RC 21900 (98525) 5 December 2000 **Computer Science**

IBM Research Report

On Achieving Precise Exceptions Semantics in Dynamic Optimization

Michael Gschwind, Erik Altman
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

On Achieving Precise Exception Semantics in Dynamic Optimization

Michael Gschwind, Erik Altman

IBM T.J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

Maintaining precise exceptions is an important aspect of achieving full compatibility with a legacy architecture. While asynchronous exceptions can be deferred to an appropriate boundary in the code, synchronous exceptions must be taken when they occur. This introduces uncertainty into liveness analysis since processor state that is otherwise dead may be exposed when an exception handler is invoked. Previous systems either had to sacrifice full compatibility to achieve more freedom to perform optimization, use less aggressive optimization or rely on hardware support.

In this work, we demonstrate how aggressive optimization can be used in conjunction with dynamic compilation without the need for specialized hardware. The approach is based on maintaining enough state to recompute the processor state when an unpredicted event such as a synchronous exception may make otherwise dead processor state visible. The transformations necessary to preserve precise exception capability can be performed in linear time.

1 Introduction

Dynamic compilation is a powerful technique to optimize programs based on execution behavior and to respond to changes in the execution profile. Dynamic optimization can be used either as a technique in its own right, or in combination with binary translation techniques.

Dynamic optimization includes techniques to perform code layout for improved memory behavior, optimize frequently executed program paths, speculatively execute instructions or use value prediction [7, 6, 16, 5, 4, 3, 10].

A number of other optimization techniques are also highly effective in conjunction with dynamic optimization by exploiting runtime program profile data, such as dead code elimination, code sinking, unspeculation or partial redundancy elimination [9]. These techniques are even more useful for binary translation where the original ISA may cause the program to compute extraneous state which is hard to emulate and unnecessary, such as the computation of condition codes as a side effect of every instruction [11].

To produce correct execution behavior, dynamic optimization has to be conservative in analyzing and optimizing programs. In particular, the visible state of the program has to match the state of the unoptimized program at any point during program execution. This requirement imposes significant restrictions on the types of optimizations which can be performed without impacting program correctness, because synchronous interrupts can expose parts of the state that are otherwise invisible.

In the DAISY dynamic translation project we found that on a 4-wide machine running the **SPECint95** benchmarks, ILP can be reduced by up to 18%, and by an average of 10% by the requirement that every (possibly dead) result be placed in the architected register of the source architecture and further that each result be placed in the architected register in order [2].

Consider the following code sequence:

```
1         add r4,r3,r4
2         lwz r3,0(r9)
3         add r4,r3,r3
```

Clearly, the instruction at line 1 is dead, but a page fault caused by the load instruction at line 2 could make the value of `r4` visible to the exception or signal handler. In many cases, the only action taken by the handler may be to store and restore

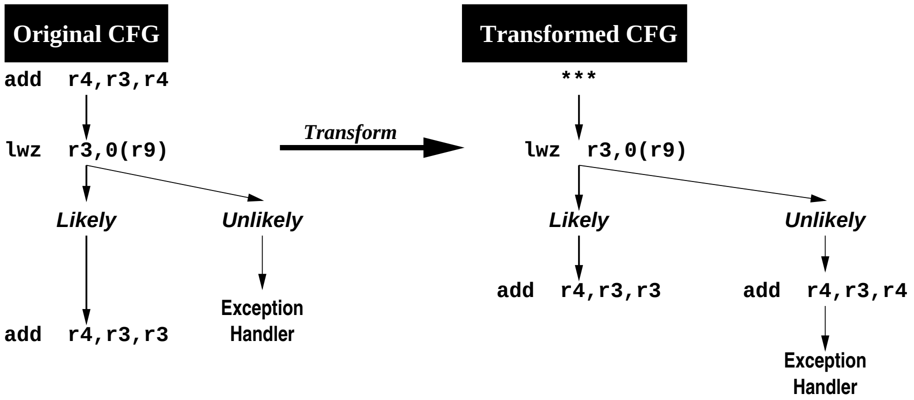

Figure 1. Control Flow Graph Transformation for Repair Code.

the value in `r4`, but if the handler bases any actions on the values stored in register `r4`, the program may fail. Thus, many dynamic optimizers have either severely restricted the amount of dead program state computation which can be eliminated [7]. Some dynamic optimizers have included a ‘safe mode’ which disables such optimizations [4, 3], but this is undesirable since this approach (1) requires to identify which program rely on extensive program state analysis in their exception handler, and (2) such programs are over their entire execution, even if no exception ever occurs.

In this work, we present a solution to allowing dead state eliminating techniques during dynamic optimization while retaining exact program behavior. In particular, this approach is based upon deferring materialization of otherwise dead code to the few instances where its results may be accessed by a synchronous exception handler. This is achieved by invoking a repair function provided by the dynamic optimizer environment which repairs the state of the program before actually passing control to a native synchronous exception handler (or its translation, in dynamic binary translation).

Dynamic compilation is key to efficiently implementing this technique and taking full advantage of it with other optimizations. A static compiler faces an exponential growth in fixup code as operations are speculatively moved past multiple branches, while a dynamic compiler only generates these fragments in the (rare) event they are actually needed.

This paper is structured as follows: we give an overview of the basic approach in Section 2. We present a sample algorithm for the elimination of dead code in Section 3 and discuss applications to other optimization techniques such instruction scheduling and unspeculation in Section 4. We describe program state repair in Section 5. We present initial results in Section 6. We discuss related work in Section 7 and draw our conclusions in Section 8.

2 Basic Approach

The technique at the heart of this approach is annotation of generated code to allow a native exception handler to repair the state of the program to reflect the in-order state at any point in program execution where a synchronous exception can arise.

Considering the example from the Introduction, we note that when an exception does not occur, eliminating instruction 1 would be a legal transformation. By introducing a repair step before the transfer of control to the exception handler, a legal code sequence can be achieved. Consider the following code, which has eliminated instruction 1, but annotated instruction 2 with repair actions to perform before control is passed to the exception handler.

```
1  ***   on exception, repair: r4 = r3+r4
2  lwz r3,0(r9)
3  add r4,r3,r3
```

Thus, when instruction 2 raises an exception, the repair actions will restore the value of `r4` to that seen in the original program, but otherwise an instruction has been eliminated.

This corresponds to the control flow graph (CFG) transformation in Figure 1 if the exception handler is viewed as a branch in the control flow graph which is arguably correct.

```
1  foreach operation op
2      if dead ( target (op) )
3          convert2repairnote (op);        %% Deletes op and inserts as repair note
4          foreach instruction killing target (op)
5              insert_use ( target (op) )
6              insert_equivalence ( target (op) == sources (op) )
```

Figure 2. Basic Algorithm.

To make program transformations based on dead state eliminating techniques safe for use in dynamic optimization, several steps are necessary. During the optimization phase, enough information must be retained to regenerate eliminated state. This includes information both about the operations which were eliminated, as well as preserving the input values feeding the operation.

When code is emitted, information about the eliminated computations has to be emitted into the translation cache so it can later be used by the repair mechanism. And, finally, when an exception occurs, a repair function must interpret the information about eliminated state and recompute it so as to restore the entire program state before control is passed to a translation of the native exception handler.

3 Algorithm for Dead Code Elimination

Our algorithm for these optimizations is best demonstrated for the simplest case, dead code elimination. While dead code elimination is not very useful for a properly optimized program in the context of dynamic optimization, many optimizations can be reformulated as having precise exception semantics by leaving the original operations in place as dead operations computing values solely for the purpose of maintaining precise exception state. Dead code elimination can then be used to eliminate these operations. Also, dead code elimination is extremely useful when used in conjunction with binary translation where it can be used to eliminate extraneous state introduced by the ISA, e.g., by condition code setting instructions in CISC ISAs such as the Intel *x86* or IBM *System/390* [11].

We will assume that the original program representation has been converted to an *Internal Representation* (IR) which has a single result value per operation. In the case of instructions with multiple results (such as compute and set condition code instructions), a machine instruction will be represented by multiple IR operations. We also assume that the IR is in SSA form.[1]

Our algorithm uses a register equivalence list for liveness analysis and register allocation, to ensure that input values of eliminated instructions will be available if they are needed to compute the exact program state.

We will denote a live-range register equivalence as $s_3 \equiv \langle s_4, s_7 \rangle$ indicating that at any point in the IR that a symbolic register s_3 is mentioned, symbolic registers s_4 and s_7 are to be considered live as well for the purpose of register allocation.

The algorithm iterates over an operation list representing a single translation group, and finds operations with a dead result. These operations can be eliminated, provided their result can be reconstructed in the event of an exception. To ensure this, the algorithm adds a use of the dead target symbolic register name after the instruction killing the result[2], and adds an entry to the register equivalence list which equates the dead result symbolic register to the symbolic input registers of the dead operation.[3] These two steps ensure that all input registers of deleted operations are live to the latest point where the target register may be live. Then, the dead instruction is removed and replaced by a repair note in the IR. (The difference between an actual instruction and a repair note can be a single bit flag field in the IR structure.)

This yields the algorithm in Figure 2, which is linear ($O(N)$ where N is the number of instructions in a CFG) in both time and the size of data structures. This algorithm can successfully deal in a single pass with a group of dead instructions which are dependent on each other, provided the liveness check at line 2 is transitive, i.e., a source register to any instruction is only

[1] Most dynamic compilers work on basic blocks or extended basic blocks, so this transformation is straightforward.

[2] The use node represents the use along the exception control flow. Since there are no instructions along that path the uses along that arc can be folded into the mainline control flow at the conceptual control flow split at exception raising instructions.

[3] The use node for the original register serves a dual function – it represents the minimum range of validity of the repair note, and consequently how long the input registers need to be available. If some output **rX** of a repair note *A* **rX = r1** *OP* **r2** is required as input of another repair note *B* **rZ = rX** *OP* **r3**, this will extend the live range of **rX**, and, because register live range equivalence is transitive, also **r1** and **r2**.

```
and.  r4,r3,r4  | 1   s4′  =  s3 & s4         | 1 {    s4′  =  s3 & s4 }
                | 2  sc0′  = (s3 & s4) cmp 0  | 2 {   sc0′  = (s3 & s4) cmp 0 }
lwz   r3,0(r9)  | 3   s3′  = [s9]             | 3      s3′  = [s9]
add   r4,r3,r3  | 4   s4′′ =  s3′ + s3′       | 4      s4′′ =  s3′ + s3′
                |                             |   use  s4′  ;  s4′ == < s3,s4 >
addi  r5,r3,80  | 5   s5′  =  s3′ + 80        | 5 {    s5′  =  s3′ + 80 }
lwz   r3,0(r10) | 6   s3′′ = [s10]            | 6      s3′′ = [s10]
addi. r5,r3,1   | 7   s5′′ =  s3′′ + 1        | 7      s5′′ =  s3′′ + 1
                |                             |   use  s5′  ;  s5′ == < s3′ >
                | 8  sc0′′ = (s3′′ + 1) cmp 0 | 8     sc0′′ = (s3′′ + 1) cmp 0
                |                             |   use sc0′  ; sc0′ == < s3,s4 >
                |                             |
    PowerPC     |           INITIAL           |  INTERMEDIATE REPRESENTATION
 ASSEMBLY CODE  | INTERMEDIATE REPRESENTATION |      AFTER ANNOTATION
      (a)       |             (b)             |            (c)
```

Figure 3. Example: *PowerPC* **Destination registers are at left. A "." after an operation means set** *condition register 0* **by comparing the result to 0.**

live if its output is live. The transitive closure of live and dead values can be computed in a single backward sweep of the dependence graph, and hence is $O(n)$.

Consider the operation of this algorithm on the *PowerPC* code sequence in Figure 3(a), and recall that excepting operations such as loads represent control flow points for our purposes. The initial IR after SSA conversion for this code is in Figure 3(b). The first operation (`s4′ = s3 & s4`) is dead after IR **Op 4** (`s4′′ = s3′ + s3′`), so as shown in Figure 3(c), a use of $s_{4'}$ is inserted and $s_{4'} \equiv \langle s_3, s_4 \rangle$. IR **Op 2** results from the fact that the *PowerPC* operation `and.` sets *condition register 0*. The value in *condition register 0* is dead at IR **Op 8**, so we insert a use of $s_{c0'}$ after **Op 8**, and $s_{c0'} \equiv \langle s_3, s_4 \rangle$, as can be seen in Figure 3(c). Finally, IR **Op 5** is dead at IR **Op 7**, so we insert a use of $s_{5'}$ after **Op 7**, and $s_{5'} \equiv \langle s_{3'} \rangle$, as can again be seen in Figure 3(c).

When this code is converted into the target assembly code, register allocation will be performed on the symbolic registers and a register map table describing how to reload physical registers from the translation to achieve the original state [2]. For eliminated registers, this table will contain the names and formulae of the symbolic registers. To reduce storage requirements, side tables may also be dynamically recomputed when an exception occurs [15].

Note that this algorithm overly conservative because repair is not necessary if no instruction can trigger a synchronous exception between the point of the original instruction and the point where its result is killed. Also, repair needs to be possible only up to the last instruction which can cause a synchronous exception.

Thus, as shown in Figure 4, we can reformulate the algorithm to insert the use operator to keep alive a value only to the last possible exception point. Dead values whose live range does not span an exception point are not backstopped by a 'use' node and will be deleted by a subsequent dead code elimination pass unless they are needed to feed a repair note which may be evaluated to reconstruct the precise exception state. This algorithm takes $O(N^2)$ time, where N is again the number of instructions in a CFG.

As mentioned earlier, we assume that the CFG is an extended basic block with a single entry point and multiple exits. Figure 5 illustrates a CFG for an extended basic block with $P = 5$ paths. Each path in the CFG is traversed in a depth-first manner, keeping track of (1) the last excepting operation on a path and (2) the last instruction on a path to write each register, as depicted in our **final** recursive algorithm in Figure 6.

To make this point clearer, consider the (extended basic block) CFG in Figure 5. Taking the leftmost path, *P1*, instruction **1** is first encountered. It writes to register `r4`. A bit later on this path, instruction **3** can raise a synchronous exception, as noted by the **E**. Finally at the end of this path, instruction **5** writes to register `r4`, thus killing (on this path) the result computed by instruction **1**. If instruction **1** is dead on all paths, then the algorithm:

- Notes that instruction **3** — the *last_excepting_op* — represents a potential use of `r4`.
- Saves the information needed to compute `r4` if an exception does occur at instruction **3**.
- Converts the *killed_ins*, instruction **1**, to a repair note and deletes it.

Similar actions occur on path *P4*. When instruction **4** is encountered, it writes to register `r4` and hence kills the result of instruction **1**. However, no excepting instructions have been encountered on this path, hence no repair note need be added.

```
foreach operation OP {
    if dead ( target (OP) ) {
        insert_equivalence ( target (OP) == sources (OP) )
        repair_ever := FALSE;
        for all paths p starting at OP {
            repair_path := FALSE
            for all operations I on path p {
                if operation I can cause synchronous exception {
                    repair_ever := TRUE;
                    repair_path := TRUE;
                    last_excepting_op := I;
                }
                if operation I kills target (OP) && repair_path {
                    insert_use ( target (OP), last_excepting_op )
                    next path;
                }
            }
        }
        convert2repairnote (OP);
    }
}
```

Figure 4. Algorithm augmented so as to avoid repair notes if they are not needed.

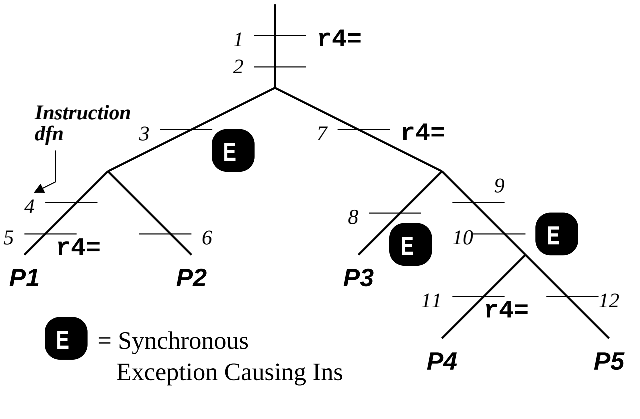

Figure 5. CFG for Extended Basic Block of code with single entry and multiple exits.

```
final (OP, prev_writer, last_excepting_op)
{
   if (!OP) {           // Handle end of recursion
      forall src {
         first_use[src].op = NULL;
         first_use[src].intervening_exception = NONE;
      }
      return first_use;
   }

   if operation OP can cause synchronous exception
     last_excepting_op := OP;

   curr_result_reg := target(OP)
   killed_ins := prev_writer[curr_result_reg];
   prev_writer[curr_result_reg] := OP;

   if killed_ins != NONE {
     if (dead (killed_ins)) {
       // it is dead along all paths; computation can be removed totally
       insert_equivalence ( target(killed_ins) == sources(killed_ins))
       convert2repairnote(killed_ins);
       if (dfn [last_excepting_op] >= dfn[killed_ins]){
         insert_use (target(killed_ins), last_excepting_op)
       } else {
         set_candidate_for_delete(killed_ins);
       }
     } else {
       // instruction is live among some paths, but dead on current path
       // candidate for code sinking (PRE), will be performed below
     }
   }

   if ! branch (OP) {
     first_use       = final (OP->left,  prev_writer, last_excepting_op)
   } else {
     first_use_left  = final (OP->left,  prev_writer, last_excepting_op)
     first_use_right = final (OP->right, prev_writer, last_excepting_op)

     // register-wise combination on control flow splits
     first_use = combine (first_use_left, first_use_right)
   }

   // perform sinking if possible, inserting repair note if necessary
   push_op_down(OP, first_use[curr_result_reg].op);
   if (first_use[curr_result_reg].intervening_exception) {
     insert_use (target(OP), first_use[curr_result_reg].intervening_exception)
     insert_equivalence ( target(OP) == sources(OP))
     convert2repairnote(OP);
   }

   forall src in sources(OP){
     first_use[src].op = OP;
     first_use[src].intervening_exception = NONE;
   }
   if operation OP can cause synchronous exception
     forall regnames defined in architecture
       if first_use[src].intervening_exception == NONE
          first_use[src].intervening_exception = OP;

   return first_use;
}
```

Figure 6. Final algorithm.

```
1   {   s4′  =   s3 & s4 }
2   {  sc0′  =  (s3 & s4) cmp 0 }
3       s3′  =  [s9]
   use  s4′  ;   s4′ == < s3, s4 >
4       s4′′ =   s3′ + s3′
5   {   s5′  =   s3′ + 80 }
6       s3′′ =  [s10]
   use  s5′  ;   s5′ == < s3′ >
   use sc0′  ;  sc0′ == < s3, s4 >
7       s5′′ =   s3′′ + 1
8      sc0′′ =  (s3′′ + 1) cmp 0
```

Figure 7. Reduced Live Range of Repair.

Continuing down path *P4*, instruction **10** is noted as *last_excepting_op*. At instruction **11**, register `r4` is written, thus killing the result computed at instruction **7**. If instruction **7** is dead on all paths, then 3 steps akin to those above on path *P1* are performed.

Note that the algorithm in Figure 6 uses *dfn* — a *depth first numbering* of nodes — to determine whether an excepting operation has occurred between an operation and its killer. This *dfn* represents the relative position of each instruction on a path, and is monotonically increasing from the start to the end of any path. For example, on path *P1* in Figure 5, the instructions' *dfn*'s are *1, 2, 3, 4, 5,* while on path *P4* they are *1, 2, 7, 9, 10, 11*.

The algorithm described here uses recursive descent to visit each node in the control flow graph in depth first order. The bottom half of this algorithm performs code sinking (partial redundancy elimination) on the upward pass of the recursive descent algorithm. Each node is visited twice (during the downward and the upward pass), so we posit that the algorithm is $O(N)$.

For each register name, the first use following the current op is maintained in `first_use`. On control flow splits, data from both paths is combined. The `combine` function propagates upward the `first_use` of the a register if it is only used along a single path, or defines the control flow split as the first use if the register is used along both paths. (Other types of combine operations are possible, but lead to code duplication. This is a trade-off which could make good use of profile data available in a dynamic compilation system.)

This algorithm can be further extended to consider register pressure when making optimization decisions, since in some circumstances the optimization technique presented here can extend two live ranges to eliminate one dead live-range, thereby increasing register pressure and forcing the register allocator to spill registers to memory.

Applying the modified algorithm to the example, the live range of repair notes is reduced as can be seen by comparing Figure 7 to Figure 3(c). However, none are actually eliminated in this particular example.

4 Other Optimizations

The algorithm presented in the previous section can be adapted trivially to schedule instructions later than their original schedule (code sinking). A repair note is then inserted in the original instruction slot. Note that no special provisions have to be made to preserve the input values of the repair note, since they are also an input to the rescheduled instruction:

```
foreach operation op
    if schedule_below ( op )
        %% Deletes op and inserts as repair note.
        convert2repairnote (op);
```

Unspeculation (partial redundancy elimination) can be handled by a combination of dead code elimination along paths where a computation is redundant, and code sinking for those paths where the instruction is needed.

```
[Initially, all registers are assumed to
 be in their home locations]

0x00  lwz  R32, 0(R9)     [ r3 := R32 ]
0x04  add  R3, R32, R32   [ r4 := R3 ]
0x08  lwz  R33, 0(R10)    [ r3 := R33 ]
0x0C  addi R5,R33,1       [ -unchanged- ]
0x10  cmpi CR0,R5,0       [ -unchanged- ]
```

Figure 8. Annotations mapping physical to architected registers.

A similar approach can also be applied to other optimizations, such as constant propagation, constant folding and commoning, where the original code becomes dead and is treated as described in Section 3.

An approach based on repairing state can also be used to eliminate memory operations if disambiguation is possible at dynamic compile time. However, this is only possible in a uniprocessor context, as multiprocessor configurations may introduce additional producers and consumers for memory values which cannot be adequately analyzed.

When performing instruction scheduling during dynamic optimization, state repair can also be used to achieve precise exceptions. We give a list scheduling algorithm modified to incorporate state repair for achieving precise exception semantics.

```
do {
  ready_ins := initially_ready(CFG);
  ins := select_ins(ready_ins);

  if (ins can cause exception){
    predecessors := predecessors (ins, CFG);
    issue_repair_notes_from_list (predecessors);
  }

  ready_ins := ready_ins UNION successors(ins, CFG)
} until (ready_ins = EMPTY_SET)
```

5 Repair Handler

Since repair notes are rarely evaluated (only on synchronous exceptions), no actual code is generated. Instead, the repair notes are stored in compact form in main memory, and interpreted by an interpretative evaluator on demand. Thus, the cost of repair notes consists of time penalties when entering the exception handler to interpret the repair notes associated with the current instruction group, and the cost to store the repair notes and the interpretative evaluator for the repair notes.

When a synchronous exception occurs, control first passes to the repair handler. To compute the entire program state, the repair handler sequentially evaluates all repair notes in a single forward sweep. Then, all registers are assigned to their “home locations” (typically, the identity mapping) before control is transferred to the translation of the exception handler.

Consider again the previous code example, which may have been assembled into the *PowerPC* code fragment in Figure 8. Because the algorithm did not consider register pressure, the optimized code fragment requires more than the original number of registers, leading to register numbers greater than R31. It is desirable to utilize available registers if the target architecture has more registers than the source architecture, but could lead to performance degradation otherwise, making consideration of register pressure an important aspect.

Register mappings are updated incrementally, and indicated after each assembly instruction, as shown in Figure 8. Target architecture registers are indicated by capitalized register names.

Figure 9 shows the repair notes stored for the code fragment in Figure 8. Note that to reduce the number of bits necessary for storing the symbolic registers associated with repair notes, their (separate) name space can also be allocated using coloring, as is done in Figure 9.

```
      S0   =  R3 & R4
      SC0  = (R3 & R4) cmp 0
0x00: [r4 :=  S0;  cr0 := SC0 ]
      S0   =  R3 + 80
0x08: [r5 :=  S0;  cr0 := SC0 ]
```

Figure 9. Repair Notes.

6 Results

To evaluate the performance potential of dead code elimination and code sinking in dynamic optimization environments, we used the DAISY environment to evaluate the optimization opportunity. This evaluation was performed for two systems, IBM *PowerPC* and IBM *System/390*.

To gauge the performance opportunity, we applied the algorithm in Figure 6 to the DAISY group intermediate representation to determine the number of intermediate operations that can be eliminated from the execution path. The intermediate operations are defined as having a single destination and a variable number of inputs. *PowerPC* and *System/390* instructions requiring multiple destinations were cracked into a sequence of simpler instructions.

Figure 10 shows the number of intermediate operations eliminated compared to the case when optimization is retarded by conservative assumptions about excepting instructions. This number of operations directly reflects the number of primitive operations which must be executed on a VLIW platform (such as BOA [10]). The number similarly reflects the number of simple micro-ops that would be executed in a layered instruction set implementation of a superscalar.

The percentage of IR operations which can be eliminated in the benchmarks presented have been computed for two different system operation points of the DAISY dynamic compilation system. These correspond to aggressive and conservative ILP extraction policies, labelled **(a)** and **(c)** respectively for each of the benchmarks in Figure 10.

For *PowerPC* code almost 5% of primitive operations are removed on average in the aggressive case compared to only about 3% in the conservative case. For reasons explained below, more primitive operations are removed for *System/390* code: 12% and 10% on average respectively for the aggressive and conservative cases.

The code which was analyzed here was compiled with high optimization levels to mirror typical tuned SPEC code. Hence, any optimization opportunity found here is over what any state-of-the-art offline compiler can achieve. In particular for *System/390*, an additional improvement is achieved by eliminating the computation of dead condition codes which are nearly always set as a byproduct of *System/390* arithmetic instructions. Since emulating condition codes of one architecture on another often requires a long sequence of instructions, eliminating these computations is particularly important for achieving good performance in system emulation [10].

Differences between the aggressive and conservative ILP extraction policies were the result of several factors: the thresholds used for determining when groups are extended, the maximum allowable group size, and the infinite resource ILP target [8]. Aggressive group formation policy generates larger instruction groups in an effort to extract more ILP from the code. As expected, larger group size did lead to more opportunity for dead code elimination, since all registers must be considered live on group transitions.

7 Related Work

While early work on dynamic compilation was concerned mostly with reducing the overhead per translated instruction, the applicability of more aggressive optimizations has become an issue in more recent work.

Special purpose optimizations for deferring the full materialization of condition codes have been performed in previous architecture emulation systems, such as Wabi [13]. However, this type of deferred materialization has usually required that all source values be copied to defined storage to be used for later materialization. This required significantly more overhead than the present approach, but was a significant performance improvement compared to full instantiation of condition codes which usually requires quite complex operations to match the semantics of the emulated architecture.

DAISY explores the use of aggressive ILP optimizations in dynamic binary translation [7, 6, 11, 8, 1]. DAISY uses aggressive speculation, but performs in-order commit operations to the emulated processor state to achieve precise exceptions.

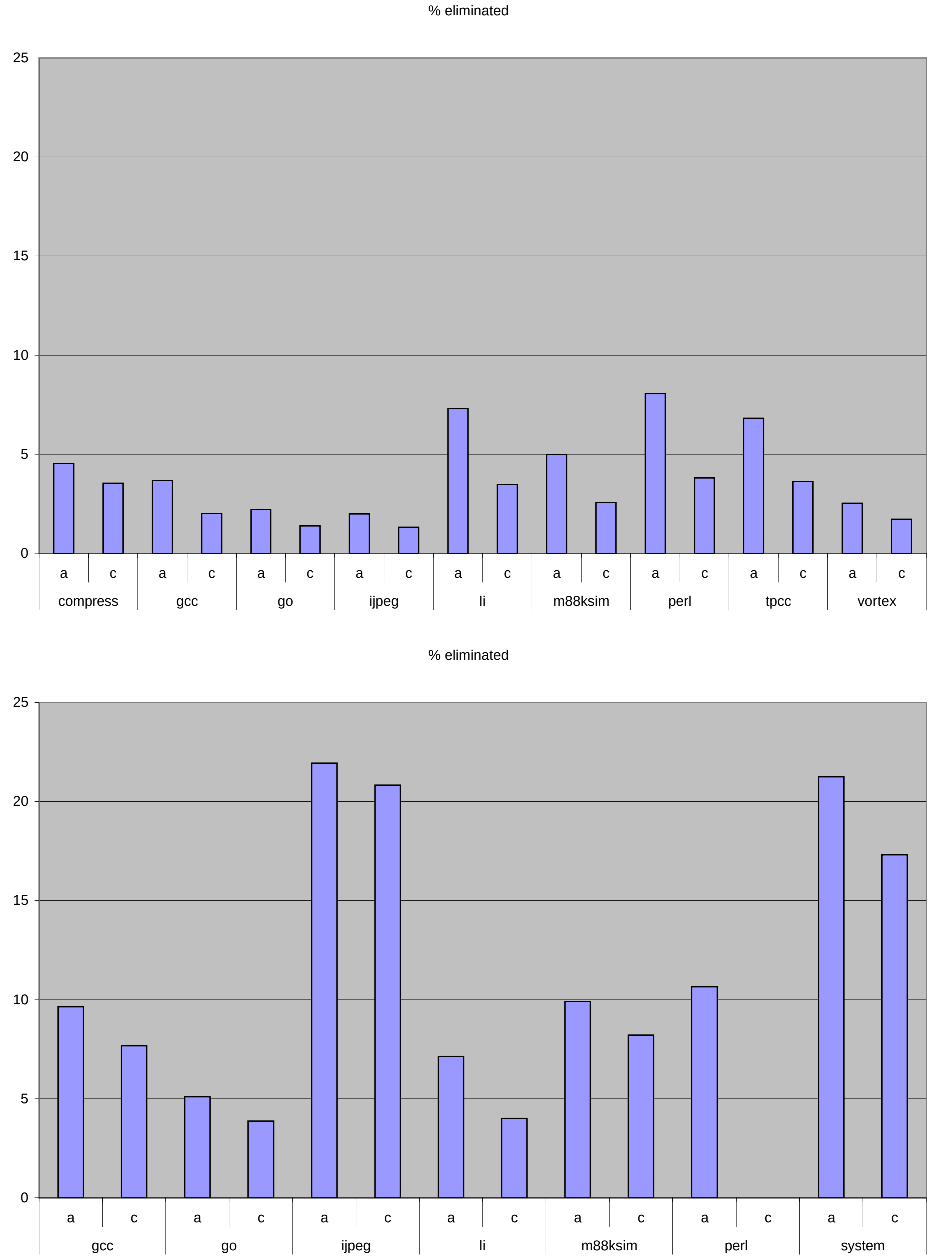

Figure 10. Dead code elimination opportunities for IBM *PowerPC* (top) and IBM *System/390* (bottom)

DAISY exploits the atomic nature of VLIW instructions in the target architecture to perform dead code elimination in the scope of a single long instruction word.

The DYNAMO dynamic optimization system performs dynamic optimization on HP-PA binaries, with the target being the HP-PA instruction set [4, 3]. DYNAMO allows for aggressive optimizations, but uses program annotation or a user-selectable conservative optimization mode to deal with binaries where precise exception behavior is an issue.

The Transmeta binary translation system for Intel *x86* code [14] and the BOA system for IBM *PowerPC* code [10] use a hardware rollback/commit scheme to ensure precise exception behavior. At the beginning of each translation, the entire processor state is checkpointed at the entry of each translation group. When an exception is raised, the entire processor state is rolled back to the translation fragment entry state, and then the interpreter interprets instructions sequentially to compute all processor state.

Le [15] and Altman *et al.* [2] show how a repair mechanism can be used to reduce the cost of register allocation in binary translation.

Maintaining full program state for exception handling is related to the problem of presenting the full program state of optimized programs to debuggers. In both cases, otherwise unused state which is not being computed by the optimized program may be accessed [12].

The constraints for a presenting program state in a debugger are different, since it may be acceptable to devote more time to both the compilation and state recovery process. On the other hand, an optimizing compiler must be able to deal with a state query at arbitrary points, whereas a dynamic compilation system is aware that such queries are by restricted to those points where an instruction can raise a synchronous exception.

In fact, debuggers running under a dynamic compilation system present an interesting mix of these dual requirements. In DAISY, we solve this by detecting code modification or similar events (due to the setting of a breakpoint) and can dynamically recompile the effected code.

8 Conclusion

Maintaining precise exceptions is an important aspect of achieving full compatibility with a legacy architecture. While asynchronous exceptions can be deferred to an appropriate boundary in the code, synchronous exceptions must be taken when they occur. This introduces uncertainty into the liveness analysis since otherwise dead processor state may be exposed when an exception handler is invoked. Previous systems either had to sacrifice full compatibility to achieve more freedom to perform optimization, use less aggressive optimization or rely on hardware support.

In this work, we have demonstrated how aggressive optimization can be used in conjunction with dynamic compilation without the need for specialized hardware. The approach is based on maintaining enough state to recompute the processor state when an unpredicted event such as a synchronous exception may make otherwise dead processor state visible. The transformations necessary to preserve precise exception capability can be performed in linear time.

References

[1] E. Altman and K. Ebcioğlu. Simulation and debugging of full system binary translation. In *Proc. of the 13th International Conference on Parallel and Distributed Computing Systems*, pages 446–453, Las Vegas, NV, August 2000.

[2] E. Altman, K. Ebcioğlu, M. Gschwind, and S. Sathaye. Efficient instruction scheduling with precise exceptions. In preparation.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent Dynamic Optimization System. SIGPLAN PLDI, pages 1–12, June 18-21, 2000, Vancouver, BC, June 2000.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of Dynamo. Technical Report 99-78, HP Laboratories, Cambridge, MA, June 1999.

[5] H. Chung, S.-M. Moon, and K. Ebcioğlu. Using value locality on VLIW machines through dynamic compilation. In *Proc. of the 1999 Workshop on Binary Translation*, IEEE Computer Society Technical Committee on Computer Architecture Newsletter, pages 69–76, December 1999.

[6] K. Ebcioğlu and E. Altman. DAISY: dynamic compilation for 100% architectural compatibility. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 26–37, Denver, CO, June 1997. ACM.

[7] K. Ebcioğlu and E. Altman. DAISY: dynamic compilation for 100% architectural compatibility. Research Report RC20538, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1996.

[8] K. Ebcioğlu, E. Altman, S. Sathaye, and M. Gschwind. Execution-based scheduling for VLIW architectures. In *Euro-Par '99 Parallel Processing – 5th International Euro-Par Conference*, number 1685 in Lecture Notes in Computer Science, pages 1269–1280. Springer Verlag, Berlin, Germany, August 1999.

[9] K. Ebcioğlu, R. Groves, K. Kim, and G. Silberman. VLIW compilation techniques in a superscalar environment. In *Proc. of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, volume 29 of *SIGPLAN Notices*, pages 36–48, Orlando, FL, June 1994. ACM.

[10] M. Gschwind, E. Altman, S. Sathaye, P. Ledak, and D. Appenzeller. Dynamic and transparent binary translation. *IEEE Computer*, 33(3):54–59, March 2000.

[11] M. Gschwind, K. Ebcioğlu, E. Altman, and S. Sathaye. Binary translation and architecture convergence issues for IBM System/390. In *Proc. of the International Conference on Supercomputing 2000*, Santa Fe, NM, May 2000. ACM.

[12] J. Hennessey. Symbolic Debugging of Optimized Code. ACM Transactions on Programming Languages and Systems, July 1982, Volume 4, Issue 3, pages 323–344, ACM Press.

[13] P. Hohensee, M. Myszewski, and D. Reese. WABI CPU emulation. In *Hot Chips VIII*, Palo Alto, CA, 1996.

[14] E. Kelly, R. Cmelik, and M. Wing. Memory controller for a microprocessor for detecting a failure of speculation on the physical nature of a component being addressed. US Patent 5832205, November 1998.

[15] B. Le. An out of order execution technique for runtime binary translators. In *Proc. of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 33 of *SIGPLAN Notices*, pages 151–158, San Jose, CA, 1998. ACM.

[16] S. Sathaye, P. Ledak, J. LeBlanc, S. Kosonocky, M. Gschwind, J. Fritts, Z. Filan, A. Bright, D. Appenzeller, E. Altman, and C. Agricola. BOA: Targeting multi-gigahertz with binary translation. In *Proc. of the 1999 Workshop on Binary Translation*, IEEE Computer Society Technical Committee on Computer Architecture Newsletter, pages 2–11, December 1999.