }
    void undeclAllLocals() { delete all locals; }
    String  getName( int vreg ) {
        return the recorded name of vreg;
        }
    String ask( IHost host, int vreg, boolean isFetch ) {
        hreg = hard register assigned to vreg;
        if( hreg != NULL ) {
            if( !isFetch )
                mark REG_DIRTY of hreg;
            return name of hreg;
            }
        rtn = alloc( vreg );
        if( rtn == NULL ) {
            spill( host );
            rtn = alloc( vreg );
            }
        if( vreg is global ) {
            if( isFetch )
                host.emitFetch( host, vreg );
            else
                mark REG_DIRTY of assigned hard register;
            }
        return rtn;
        }
    void kill( int vreg ) {
        preg = hard register assigned to vreg;
        mark REG_SPILLABLE of preg;
        }
    private String alloc( int vreg ) {
        forall hard registers hreg  {
            if( hreg is free ) {
                unmark REG_FREE of hreg;
                assign hreg to vreg;
                return name of hreg;
                }
            }
        }
    private void spill( IHost host ) {
        forall virtual registers vreg {
            hreg = hard register assigned to vreg;
            if( hreg != NULL && hreg is spillable ) {
                host.emitFlush( host, vreg );
                unmark REG_DIRTY of hreg;
                mark REG_FREE of hreg;
                return;
                }
            }
        }
    }
```

Fig. 8. Register allocation algorithm.

Benchmark	icount	native		hybrid		C-emitting		lazy		greedy	
	K	MS	MIPS	MS	MIPS	MS	MIPS	MS	MIPS	MS	MIPS
COUNT	30000	11	272	14	272	52	75	100	30	100	30
IDCT	1670	6	278	9	209	32	52	34	49	69	24
Viterbi	23638	116	203	142	185	430	54	450	53	948	25
FIR	8169	27	302	78	122	178	45	167	49	254	32
LD	18447	69	266	198	105	336	54	435	42	632	29
LM1	325	1	325	3	108	6	54	8	40	14	23
LM2	511	4	127	6	85	13	39	13	39	22	23
LM3	198	0.9	220	1	198	3	66	5	39	8	24
LM4	1067	5	213	8	133	20	53	27	40	44	24
LM5	328	1	328	3	109	6	55	8	41	14	23
LM6	7853	25	314	41	191	136	58	193	41	316	24
LM7	274	2.3	119	3.5	78	7.5	37	5.4	51	13	21.2
LM8	1128	6	188	9.6	118	27	41	26	44	47	23
LM9	2127	9.3	228	17	124	44	47	40	53	91	23
LM10	202	0.9	224	2	101	3.7	55	5.3	38	8.2	25
LM11	140	0.7	200	1.7	82	2.6	54	3.5	40	5.7	25
LM12	14247	73.5	193	111	128	350	41	218	65	641	22
LM13	1085	7	155	9.5	114	19	57	15.7	69	50	22
LM14	521	3.2	162	5.4	96	10	52	10	52	20.5	25
LM15	2114	9.4	224	17	124	44	48	36	59	90	23
LM16	321	2.5	128	3.8	84	7.4	43	6.9	47	14	23
LM17	2047	8.5	240	15	136	38	53	26.4	78	90	23
LM18	231	1.2	192	2.3	100	4.5	51	7.2	32	8.9	26
average slowdown				1.59		4.25		4.99		8.50	

TABLE I

Comparison of simulation performance.