Method for the deferred materialization of condition code information

M. K. Gschwind

Disclosed is a translation approach to reduce the number of instructions which need to be executed by a binary translation system. More specifically, the approach is designed to allow accurate 100% system level binary translation to make use of control flow and live range analysis while preserving complete accuracy in the presence of unpredictable exceptions in full system binary translation.

To maintain a consistent architectural state, emulation and binary translation systems must compute all observable data values computed by an input program, even if these values may never be accessed subsequently. For many architectures, this information includes condition code values which may be implicitly set by all or most instructions of an architecture. (Examples of such architectures are IBM System/390, DEC VAX and Intel x86.)

This incurs computation overhead to compute unused data values. In addition, due to system-specific nature of condition code setting, significant work may have to be expended to compute the condition codes accurately on another system.

While liveness information can be used to identify unused condition code computation, this information is necessarily incomplete for full system simulation. More particularly, such liveness analysis cannot detect control flow which may occur due to exception or trap operations, e.g., due to page faults.

Thus, while liveness information can be used to eliminate the computation of condition code information when problem state programs from one instruction set architecture to another (e.g., the VEST and Wabi translators), such information cannot be readily used by full system simulation.

However, binary translators face a problem when encountering a potential exit from a translation unit, and it is unknown whether the condition code will be used in the successor group. This is addressed by dynamic deferred condition code materialization, where instructions store the operands and the source values in some storage. The assumption is that executing instructions to store these values is cheaper than actually computing the correct condition codes [3].

The problem is somewhat different on an architecture designed to efficiently support binary translation from one or more source architectures to a target architecture. However, when performing full system simulation as described in [1][2], computations of dead condition codes cannot be suppressed. As a result, a significant number of condition code computations may be required in the generated code. These computations are usually useless, but required to maintain the accurate processor state in the case of exceptions.

The proposed method uses static information to perform deferred materialization of condition codes which are dead within a translation group but which may be required in the case of an exception (e.g., due to a page fault). Unlike dynamic deferred materialization, where instruction record the source values and operation which generated the unmaterialized condition code, static deferred materialization uses static tables to derive such information. Using multiple instructions

appeared in: Research Disclosures, No. 431

to record source and operations would be more expensive than using a single computational instruction already provided in the instruction set.

To ensure that the source values necessary to materialize the condition code are accessible in the case of an exception, the live ranges of source registers have to be extended to the last point were the materialization of the condition code may be necessary. This is preferably performed in conjunction with a register renaming scheme to eliminate the cost of live range extension for registers whose live range is extended. (Extending live ranges with explicit register copies is similar in cost to dynamically deferred materialization of condition codes.)

A static table describes for every possible exception point how to materialize the condition codes. Thus, at any given long instruction word address, only a single materialization rule can apply. These rules can be recorded statically in a condition code materialization table, and source for materialization's live ranges would then be extended to the farthest reaching point were they are needed for materialization of the condition code.

To deal with control flow joins, condition codes are materialized on leaving a translation group. Group exits usually reference the condition codes, so materialization is usually necessary anyway. And in a binary translation target architecture, appropriate support for efficiently computing condition values ensures that full materialization of condition codes can be performed efficiently.

The following table shows how full materialization, dynamically deferred materialization in problem state binary translation and statically deferred materialization in system level binary translation compute condition code values:

	native exec.	system level transl.	problem state transl.
cc materialization	full	statically deferred	dynamically deferred
cc used in unit	compute	compute	compute
cc live on unit exit	compute	compute	record dynamically
no visible use of cc	compute	record in static table	do not compute
exception occurs	cc exists	cc can be generated	not modeled

Consider the following example translation of an original architecture code fragment to a DAISY architecture:

original architecture instructions	DAISY architecture instructions	CC information
		entry, cc materialized in cr0
A r4, 4(r6)	LD rt, 4(r6)	
	ADD r4, r4, rt	cc = r4 + rt
A r4, 8(r6)	LD rt', 8(r6)	
	ADD r4, r4, rt'	cc = r4 + rt'
	cc_add cr0, r4, rt'	cc materialized in cr0
ST r4, 0(r6)	ST r4, 0(r6)	
BLT label	Bcond cr0.lt, label	

The translation of the code on the left hand side to the DAISY architecture on the right hand side consists of actual target instructions, as well as condition code materialization information. When the code is entered, the condition code has been materialized in cr0. The first CISC instruction performs a memory-to-register add, and sets the condition code accordingly. This is translated by a load operation followed by an addition operation. The condition code is not computed since no

use of it is visible. The only potential use of such condition code would be an exception handler, which could be invoked by a page fault in the second A operation. To reconstruct the condition code in this case, information on how to recreate the condition code (cc = r4 + rt) is recorded. The second A operation of the original code is executed similarly, but since the condition code is used by the subsequent BLT conditional branch and may be live at group exit, it is materialized by a 'cc_add' (set condition code according to the result of an addition). This example is typical, in that condition code computation will be necessary at potential program group exit points as source for conditional branches. Thus, materialization of condition codes at potential group exits often does not incur any additional costs. (The ST instruction does not modify the condition code in the source architecture, so it remains unchanged in the target translation.)

To summarize, the present disclosure describes a method to defer the materialization of unused condition codes within a translation group. Static information in conjunction with appropriately extended live ranges of source registers used in the computation of condition code information can be used when encountering exceptions to materialize condition codes. Control flow joins are handled by always deferring condition code information at such points. The present approach eliminates the cost of using instructions for computing unused condition code information, recording information in compressible static tables which associate instructions for materializing condition codes when they are needed to create the correct exception entry processor state.

References

[1] K. Ebcioglu, E. Altman, DAISY: Dynamic compilation for 100% architectural compatibility, IBM Research Report 8052, Yorktown Heights, NY, 1996.

[2] K. Ebcioglu, E. Altman, S. Sathaye, M. Gschwind, Execution-based Scheduling for VLIW Architectures submitted to: EuroPar '99, 1999.

[3] Paul Hohensee, Mat Myszewski, David Reese, Wabi Cpu Emulation, Hot Chips VIII, 1996.