MirvSim

A high level simulator integrated with the Mirv compiler

Krisztián Flautner, Gary S. Tyson and Trevor Mudge
Advanced Computer Architecture Laboratory, EECS Department
University of Michigan
Ann Arbor, MI 48109-2122
{manowar, tyson, tnm}@engin.umich.edu

Abstract

A program's execution profile is an increasingly important source of information for optimizations. Along with its use for high-level optimizations, profile information can be used to take advantage of advanced ISA features such as branch hint bits. Profile information is collected either by instrumenting an executable with profile collection code or by running the program in a simulator. Instrumentation has the advantage that it is relatively fast, with the disadvantage that the instrumentation code can interfere with the measurements (e.g. branch prediction). Traditional simulation is slower but a more accurate process. However, since most simulators simulate the low level machine code of a program, relating the results to the high-level program structure can be problematic. MirvSim differs from other simulators – such as SimpleScalar – that are used in computer architecture in that instead of simulating a machine code level instruction set, it simulates the Mirv language, a high-level intermediate program representation (IR). This allows the simulator to be closely coupled with the rest of the compiler infrastructure and to communicate information with other compiler passes easily.

1. Introduction

A program's execution profile is an increasingly important source of information for optimizations [1]. Profile information is collected either by instrumenting an executable with profile collection code [2][3][4] or by running the program in a simulator. Instrumentation has the advantage that it is relatively fast but has the disadvantage that the instrumentation code can interfere with the measurements (e.g. branch prediction). Traditional simulation is slower but a more accurate process. However, since most simulators simulate the low level machine code of a program, relating the results to the high-level program structure can be problematic.

MirvSim differs from other simulators – such as SimpleScalar [6] – that are used for computer architecture studies in that instead of simulating a machine code level instruction set, it simulates the Mirv language, a high-level intermediate program representation (IR). Instead of memory addresses, the simulator knows about variables and types, and in lieu of a set of branches and labels, the high level control-flow structure of the program is available.

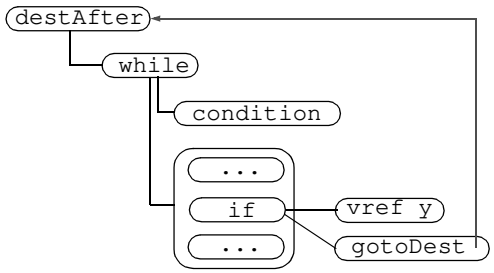
The simulator can be configured with *simulation observers* that collect information about the execution of the program in a "compiler friendly" manner. The collected information is marked at the nodes of the operator tree, allowing subsequent optimization passes to take advantage of the results. In essence, the simulation is just another analysis step that is performed in the compiler.

This infrastructure can be used to evaluate optimization options. For example, an observer can detect the branch prediction accuracy for individual control-flow constructs and after spotting problem spots, take advantage of unique features of the target processor (such as predication or compiler allocated predictors). Similar opportunities may be uncovered by observing object references and the producer-consumer relationships between them.

Section 2 provides an overview of the high level representation and the compilation model. Section 3 describes the simulation methodology and Section 4 describes the profile collection process, illustrating it with branch prediction and load-store pairing examples. Section 5 finishes with the conclusion and future directions.

FIGURE 1. C to Mirv conversion example

```
while(a < x) {
    ...
    if (y) break;
    ...
}
```



This figure illustrates how a C while loop is converted into Mirv constructs. Note that the control-flow structure of the original code is preserved (both the while loop and the if statement), and a structured goto operator is used to provide a way to exit from a nested block. The `destAfter` operator specifies a destination target (meaning that the target of an associated goto instruction is after the body of the destAfter operator). The `gotoDest` operator is used to jump to the `destAfter` label and achieve the same effect as a break statement in C.

2. IR and compilation model

The in-memory representation consists of objects that correspond to operators in the Mirv language along with type, constant and variable information. Mirv is a prefix language that was designed for use with attribute propagation based compiler techniques.

The language preserves the high-level control-flow structure of the source program, along with variable and type information (see Figure 1). Mirv in essence is a cleaned up prefix representation of the C language. It is intended to be generated and used by programs and not as a language for human consumption.

The compilation process is made up of a successive application of filters to the operator tree. Analysis filters propagate attributes over the structure of the tree and mark the outcome as node attributes on the representation. Any node, both operator or symbol table node, may contain attribute collections. Optimization filters rely on preceding analysis filters (more specifically, the node attributes generated by them) for information, and may change the structure or the node attribute values of the representation.

The Mirv Simulator fits into this model since it works on the same high-level representation as the rest of the filters. The Simulator Observers mark the information collected about the simulation as node attributes on the representation. Successive optimizations steps can base their behaviour on information generated by the simulator and its observers.

3. Simulation methodology

The core of the simulator consists of a filter that propagates inherited and synthesized attributes over the program representation. These attributes are used to perform the computations described by the program. Parsing rules are invoked based on the structure of the program, that perform the appropriate operation in the operator tree. The inherited attribute to these nodes is an evaluation context object. The evaluation context's principal role is to map variable objects to memory locations. The synthesized value is the result of the computation performed by the node. Table 1 illustrates some of the high level constructs and the associated behaviour performed by the simulator.

In order to make the simulator a good tool for analysis, computations that are related to analysis and not to simulation are encapsulated in observer objects. For each run, the simulator can be configured with any number of observers that collect information about the program's execution. The results of the observers are usually marked as node attributes on the operator tree and can be used by later analysis and transformation steps.

Before a simulation is run, variables are resolved to memory locations, so that pointer arithmetic can be carried out accurately. This also facilitates interfacing to native compiled code by storing data in the machine's native format. The simulator has access to a mapping of memory addresses to variables, which can be used in the observation process.

TABLE 1. Some operators and their behaviour during simulation

Operator node	Behaviour
destAfter	•Enter a nested exception handler context. •Evaluate the body. •If an associated exception is caught, then return.
funcCall	•Evaluate the function call arguments. •Set up the called function's evaluation context. •Evaluate the called function.
gotoDest	•Throw an exception (include the destination name in it).
ifElse	•Evaluate the condition. •If the condition is true, then evaluate the ifBody, else evaluate the elseBody.
sequence	•Evaluate every element in the sequence in forward order.
while	•Evaluate the condition. •If the condition is true, then evaluate the body, else return. •Repeat the first two steps.
binary expr	•Evaluate the first argument. •Evaluate the second argument. •Perform the specified operation and return the result.
varRefDirect	•Look up the memory location of the specified variable in the symbol table. •Read the value from the memory location and return it as the result.

Before commencing simulation, an optimization filter traverses the representation to cache information at the nodes of the operator tree that speeds simulation. Currently two sets of information are cached:

- Type information which can be used to eliminate double dispatch in expression value computations by determining the type of the operands and the operator.
- The size and class of data (integer, float, memory address or aggregate) that is being accessed in memory.

Native functions can be invoked from the simulator using a native code interface. Currently the C library functions are accessed using it.

4. Profiling with MirvSim

The advantage of the simulator lies in its ability to associate information directly with the high level representation. This information can be used to generate code that better utilizes features of the underlying microprocessor or simply to get a better understanding of the characteristics of the simulated code. Information that is currently of interest:

- Control-flow information.
- Producer-consumer behaviour of variable references.
- Value profile collection to facilitate partial evaluation / memoization.
- The existence of function side effects for extracting parallelism from a program.

The Mirv representation allows one to interpret the results in the high level context of the program. For example when doing branch prediction studies, one can find out what type of node (i.e. while loop, doWhile loop, if, ifElse, etc.) a branch belongs to. When looking at the producer-consumer behaviour of variable references, the set of communicating variables are known instead of just a set of memory addresses. Similarly, when looking at the value profile of a program, the profile of individual variables is also available.

The following sections give representative examples of the analyses outlined in the first two bullets.

FIGURE 2. Control flow information about two nodes in the adpcm benchmark

```
<MVOpIfElse: 0x4b6da8>
  | bpGShareA attribute pool
  |   Accuracy -> 0.96787109375
  |   CorrectWeak -> 515
  |   IncorrectWeak -> 311
  |   IncorrectStrong -> 18
  |   CorrectStrong -> 9396
  | BranchInfo attribute pool
  |   branchTaken -> 5120
  |   BranchPattern -> <0x8496544>
  |   branchNotTaken -> 5120
  |   Classification -> FlipFlopTNT
```

```
<MVOpIf: 0x4bd46c>
  | bpGShareA attribute pool
  |   Accuracy -> 0.56376953125
  |   IncorrectWeak -> 2174
  |   CorrectWeak -> 2426
  |   CorrectStrong -> 3347
  |   IncorrectStrong -> 2293
  | BranchInfo attribute pool
  |   branchNotTaken -> 4327
  |   BranchPattern -> <0x8498668>
  |   branchTaken -> 5913
  |   Classification -> Random
```

The picture above illustrates some of the information gathered about two control-flow nodes in a program. Two attribute pools are present at each node, containing information about a GShare branch predictor (16-bit history 64K 2-bit counters) and other miscellaneous branch information about the node.

4.1 Control-flow behaviour

An important class of optimizations in a compiler are the optimizations related to the control-flow structure of a program. Processors supply a varying amount of hardware support for these kinds of optimizations. Some processors support predicated execution and branch hint bits or even compiler scheduled branch predictors, others do not.

The Mirv Simulator can be configured with observers that collect information about the behaviour of control flow nodes. These observers can simply mark the number of times a control-flow node was evaluated, keep history tables for further analysis, or can even map branch-prediction results to the individual control-flow nodes.

Even though programs are stored in their high level forms, there exists a straight forward mapping to the low level operations that constitute the high level construct. For branch predictor accuracy measurements this low level structure becomes useful, since it is branches that are predicted, not the high level control structures.

As an example this information can be used to classify the control-flow structures into categories that are potentially useful for the code generator for fine tuning the branch predictor (see Figure 2 for examples of these classifications). Current classifications are based on:

- Accuracy - This information can be based on actual measurements from a branch predictor or can be "guessed" based on the characteristics of the control flow outcomes. A hard to predict branch can be a hint to following optimization filters to use predication.
- Static predictability - FlipFlopTNT and FlipFlopNTT means that the control flow of the operator flip flopped between the taken and not taken states (starting with taken and not taken respectively), meaning that its branching behaviour is completely described.
- Dynamic predictability - MostlyTaken and MostlyNotTaken describe control flow nodes that are very highly biased in one direction or the other.

Figure 2 illustrates the information that is collected about control-flow nodes. To get this data, the simulator was outfitted with three observers:

- A generic branch observer that simply marked the number of times the condition of the node was true and false. This information was marked in the BranchTaken and BranchNotTaken attributes.
- Another observer implemented a branch GShare branch predictor with 16 bits of history and a 64K entry 2-bit counter table. The observer computes the accuracy of the prediction for each control-flow node along with the accuracy of the entire program.
- Yet a third observer was used to record the control-flow patterns of the node and a pointer to the generated table was saved in the BranchInfo attribute pool.

After the simulation was done a filter traversed the operator tree and computed a classification for each node. This information was based on the BranchPattern attribute. The branch outcome of the ifElse node flip-flops between taken and not taken throughout the entire execution of the program, while the outcomes of the if

TABLE 1. Prediction accuracy for conditional control flow nodes in the adpcm benchmark

Node type	Correct prediction		Incorrect prediction		Accuracy
	Strong bias	Weak bias	Strong bias	Weak bias	
if	0.662663	0.12831	0.110038	0.098990	0.790973
ifElse	0.794229	0.089364	0.058940	0.057466	0.883594
while	0.951033	0.020456	0.002246	0.026266	0.971489

node are very dependent on the input set of the program. While there is not much a branch predictor can do about the control-flow node in the second example, branch prediction accuracy for the first example should have been 100%. Given a suitable processor architecture and compiler, this information could be encoded in the ISA with hint bits to increase prediction accuracy.

Table 1 illustrates the break down of prediction accuracy for the individual conditional control-flow nodes in the adpcm benchmark run on a random input set. The branch predictor used in this example was a GShare predictor with 16bits of history and 64K 2bit counters. The information is presented in order to illustrate the capabilities of the simulator. These kinds of mappings can provide valuable insight into the interaction of the high-level program and the underlying hardware.

Another important piece of information about loops is whether the number of iterations is controlled by a single variable, and weather the value of that variable changes in a regular manner. The induction variable observer attempts to find the induction variable of a loop and the amount it changes during each iteration. If this amount is the same from one iteration to another, then the iteration characteristics of the loop are fully known (provided that there are no breaks in the loop body). These loops do not really need to be predicted, since their control-flow characteristics are fully known once the induction variable receives its initial value. Given a suitable architecture, the induction variable could be assigned to a counter register (i.e. on the PowerPC), which – along with stride information – can be used to generate perfect prediction for the loop.

4.2 Load-store pairings

Another important area of optimizations is the memory system. Memory renaming [5] is a technique that aims to predict dependencies between load and store instructions. In conventional processors ambiguous memory dependencies require loads to stall until all store addresses prior to the load have been computed. This approach is overly conservative, since a load is not necessarily dependent on all prior stores. Knowing which loads and stores pass values between each other would allow the dynamic instruction scheduler to be less conservative about when loads can execute.

The Producer-Consumer observer generates a mapping of assignment operators to variable reference operators (weighted by the number of occurrences of the

TABLE 2. Number of producers by consumer type for the adpcm and g721 benchmarks

Consumer type	Smallest number of producers	Largest number of producers	Average number of producers	
			Indirect	Indirect + Direct
Integer	1	22	7.8	2.3
Float	1	18	7.2	4.4
Pointer	2	24	13.3	3.2
Array element	1	22	13	13
Struct member	1	9	5.4	5.4

mapping). A connection between an assignment and a variable reference operator corresponds to data flowing between the two nodes. In the low-level code, this relationship manifests itself as communication between a load and a store instruction. One way to look at this is that the Producer-Consumer observer captures the Static Single Assignment form for the specific execution of the program. However, unlike in the statically computed SSA form conservative assumptions with respect to variable aliasing are not needed, since the exact information is available from the profile. This information can be used by the compiler to aid memory renaming by assigning the tags to communicating instructions at compile time.

Table 2 gives a break down of the number of producers per consumer for variable reads of different types. The information was collected using a filter that traversed the operator tree after the simulation, computing the results based on node attributes marked by the Producer-Consumer simulator observer. There are two kinds of averages shown: one for indirect references only and another for all references. The reason for the separation is that indirect references involve pointer arithmetic and thus have to access memory. Most direct references, on the other hand can be allocated to registers.

5. Conclusions and future work

MirvSim has demonstrated that simulation of the high-level representation is a viable alternative to low-level simulation. The simulator can be easily outfitted with observers to profile different aspects of the execution and the results of the analyses can be readily fed back to the compiler.

MirvSim enables architectural studies that evaluate results in the context of the high-level program. The main strength of the simulator is its tight integration with a compiler and as such is a good platform for exploring profile-feedback based optimizations.

6. Acknowledgments

This research was supported by DARPA grant DABT63-97-C-0047 and NSF grant CCR-9812415. Special thanks to Peter L. Bird for invaluable insights and for laying the foundation of the Mirv compiler.

References

[1] Andrew Ayers, Stuart de Jong, John Peyton and Richard Schooler "Scalable Cross-Module Optimization" *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)* pp. 301-312, (1998).

[2] Amitabh Srivastava and Alan Eustace "ATOM: A System for Building Customized Programs Analysis Tools" *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)* pp. 196-205, (1994).

[3] Susan L. Graham, Peter B. Kessler and Marshall K. McKusick. "Gprof: A Call Graph Execution Profiler" *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction* pp. 120-126 (1982).

[4] Bob Cmelik and David Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling" *Proceedings of the 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems* pp. 128-137 (1994).

[5] Gary S. Tyson and Todd M. Austin, "Improving the Accuracy and Performance of Memory Communication Through Renaming" *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO)*, pp. 218-227 (1997).

[6] Doug Burger and Todd M. Austin, "The SimpleScalar Tool Set, Version 2.0," Technical Report #1342, Computer Sciences Department, University of Wisconsin (Jun 1997).