USING STATIC PROGRAM ANALYSIS TO COMPILE FAST EXECUTION-DRIVEN SIMULATORS

Vesa Hirvisalo
Helsinki University of Technology
Laboratory of Information Processing Science
P.O. Box 5400, FIN-02015 HUT, Finland
E-mail: Vesa.Hirvisalo@hut.fi

KEYWORDS

Simulation methodology, cache simulation, simulation optimization, execution-driven simulation.

ABSTRACT

This paper presents a generic approach for compiling fast execution-driven simulators, and applies the approach to simulating the effects of program execution in computer hardware. Our approach is based on using static program analysis to guide partial evaluation and slicing of simulators. Because results of some simulation operations are known before execution, a cache simulator program can be partially evaluated during its compilation. Program slicing can be used to remove the computations that have no effect on the simulation result.

Our experimental work with cache analysis shows that our approach significantly speeds up simulations. Fast cache simulation is needed in development of both computer software and hardware. To properly understand the cache behavior caused by a computer program, simulations must be done with sufficiently many inputs. Traditional simulation of memory operations caused by a computer program can be orders of magnitude slower than execution of the program. Our approach reduces the time needed in cache performance evaluations without losing accuracy of the results.

INTRODUCTION

This paper discusses how static program analysis can be used to compile fast execution-driven simulators. In an execution-driven simulation, the execution of a computer program is interleaved with the simulation that describes behavior implied by the computer program.

Execution-driven simulation is a straightforward approach to analysis of computer programs. It is performed by simply executing a subject program with instrumentations that collect the analysis data and simulating the behavior that we are interested in.

Static program analysis is an other approach to analysis of computer programs. In static analysis, we try to understand the run-time behavior of a program without executing it with a specific input. Static analysis is usually motivated by its ability to simultaneously give results for a set of inputs, often for all inputs of a program.

For some analysis tasks, simulation methods are slow, and any speed-up of simulation is useful. Our approach is to use partial evaluation and program slicing guided by static analysis to compile fast simulators.

In our presentation, we concentrate on a specific application: simulating the cache memory behavior that is caused by an execution of a computer program. Such cache performance evaluation is a demanding program analysis task, and thus a good test of the applicability of our approach.

Understanding memory performance of computer programs is hard. The steps executed and the related memory references can be seen from the code of a program. However, cache misses and the related executions stalls cannot be seen. Typical hardware does not support analysis of memory operations (Horowitz et al. 1996). Therefore, the memory operations caused by program execution are often simulated.

The traditional cache analysis method is trace-driven simulation (Uhlig and Mudge 1997). A trace-driven simulation has two main phases. In the first phase, an access trace is collected. Because hardware support for tracing is rare (Horowitz et al. 1996), the collection is typically done by augmenting the subject program with trace emitting code. In the second phase, a memory simulation is executed using the collected trace as the input.

Execution-driven simulation can yield more accurate results than trace-driven simulation. In an execution-driven simulation, the execution of a software is interleaved with the simulation of the underlying hardware. Thus, execution-driven simulation allows feed-back from the hardware simulator to the software. Such a simulation technique is useful in program performance analysis and simulating parallel and distributed systems.

Simulation is a flexible and accurate method, but simulating memory operations of a program can be orders of mag-

Proceedings 16th European Simulation Symposium
György Lipovszki, István Molnár © SCS Press, 2004
ISBN 1-56555-286-5(book) / ISBN 1-84233-106-x(CD)

nitude slower than execution of the program (Uhlig and Mudge 1997). To properly understand the memory behavior of the program, simulation must be done with sufficiently many inputs. This leads to simulation times that are often infeasible in system development.

Because of the central role of simulation in cache performance analysis, a variety of methods have been developed for speeding up simulations. Many of the methods are designed for trace-driven simulation. The traditional speed up methods operate between the trace generation and the trace consumption (i.e., simulation). They modify the trace in a way that makes the processing faster. Such method include, for example, packing methods (Ha and Johnson 1994, Samples 1989), stack deletion methods (Smith 1977), cache filtering methods (Puzak 1985), and spatial blocking (Agarwal and Huffman 1990).

Recent methods for speeding up cache simulations often apply parallel and distributed techniques. Such methods divide the simulation task into a number of smaller subtasks that can be simulated in parallel by a number of processors. Such methods include, for example, set and time partitioning (Heidelberg and Stone 1990), stack distance methods (Nicol and Greenberg 1992), and methods based on applying generic parallel discrete event simulation techniques (Fujimoto 1999).

Our method for speeding up simulations is based on static program analysis. Our static analysis finds out memory references that always cause cache hits or always cause cache misses. Based on the static analysis, parts of the simulation task can be solved during compilation of an execution-driven simulation program. This is done by omitting simulation at references, whose effects are statically known, and using simplified simulation at references, whose effects are partially known.

The structure of this paper is the following. The second section discusses the problem of cache performance analysis of programs. The third section presents a model of execution driven simulation. The fourth section discusses static cache analysis, and the fifth section describes how it can be used to compile fast cache simulators. The sixth section presents our experimental results, which show that significant speed up can be achieved by using the method. In the last section, we draw some conclusions.

CACHE MEMORIES

Cache memories (see e.g., Przybylski 1990) improve memory access times. They reduce the number of cycles a processor is waiting for data; in the best case, the processor can continue its operation without any stall. Present day first level caches can give access to data two orders of magnitude faster than main memory. Thus, memory can become a major performance bottleneck, and careful design can significantly improve performance of systems using cache memory.

Cache memories consist of blocks called *cache lines*, which are used to store frequently used blocks of memory. We denote the length of a cache line by L. Cache lines are organized into *cache sets* of equal size. The size of a cache set is also called *associativity* of the cache (typically, it is 1–16 lines), which we denote by A. We denote the number of cache sets by N. Thus, the size of a cache is $N \cdot A \cdot L$.

A *memory line* is an aligned block in memory that is of the size of a cache line. Each memory line (and thus also each memory address) is uniquely mapped to a set. Two memory accesses (or references[1]) *conflict* if their addresses are mapped to the same cache set.

Before accessing the main memory, the computer hardware checks whether the addressed datum is stored in a cache line (in the cache set of the memory address). If the datum is in the cache, then a *hit* occurs. Otherwise, a *miss* occurs and a block in the cache is replaced for the new one. The misses can be categorized into three:

- *Compulsory misses*, the first access to a line causes always a miss.
- *Capacity misses* occur when the cache is too small to hold all of the lines needed during an execution of a program.
- *Conflict misses* occur when the cache has sufficient space, but the organization of the cache does not allow the data to be kept in the cache.

Because of the complexity of the memory hardware, interactions of memory references are complex. To improve performance, we must understand the cache behavior that the references cause. Typical hardware does not support analysis of memory operations (Horowitz et al. 1996). Therefore, the memory operations of such programs are usually simulated.

Simulation of memory operations of a program can be orders of magnitude slower than execution of the program (Uhlig and Mudge 1997). Traditional simulation of the effect caused by a single access includes several operations:

- pass the accessed address to the simulator
- break up the address into tag, block number, and block offset
- compute the set number
- search the block in the corresponding set
- update set status and performance metrics

Updating the set status typically consists of several operations, which depend on the cache replacement algorithm that is used in the simulated hardware. In the following, we will assume LRU (Least Recently Used) replacement.

[1] A static read or write in a program is a *reference*. An execution of the read or write at runtime is an *access*.

EXECUTION-DRIVEN SIMULATION

In an execution-driven simulation, the execution of a program is interleaved with the simulation of the underlying hardware. In the rest of this paper, we will use an abstract model of execution-driven simulation to explain our method of building fast simulators.

Let P be a program and O its output corresponding to input I, i.e., the program computes:

$$[\![P]\!]\langle I\rangle = O \tag{1}$$

To build a simulator that is driven by P, we must instrument P with code that simulates the hardware. The instrumented program P^A computes:

$$[\![P^A]\!]\langle I, I^+\rangle = \langle O, O^+\rangle \tag{2}$$

where I^+ is the input for the instrumentation and O^+ is the analysis output that is measured by the instrumentation.

In an execution-driven cache simulator, each memory reference in the original program code must be instrumented with cache simulation code. Instead of explicitly giving the instrumentation code that is needed, we define a short abstract instrumentation for cache analysis.

The abstract instrumentation uses a mapping `incache` and a set of queues $\mathtt{Q}_t$ (one queue per each cache set). For each cached line t, `incache`(t) is 1. Otherwise `incache`(t) is 0. For each cache line t, Q_t is the replacement queue of the set of the line. The queues represent the total order needed by LRU management – they are last-in-first-out queues. For each cache set, the head of its queue is the least recently used cache line.

Analysis instrumentation for a memory reference addressing line *l* is:

```
t = line(l, N, A, L);
if not incache(t) then
   set_not_incache(remove_head(Q_t))
   set_incache(t)
else
   remove(Q_t, t)
end
insert_tail(Q_t, t)
```

In the instrumentation, *line* returns the memory line referred to, *remove_head* removes the head of a queue and returns it, *insert_tail* inserts a cache line to the tail of a queue, and *remove* removes a cache line from the inside of a queue.

STATIC ANALYSIS

In static analysis, we estimate the execution state of a program without actually executing the program. Our static analysis follows the concept of abstract interpretation presented by (Cousot and Cousot 1977). The concept is a formalization of flow analyses used in many optimizing compilers.

To understand the hits and misses, our static cache analysis approximates concrete cache states, which are simulated by the instrumentation presented in the previous section. In the approximation, we use *cache ages* that describe, how recently a data element has been referred to. The age of an element can be $\{1, ..., A, \top\}$, where $\top$ means that the element is not in cache.

At each program point, we statically compute the upper and lower bounds for cache ages of data elements. I.e., we approximately compute the position of each data element in the Q_t queues described in the previous section. In *must* analysis of cache states, data elements are mapped to their maximum cache age. In *may* analysis of cache states, data elements are mapped to their minimum cache age. Memory references that always hit are found by *must* analysis and memory references that always miss are found by *may* analysis:

- If the age of a referred to data element is always less than or equal to A, then the reference is always a hit.
- If the age of a referred to data element is newer less than or equal to A, then the reference is always a miss.
- Otherwise, we do not know.

For example, let associativity of a cache be two. Consider the following piece of code using four pointer variables a, b, c, and d that point to different memory lines belonging to a single cache set.

```
y := *b
x := *a
if x > y then
   x := *c
if x > y then
   y := *a
else
   y := *d
```

Consider state of the cache before the second **if** statement. In our *must* analysis the abstract state is $\{(\hat{a}, 2), (\hat{b}, \top), (\hat{c}, \top), (\hat{d}, \top)\}$. Thus, the maximum age of the data element pointed by a is 2. For other locations, it is unlimited, i.e., $\top$. In our *may* analysis, the abstract state is $\{(\hat{a}, 1), (\hat{b}, 2), (\hat{c}, 1), (\hat{d}, \top)\}$, i.e., three locations may be cached. Thus in the second **if** statement, the memory reference using pointer a is always a hit and the memory reference using pointer d is always a miss.

COMPILING FAST SIMULATORS

Our combined analysis has three phases: a compilation phase, an execution phase, and a summary phase. In the compilation phase, a subject program is statically analyzed and a simulator is built. In the execution phase, the simulator is executed (typically with several inputs). In the summary phase, the analysis information of the compilation phase and the analysis information of the simulation phase are combined.

The execution phase follows the typical procedures of simulation studies. The summary phase simply merges the results of the static analysis and simulation. The compilation phase is special, therefore, we will describe it in detail.

The compilation phase consists of three program transformation steps: program instrumentation, partial evaluation, and program slicing. The first step creates an execution-driven simulator and the last two are program specializations, which make it faster and more compact. The two specialization steps need static analysis information (i.e., results of the analysis explained in the previous section).

Partial evaluation

Partial evaluation (see e.g., Jones et al. 1993) is a program transformation that is given a subject program with part of its input data. It constructs a new program that, when given the remaining input, will yield the same result that the original subject program would have produced given both inputs.

Consider the program P^A computing $[\![P^A]\!]\langle I, I^+\rangle = \langle O, O^+\rangle$. Let *peval* denote the partial evaluator, then

$$[\![peval]\!]\langle P^A, I^+\rangle = P^A_{I^+} \Rightarrow [\![P^A]\!]\langle I, I^+\rangle = [\![P^A_{I^+}]\!]\langle I\rangle$$

for all I. Thus, we fix the analysis initialization (input) and evaluate statically the part of the analysis that we can.

Our partial evaluation is based on static analysis of the original program P. The static analysis is done for the same task as the preceding instrumentation for the simulation (which produced P^A from P). Static analysis can give us static values of both the original program and its instrumentation.

Instead of using the complete analysis instrumentation given in the third section, we can use simplified instrumentations at references that are known to hit or miss, i.e., only one branch of the **if** statement is used and the condition is omitted. Further, we can use a faster operation *line'(l)* instead of the complete operation *line(l, N, A, L)*, because *N*, *A*, and L are constants.

Slicing

Program slicing (see e.g., Tip 1995) is an operation that identifies semantically meaningful decompositions of programs. Usually, two kinds of slices are identified:

- A *backward slice* of a program P with set of program elements S consists of all program elements that might affect the values computed by S.
- A *forward slice* of program P with set of program elements S consists of all program elements that might be affected by the values computed by members of S.

S is called the *slicing criterion*. In our analysis, the slicing criterion consists of input statements of the original program, and output statements of the analysis instrumentation. We use our slicing criterion to compute a backward slice of the simulator.

As described in the preceding, a partially evaluated analyzer computes both the original output and the analysis output:

$$[\![P^A_{I^+}]\!]\langle I\rangle = \langle O, O^+\rangle$$

We do not need the original output. Further, we do not need the program elements that do not affect our analysis result.

We use slicing to implement a program transformation that yields a program $P^A_{I^+S}$ computing only the analysis output. Let $slicer$ be the transformation, then for all I:

$$[\![\textit{slicer}]\!]\langle P^A_{I^+}, S\rangle = P^A_{I^+S} \Rightarrow$$
$$[\![P^A_{I^+S}]\!]\langle I\rangle \overset{O^+}{=} [\![P^A_{I^+}]\!]\langle I\rangle \;\&\; [\![P^A_{I^+S}]\!]\langle I\rangle \overset{O}{=} \emptyset$$

where S is the slicing criterion, $\overset{O^+}{=}$ denotes equality of analysis output, and $\overset{O}{=}$ denotes equality of original output.

Program slicing is done by analyzing relations between program elements. Program elements do computations by using values to define new ones or to control program flow. If a value or a flow of control is not used, then the elements defining the value or controlling the flow can be removed.

Optimizing compilers typically apply simple forms of program slicing, e.g., dead code elimination. Dead code elimination removes program elements that compute unused values. Analyzing control dependencies caused by jumps and especially subroutine calls is more complex. Compiler rarely do such analysis.

Consider the analysis instrumentation of our cache simulator. A hit removes a cache line from the replacement queue. If we known that a reference is always a hit, then it must be preceded by a reference that places the cache line in the queue. Thus, we have a pair of the form:

```
insert_tail(Q_t, t)
...
remove(Q_t, t)
```

Actually, there is no need to insert the cache line in the replacement queue, because of the following hit. We know that the inserted line will never reach the head of the queue. The insertion and the related removal can sliced away.

To cope with undetected pointer aliasing we use a counter of pending insertions for each cache line instead of the simple *incache* flag. The slicing can be complicated by the branching structure of a program. We can slice away only those insertions that are definitely followed by a hit. By using such a conservative approach, the sliced simulator computes the same cache behavior for a program than the original simulator.

Such a code transformation may seem insignificant. However, most of references in a typical program are hits, and sequences of hits are common. (That is the reason why cache memories are efficient.) For such re-hits, both the removal and the insertion are sliced away, and thus, also the computation needed to identify the line is not needed. No instrumentation remains after the slicing at such references.

The slicing can proceed even further. If there is no instrumentation at a memory reference, it is possible that there is no need to compute the reference. Thus, it becomes possible to slice away parts of the original program that is driving the simulation.

EXPERIMENTAL RESULTS

We experimented with our method to show its potential. We found out in our experiments that a relatively simple static analysis is sufficient to yield significant speed up of simulations. In the following, we describe our experiments that were solely based on *must* analysis of abstract cache state.

Instead of considering some specific hardware, we analyzed the operation of a generic load/store architecture machine, which we call SM (Simple Machine). SM is a register machine with a simple instruction set. The main feature of SM is that its memory system can be parameterized to conform with various memory configurations. In the context of cache analysis, only instructions addressing the memory are significant. Therefore, SM is a representative for a large set of machines using the load/store architecture.

In our experiments, we used a tool set called MSE (Memory Simulation Environment) (Hirvisalo 2004). MSE has a compiler that generates code for SM. We used the compiler to compile three applications that were written in C language:

- `di`: A message dispatcher, which receives messages, decodes them and routes them further. The decoding and routing is implemented hard coded. Addresses of most memory references are dynamically computed.
- `da`: A relational database application, whose index is implemented as an unbalanced binary tree. Addresses of most memory references are dynamically computed.
- `co`: A control application, which operates like a device driver. The data structures of the application are mostly static.

We did two kinds of experiments, which we call static and dynamic experiments. In both experiments, we used two tools: one that built traditional simulators (as explained in the third section) and one that built specialized simulators (as explained in the previous section). In static experiments, we measured and compared the simulators. In dynamic experiments, we measured and compared their execution.

appl.	associativity=1		associativity=16	
	static solution	speed incr.	static solution	speed incr.
`di`	15%	25%	52%	90%
`da`	60%	110%	62%	125%
`co`	42%	60%	70%	175%

The performance of the method depends mostly on the target program and associativity of the cache. Our static cache analysis could classify 15% to 70% of the memory references. The performance of the static analysis depended on the dynamism of the addressing and on the interleaving of the memory references. Accesses of `Database` are more local than accesses of `Dispatcher`. They both use dynamic addressing, but `Control` uses static addressing. The actual (i.e., simulated) cache hit ratio for all the applications was typically around 90%.

We used a Pentium computer to run the dynamic experiments. The specialized simulators were 25% to 175% times faster than the original simulators. The speed-up is mostly caused by slicing. The direct effect of partial evaluation is minor, but it makes the slicing possible by removing branching in the instrumentations.

CONCLUSIONS

This paper presented an approach for building fast simulators. The approach combines simulation and static program analysis: it uses simulation to fill in the results of static analysis and static analysis to speed up simulation.

We use two kinds of program specialization in speeding up simulation: partial evaluation and program slicing. They supplement each other: partial evaluation works forwards and slicing works backwards in the flow of control of a program.

Our approach is an abstract one. This leaves several options available in implementing a specific analysis. The instrumentations needed in simulation depend on the analysis to be done. Also, the instrumentation mechanism can differ. The same is true for the static analysis suggested. Abstract interpretation gives a theoretical framework for static analysis. Several implementations are possible within the framework, e.g., the work-list algorithm (see e.g., Nielson et al. 1999).

Alternatives exists also for implementing partial evaluation and program slicing. For example, simple methods like

constant folding or complex methods like polyvariant specialization (Jones et al. 1993) can be used in partial evaluation. In program slicing, there exists methods, which are based on data-flow equations, information-flow relations, and dependence graphs (Tip 1995).

As an application of the generic approach we discussed cache performance evaluation. Many processors have counters for cache misses and hits. Cache performance can be directly measured by using them. However, it is very hard to link those results with the program code and data structures. Thus, simulation has usually been the choice for performance studies and the related tools giving detailed information (e.g., Martonosi and Ga 1992).

Cache simulations can be speeded up, because most of memory references in a typical program always cause cache hits. The approach can be combined with parallel and distributed simulation methods. Our experimental results show that significantly faster analysis is achieved by using our approach.

REFERENCES

A. Agarwal and M. Huffman, 1990. "Blocking: Exploiting Spatial Locality for Trace Compaction." In *Proceedings of 90' ACM SIGMETRICS*, Performance Evaluation Review, 18(1), pages 48–57.

P. Cousot and R. Cousot, 1977. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints." In *Fourth ACM Symposium on Principles of Programming Language*, pages 238–252, Los Angeles, California. ACM Press, New York.

R. Fujimoto, 1999. "Parallel and Distributed Simulation." In *Proceedins of 1999 Winter Simulation Conference*, pages 122–131.

J. Ha and E.E. Johnson, 1994. "PDATS: Lossless Address Trace Compression for Reducing File Size and Access Time." In *Proceedings of 1994 IEEE International Conference on Computers and Communications*. IEEE.

P. Heidelberg and H. Stone, 1990. "Parallel Trace-Driven Simulation by Time Partitioning." In *Proceedings of 1990 Winter Simulation Conference*, pages 734–737.

V. Hirvisalo, 2004. *Using Static Program Analysis to Compile Fast Cache Simulators*. Helsinki University of Technology, Laboratory of Information Processing Science, Espoo, Otaniemi. PhD thesis.

M. Horowitz, M. Martonosi, T.C. Mowry, and M.D. Smith, 1996. "Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors." In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 260–270.

N.D. Jones, C.K. Gomard, and P. Sestoft, 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York.

M. Martonosi, A. Gupta, and T. Anderson, 1992. "MemSpy: Analyzing Memory System Bottlenecks in Programs." In *Proceedings of the 1992 SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 1–12.

D.M. Nicol, A.G. Greenberg, and B.D. Lubachevsky, 1994. "Massively Parallel Algorithms for Trace-Driven Cache Simulations." *IEEE Transactions on Parallel and Distributed Systems*, 5(8):849–859.

F. Nielson, H.R. Nielson, and C. Hankin, 1999. *Principles of Program Analysis*. Springer.

S.A. Przybylski, 1990. *Cache and Memory Hierarchy Design*. Morgan Kaufmann Publishers, Inc., Palo Alto.

T.R. Puzak, 1985. *Analysis of Cache Replacement Algorithms*. University of Massachusetts, Department of Electrical and Computer Engineering. PhD thesis.

A.D. Samples, 1989. "Mache: No-Loss Trace Compaction." In *Proceedings of the Sigmetrics 89 Conference*, pages 89–97.

A.J. Smith, 1977. "Two Methods for the Efficient Analysis of Memory Address Trace Data." *IEEE Transaction on Software Engineering*, SE-3(1).

F. Tip, 1995. "A Survey of Program Slicing Techniques." *Journal of programming languages*, 3(3):121–189.

R.A. Uhlig and T.N. Mudge, 1997. "Trace-driven Memory Simulations: A Survey." *ACM Computing Surveys*, 29(2):128–170.