Retargetable Program Profiling Using High Level Processor Models

Rajiv Ravindran and Rajat Moona

Department of Computer Science & Engineering
Indian Institute of Technology
Kanpur 208016, India {rajiva, moona}@cse.iitk.ac.in

Abstract. Program profiling helps in characterizing program behavior for a target architecture. We have implemented a retargetable simulation driven code profiler from a high-level processor description language, *Sim-nML*. A programming interface has been provided for building customized profilers. The retargetability makes the profiling tool independent of the target instruction set.

1 Introduction

During the design of embedded systems, need is felt to automatically generate processor and application development tools like assemblers, disassemblers, compilers, instruction-set simulators etc. Automated generation of such tools yields faster turnaround time with lower costs for system design and simplifies the process of incorporating design changes. We have developed a retargetable environment in which processors are modeled at a high level of abstraction using the *Sim-nML* [3] specification language.

We use *Sim-nML* to describe the instruction set architecture of the processor from which various tools to aid processor design are automatically generated. The overall goal of the *Sim-nML* based project is to model a complete system environment with a processor core specified using *Sim-nML* for automatic architecture exploration. As part of the integrated development environment, we have developed a retargetable functional simulator [5], cache simulator [6], assembler, disassembler [13] and a compiler back-end generator. Work on a cycle accurate timing simulator is in progress from the *Sim-nML uses* [14] model. In this paper, we describe a mechanism for code analysis. Program analysis tools are extremely useful for understanding program behavior. Computer architects could employ them to analyze program performance on new architectures. It can be used to characterize instruction usage, branch behavior etc. Critical and time consuming portions of the code can be identified for optimizations. Compilers could use the profile information to guide its optimization pass.

Program profiling has always been instruction-set specific. Profiling tools instrument the input binary at specific points to sample the program behavior. Other techniques employed include hardware counters which are built into the processor. But most of these techniques are tied to a particular architecture. In the embedded world, the designer has to iterate over multiple design options to decide on the best target architecture for a given application, within a short period of time, before actual processor design. Hence, there is a need for a more generic model.

We have tried to build a mechanism that provides architects and software developers a means to implement various profiling policies in a retargetable manner. A retargetable instruction set simulator provides the platform for code profiling. A retargetable functional simulator generator - *Fsimg* [5], generates a processor specific functional simulator from the processor model written in *Sim-nML* for a given program. A high-level processor modeling language makes our design retargetable. In our approach, the profiling of code is accomplished by instrumenting the functional simulator, for the given program on the target architecture, at chosen points. For example, to count the basic blocks traversed at run time, a counter could be placed at the end of each basic block. Similarly, to analyze branch behavior, routines could be added after conditional branch instructions. We view the program as a set of procedures each containing a collection of basic blocks, each of which is further composed of processor instructions. A user defined procedure for profiling the application program can be inserted before or after an instruction, a basic block or a procedure. We have provided an application programming interface (API) using which the user can insert his function calls at chosen points in the input binary. This technique is inspired from ATOM [7] which is a framework for building a wide range of customized program analysis tools. Through this approach, the user can construct a custom profiling tool. The retargetable program profiler is a step towards our original goal of complete system simulation consisting of a processor simulator, cache simulator etc. from *Sim-nML*.

Our code profiling strategy consists of two phases. In the first phase, the retargetable functional simulator generator *Fsimg*, takes as input the *Sim-nML* processor model describing the target architecture and the program binary. The API-calls helps the user define his routines and the points in the program binary to be instrumented. The granularity of instrumentation can be at the procedural, basic block or at instruction boundaries. *Fsimg* then generates an instruction set simulator -*Fsim*, specific to the given program and target processor instruction set. The instrumentation routines are linked with the simulator with user provided calls added at specified points in *Fsim*. In the second phase, the user runs the functional simulator which executes the instrumented code while simulating the input program. Thus the program profile is generated.

Our target through this paper has been two fold. Firstly, as an extension of the integrated development environment we have implemented a code profiler. Unlike earlier works on profiling, we have attempted to develop a retargetable profiler from a high-level processor description. Secondly, the API-calls provides an infrastructure to create a customized profiling environment through user specified functions.

The rest of the paper is organized as follows. In section 2, we list the related work. In section 3, we give an overview of *Sim-nML*. We describe the design of the code profiler in section 4. We explain the API with some examples of how they could be used for different kinds of profiling. Finally in section 5, we present some sample simulation and profiling results for simple programs and conclude.

2 Related Work

Performance modeling of a system is a growing area and a lot of research has been pursued in this area. We briefly review some of them here.

ATOM [7] provides a framework for providing customized program analysis tools. It instruments the input program through a programmable interface. ATOM uses no simulation or interpretation.

Pixie [8] is a utility that allows one to trace, profile or generate dynamic statistics for any program that runs on a MIPS processor. It works by annotating executable object code with additional instructions that collect the dynamic information during run time.

QPT [9] [10] is a profiler and tracing system. It rewrites a program's executable file (a.out) by inserting code to record the execution frequency or sequence of every basic block or control-flow edge.

EEL [11] (Executable Editing Library) is a C++ library that provides abstractions which allow a tool to analyze and modify executable programs without being concerned with particular instruction sets, executable file formats, or consequences of deleting existing code and adding foreign code.

EXPRESSION [1] is similar to *Sim-nML* and is used in architecture exploration. It has been used for automatic generation of compiler/simulator toolkit.

LISA [2] is a machine description language for generation of bit and cycle accurate models for DSP processors based on behavioral operation description.

UNIX tools prof and gprof record a statistical sample.

All of these tools are architecture specific or implement a specific set of profiling techniques (except ATOM). We try to improve upon them through retargetability and customization.

3 Sim-nML

Sim-nML[3][4] is a direct extension of *nML*[15] machine description formalism. It includes several features that are useful for performance simulation that are not present in *nML*.

Sim-nML is targeted for describing wide range of processor architectures including CISC, RISC and VLIW architectures at the instruction set level hiding its implementation details. Some of the target architectures described using *Sim-nML* include PowerPC, ARM, UltraSparc, MIPS-IV, 8085, Motorola 68HC11, ADSP2101. The instruction set is described in a hierarchical manner. The semantic actions of the instructions are captured as fragments of code spread all over the instruction tree. *Sim-nML* specifications are described using an attribute grammar. There is a fixed start symbol called *instruction,* and two types of productions, *and-rule* and *or-rule*.

There are certain fixed attributes defined for *and-rules* that capture various aspects of the instruction set. The *syntax* attribute captures the textual assembly language syntax of the instructions. The *image* attribute captures the binary image of the instructions. The *action* attribute captures the semantics of the instructions. The *uses* attribute captures the resource usage model and is used for timing simulation. The resource usage model captures the micro-architectural features of the processor and is used in cycle accurate timing simulation. The details of the uses model is given in [14].

The following illustration is a specification for a simple processor with four instructions – *add, sub, bim* (branch immediate) and *bin* (branch indirect).

The processor has two addressing modes – immediate and register indirect. The *mode* rule is associated with a *value* attribute, ‘n’ in case of *IMM* and ‘R[n]’ in case of *REG_IND*, where ‘n’ is a constant provided in the instruction encoding. The four instructions are hierarchically described. The branch immediate (*bim*) instruction modifies the PC with an immediate branch offset. The branch indirect (*bin*) takes the branch address in the specified register and copies it to the PC. The *add* instruction adds two registers and puts the results in the first register. The *sub* instruction subtracts two registers and puts the result in the first register.

```
let REGS = 5
type word = int ( 16 )

reg R [ 2**REGS, word ]
reg PC [ 1 , word ]

resource bu, alu

mode IMM ( n : card ( 12 ) ) = n
  syntax = format( “%d”, n )
  image  = format( “%12b”, n )

mode REG_IND ( n : card ( 5 ) ) = R [ n ]
  syntax = format( “r%d”, n )
  image  = format( “%5b”, n )

op instruction ( x : instr_action )
  uses   = x.uses
  syntax = x.syntax
  image  = x.image
  action = {
             PC = PC + 2;
             x.action;
           }
op instr_action = branch_inst | arithmetic_inst

op branch_inst = bim | bin

op bim ( d : IMM )
  uses = bu #1
  syntax = format( “bim %s”, d.syntax )
  image  = format( “1000%s”, d.image )
  action = { PC = PC + (d << 4); }
```

```
op bin ( r : REG_IND )
  uses = bu #1
  syntax = format( “bin %s”, r.syntax )
  image  = format( “10010000000%s”, r.image )
  action = { PC = r; }

op arithmetic_inst = add | sub

op add ( r1 : REG_IND, r2 : REG_IND )
  uses = alu #1
  syntax = format( “add %s %s”, r1.syntax, r2.syntax )
  image  = format( “101000%s%s”, r1.image, r2.image )
  action = { r1 = r1 + r2; }

op sub ( r1 : REG_IND, r2 : REG_IND )
  uses = alu #1
  syntax = format( “sub %s %s”, r1.syntax, r2.syntax )
  image  = format( “101100%s%s”, r1.image, r2.image )
  action = { r1 = r1 − r2; }
```

Retargetable tools flatten out the hierarchical description to enumerate out the complete instruction set and its associated attribute definitions. Different tools, depending on their needs, use different attributes. For example, an assembler uses the definition of attributes *syntax* and *image*. A detailed description of *Sim-nML* can be found in [3][4] (*http://www.cse.iitk.ac.in/sim-nml*).

4 Code Profiling

In this section, we give an overview of the functional simulator, discuss the design and implementation of the code profiler, the API and how they are incorporated into the functional simulator.

4.1 Overview of Functional Simulation Process

Fsimg [5] initially flattens the hierarchy in *image* and *action* attributes of the *Sim-nML* description. The *action* attribute captures the semantics of the instruction-set. For each machine instruction, *Fsimg* emits a corresponding unique *C* function. This function is obtained by translating flattened *action* attribute definition to *C*. *Fsimg* then reads the program binary and for every matched input instruction image, generates a call to the corresponding function defining that instruction. All these calls to functions are captured in a table of instruction function pointers with each entry pointing to the function corresponding to the respective instruction. Along with this table, *Fsimg* also generates data structures for memory, registers and a driving routine for simulation. Thus, it generates a list of function calls corresponding to all instructions in the input program.

The driver routine of the functional simulator simulates the program by calling these functions sequentially until the program terminates.

4.2 Instrumentation

The table of instruction function pointers generated by the functional simulator provide a convenient means for code instrumentation. To create a customized profiling tool, the user defines a set of routines which are to be executed at chosen points in the program. Calls to these user defined routines are inserted into the table of instruction function pointers between function calls at specified instrumentation points which represent the instruction boundaries. A set of predefined routines - an application programming interface (API) is provided which allows the user to add his procedure calls before or after instructions. A set of *Basicblock analysis* routines are provided for profiling at the level of procedures, basic blocks and instructions within basic blocks.

The profiler generation consists of the following steps.

- *Fsimg* analyzes the input program for basic blocks. *Fsimg* is provided with the set of conditional and unconditional control flow instructions in the *Sim-nML* hierarchy. Since *Sim-nML* is a hierarchical description, if the hierarchy allows, we could provide the top level branch node instead. The actual branch instruction is then enumerated from this. Once a list of branch instructions are enumerated, we split the input instruction stream at procedure boundaries. For a given procedure, the basic block boundaries are marked just after every branch instruction. The branch target address can be calculated from the *action* corresponding to the branch instructions.
- User adds his routines through the instrumentation-API within a predefined *Instrument* function.
- During generation of the functional simulator, the API calls are used to instrument the application program at appropriate places in the generated simulator.
- User runs the simulator which executes the instrumented code.

Application Programming Interface - API For code instrumentation, we list some important instrumentation-API below.

- **AddCallFuncbyName(iname, func, position)**: This function can be used to instrument the application program before/after (*position*) the specific instruction (*iname*) with the user defined routine (*func*).
- **AddCallFunc(inst, func, pos)**: This function can be used to instrument the application program at specific addresses i.e, whenever an instruction is fetched from the address *inst*.
- **AddTrailerFunc(func)**: The user can add any routines (*func*) to be executed after simulation. *Fsimg* adds these routines after the simulation engine. They can be used to collect the final statistics, dump profiling information etc.
- **GetFirstProc, GetNextProc, GetFirstBlock, GetNextBlock, GetLastInst**: These procedures are used at the procedure and basic block level. They can be used as iterators over all procedures and basic blocks in the given program.

For more details refer to [12].

Implementation The instrumentation routines are added in 3 files - *instrument.c, bblockanal.c, userfuncs.c*. *Instrument.c* contains a call to a predefined routine *Instrument* in which the user adds the API calls.

For example, to count the occurrences of *add* instructions executed in the program, we specify the following

```
void Instrument()
{
   AddCallFuncbyName(“add”, INSTR_TYPE, “addcounter”, AFTER);
   AddTrailerFunc(“printaddcnt”);
}
```

The file *userfuncs.c* contains the user defined routines. The function *addcounter* could be defined as follows:

```
long addcnt = 0;
void addcounter()
{
      addcnt++;
}
```

where *addcnt* is a global counter. *AddTrailerFunc* is used to add the user function *printaddcnt* at the end of simulation which could be defined as follows

```
void printaddcnt()
{
      printf(“%d”, addcnt);
}
```

The file *bblockanal.c* contains the instrumentation routines associated with basic block related analysis. It contains a call to a predefined routine *BasicblockAnal*, in which the user adds the API calls for basic block related profiling.

For example, to count the number of basic blocks that are traversed during program execution, we specify the following

```
void BasicblockAnal()
{
   Proc *p;
   Block *b;
   Inst inst;
   for (p = GetFirstProc(); p; p = GetNextProc(p))
     for (b = GetFirstBlock(p); b; b = GetNextBlock(b))
       inst = GetLastInst(b);
       AddCallFunc(inst, “countbb” AFTER);
   AddTrailerFunc(“printbb”);
}
```

The user defined function *countbb* is added *after* the last instruction in each basic block. The user might want to call different functions at the same address boundary. Multiple user defined instructions can be engineered at address boundaries by calling *AddCallFunc* with different function names at the same instruction address.

5 Results

Five benchmarks program were written in C (Table 1) and compiled for *PowerPC603 Sim-nML* processor description.

Program	Description
mmul.c	Integer matrix multiplication. This program initializes two integer matrices of 100x100 size and multiplies them.
bsort.c	Bubble sort. This program initializes an array of 1500 integers in descending order and sorts them to ascending order using bubble sort algorithm.
qs.c	Quick sort. This program initializes array of 1,00,000 integers in descending order and sorts them to ascending order using quick sort algorithm.
fmmul.c	Matrix multiplication of floating-point numbers. Initializes and multiplies two floating point matrices of size 100x100.
nqueen.c	This program finds all possible ways that N queens can be placed on an NxN chess board so that the queens cannot capture one another. Here N is taken as 12.

Table 1. Benchmark Programs

Table 2 gives the total number of dynamically executed instructions during the simulation.

Program	Total No. of Instructions
mmul.c	91,531,966
bsort.c	60,759,034
qs.c	80,773,862
fmmul.c	92,131,966
nqueen.c	204,916,928

Table 2. Total number of instructions simulated for test programs.

We have implemented a simple profiling tool which counts the number of basic blocks that are traversed at run time. At the same time, the number of *PowerPC603*

addi instructions executed is found. The code instrumentation technique used is the one specified in section 4.2.

The profiling output is given in table 3.

Program	Total No. of basic block traversed	Total No: of addi instructions executed
mmul.c	2081207	1030305
bsort.c	4506008	2253005
qs.c	7315513	242144
fmmul.c	2081207	1110305
nqueen.c	40030204	60766515

Table 3. Profiling output for test programs.

The functional simulator performance without any profiling code is shown in table 4

Program	Total Time in Seconds	Instructions per second
mmul.c	62	1,476,322
bsort.c	106	573,198
qs.c	109	741,044
fmmul.c	64	1,439,549
nqueen.c	225	910,741

Table 4. Performance Results of the functional simulator

We compare the simulation slow down from different profiling techniques (those in table 3) in table 5.

Program	Slowdown factor
mmul.c	1.01x
bsort.c	1x
qs.c	1.01x
fmmul.c	1x
nqueen.c	1.01x

Table 5. Performance results of profiling of test programs.

6 Conclusion

In this paper, we presented a simulation driven program profiler. The profiler is generic in the following ways. Firstly, we have used a retargetable functional simulator generated from a high-level processor description language, *Sim-nML*. Secondly, through a programming interface, we have provided a mechanism for implementing customized profilers. Thus, we do not tie the profiler to a particular instruction-set or to specific profiling techniques.

References

1. Ashok Halambi, Peter Grun , Vijay Ganesh, Asheesh Khare, Nikil Dutt, Alex Nicolau: *EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability.* Proceedings of the Conference on Design, Automation and Test in Europe, Munich, Germany, March 1999
2. S. Pees, A. Hoffmann, V. Zivojnovic, H. Meyr: *LISA - Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures.* Proceedings of the 36th Design Automation Conference, New Orleans, June 1999
3. V. Rajesh, Rajat Moona: *Processor Modeling for Hardware Software Co-Design.* Proceedings of the 12th International Conference on VLSI Design, Goa, India, January,1999
4. Rajat Moona: *Processor Models for Retargetable Tools.* Proceedings of Rapid Systems Prototyping 2000 (IEEE), Paris, June, 2000
5. Subhash Chandra and Rajat Moona: *Retargetable Functional Simulator Using High Level Processor Models.* Proceedings of the 13th International Conference on VLSI Design, Calcutta, India., January, 2000
6. Rajiv Ravindran and Rajat Moona: *Retargetable Cache Simulation Using High Level Processor Models.* Proceedings of the 6th Australasian Computer Systems Architecture Conference, Gold Coast, Australia, January, 2001
7. Amitabh Srivastava and David Wall: *ATOM: A System for Building Customized Analysis Tools.* Proceedings of the SIGPLAN '94 Conference of Programming Language Design and Implementation, June, 1994, 196-205
8. Michael D. Smith: *Tracing with Pixie.* Memo from Center for Integrated Systems, Stanford Univ., April, 1991
9. James R. Larus: *Efficient Program Tracing.* IEEE Computer, May, 1993, 26(5):52-61
10. James R. Larus, Thomas Ball: *Rewriting Executable Files to Measure Program Behavior.* Software: Practice and Experience, Feb, 1994, 24(2):197-218
11. James R. Larus, Eric Schnarr: *EEL: Machine-Independent Executable Editing.* SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 1995
12. Rajiv Ravindran: *Retargetable Profiling Tools and their Application in Cache Simulation and Code Instrumentation.* Masters thesis report, Dept. of Computer Science and Engg., IIT Kanpur, India, Dec 1999. http://www.cse.iitk.ac.in/research/mtech1998/9811116.html
13. Nihal Chand Jain: *Disassembler Using High Level Processor Models.* Masters thesis report, Dept. of Computer Science and Engg., IIT Kanpur, India, January 1998. http://www.cse.iitk.ac.in/research/mtech1997/9711113.html
14. Anand Shukla, Arvind Saraf: *A Formalism for Processor Description.* Bachelors thesis report, Dept. of Computer Science and Engg., IIT Kanpur, India, May 2001.
15. M. Freerick: *The nML Machine Description Formalism.* http://www.cs.tu-berlin.de/˜mfx/dvi_docs/nml_2.dvi.gz, 1993