Register Liveness Analysis for Optimizing Dynamic Binary Translation *

Mark Probst and Andreas Krall and Bernhard Scholz
CD Laboratory for Compilation Techniques
Institut für Computersprachen, Technische Universität Wien
Argentinierstraße 8, A-1040 Wien
{schani,andi,scholz}@complang.tuwien.ac.at

Abstract

Dynamic binary translators compile machine code from a source architecture to a target architecture at run time. Due to the hard time constraints of just-in-time compilation only highly efficient optimization algorithms can be employed. Common problems are an insufficient number of registers on the target architecture and the different handling of condition codes in source and target architecture. Without optimizations useless stores and computations are generated by the dynamic binary translator and cause significant performance losses. In order to eliminate these useless operations, a very fast liveness analysis is required.

We present a dynamic liveness analysis algorithm that trades precision for fast execution and conducted experiments with the SpecInt95 benchmark suite using our PowerPC to Alpha translator. The optimizations reduced the number of stores by about 50 percent. This resulted in a speed-up of 10 to 30 percent depending on the target machine. The dynamic liveness analysis results are very close to the most precise solution.

1 Introduction

In dynamic binary translation code is translated at run time to machine code of the target architecture. As the translation time is included in the run time, optimizations are beneficial only where the shorter execution time pays off the higher translation time. Often the number of registers of the emulated architecture is larger than the number of registers of the target architecture. A common approach is to keep the emulated registers in memory and to perform local register allocation for the translation units. Without liveness information this leads to dead stores of emulated registers. Another common problem is the implicit computation of condition codes and other side effects of instructions. These side effect computations are expensive, but the results are used rarely. Therefore, optimizations such as the elimination of dead register stores and dead computations improve the efficiency of binary translators.

A prerequisite for these optimizations is liveness information of registers and computations. Precise liveness analysis requires an iterative analysis algorithm which is too expensive to be executed at run time. We have developed a simple dynamic liveness analysis which is precise enough for dead register stores and computation elimination.

In Section 2 we give an overview of our binary translator `bintrans`, describe how liveness information is computed and show how this information can be used for elimination of stores and condition code computations. Section 3 goes into the details of liveness analysis and explains why our dynamic approach cannot achieve the most precise solution. In Section 4 we present detailed experiments which demonstrate the effectiveness of our simple liveness analysis. In the last section we draw our conclusions and give an outline of the future work.

2 `bintrans`

`bintrans` is a freely available dynamic binary translator that translates binary code for a source architecture into instruction sequences for a target architecture and executes the translated code on-the-fly. `bintrans` supports several source-target combinations whereas we focus on the PowerPC to Alpha translator in this paper.

For `bintrans` the unit of translation is a basic block which is a sequence of instructions with a single entry and a single exit. Jumps in the target-architecture are replaced by jumps to a *dispatcher*. The task of the dispatcher is to look up the target address in a lookup table. If the corresponding basic block is already translated, it will simply branch to the translated block. Otherwise, the basic block is translated and the newly translated basic block is executed. In case of direct jumps `bintrans` resolves the jumps if the target

*This research is partially supported by the Austrian Science Fund (Project P13444) and the Christian Doppler Forschungsgesellschaft.

of the jump is already translated. Then, the dispatcher is not invoked anymore and the direct jump is "hard-wired". However, indirect jumps are always resolved through the dispatcher since the target address is not known apriori.

The PowerPC to Alpha translator has to deal with the problem that the source architecture features more registers than the target architecture. Therefore, source registers of the PowerPC architecture are kept in a memory area which we call *register save area*. Within blocks, register allocation techniques are applied for reducing the number of load and store operation. If the number of registers used within a block does not exceed the number of target registers, the used source registers are kept in target registers for the whole block. In that case, all required registers are loaded at the beginning of the block from the register save area and stored back into the register save area at the end. If the block uses more source registers than available target registers, loads and stores are inserted within the block as well.

The memory traffic generated by loads and stores at the beginning and at the end of basic blocks deteriorates performance significantly, especially on in-order machines with a slow memory subsystem. Without liveness information, we must assume that all registers modified within a block might be read in consecutive basic blocks. With liveness information, we can avoid storing modified register values to the register save area if we know that they are not *alive*, i.e. they are not used in a later point in time.

2.1 Register Mapping

The PowerPC has 32 general purpose and 5 special purpose integer registers. One of the 5 special purpose registers is the condition register, which can be seen as 8 separate 4 bit fields. Each comparison instruction sets one of those fields. Three of the four bits are set to the results of the comparison (less than, greater than, equal). The fourth bit is set to a copy of a specific bit of another register, the purpose of which is not relevant for our discussion. A conditional branch instruction branches on an individual bit (specified in the instruction word) in the condition register. Many PowerPC instructions have an alternative form which automatically compares the result of their operation with zero and set the first condition register field (the four most significant bits) accordingly.

The Alpha has 31 general purpose integer registers and no special purpose registers of interest. A comparison writes zero or one into a general purpose register, depending on the outcome. Conditional branches branch on the contents of general purpose registers.

Since inserting condition bits into the condition register is inefficient, `bintrans` improves the access to the first condition register field by keeping each of the four bits in separate Alpha registers. With only one instructions a condition bit can be generated and tested.

2.2 Optimization

Consider the following basic block of two PowerPC instructions:

```
addi. r3,r3,-1
beq somewhere
```

The first instruction decrements register `r3` and compares the result against zero. The first condition register field is set according to the outcome of the comparison. The second instruction jumps to `somewhere` if the "equal"-bit in the first condition register field is set, otherwise it falls through.

The Alpha code generated for this block is given as follows by assuming that the branches are already hard-wired:

```
  ldl   $5,o_r3($27)
  subl  $5,1,$5
  cmplt $5,0,$7
  cmpgt $5,0,$8
  cmpeq $5,0,$9
  bne   $9,branch_taken
  stl   $5,o_r3($27)
  stl   $7,o_lt($27)
  stl   $8,o_gt($27)
  stl   $9,o_eq($27)
  b     fallthrough_target
branch_taken:
  stl   $5,o_r3($27)
  stl   $7,o_lt($27)
  stl   $8,o_gt($27)
  stl   $9,o_eq($27)
  b     somewhere_target
```

The first instruction loads the value of the source register `r3`. Note that the names with prefix "`o_`" are symbolic names for the offsets of registers in the register save area.

The second instruction performs the subtraction and the three instructions after that generate the three condition bits of the comparison with zero. After the computation of the condition bits the actual branch is executed. Before terminating the basic block the registers which are held in registers of the target architecture are stored back to the register save area.

When translating a block, liveness information is used for two different purposes. First, only values which are alive after the instruction in which they are produced are actually generated. In the above example, assume that none of the three condition bits are live at the end of the block. In that case, only the equal bit would be generated, because it is alive after the compare instruction (because the conditional branch depends on its value). The other two bits are dead, so they would not be generated.

The second opportunity for using liveness information during translation is the removal of register stores. Assume register `r3` was alive on the fallthrough edge but dead on the edge to the block `somewhere`. In that case, we would not generate the store for register `r3` before branching to `somewhere_target`.

To illustrate the effect of those two transformations, assume that the three condition bits were dead on both exits and `r3` alive on the fallthrough exit but dead on the other one. The generated code would then look like this:

```
  ldl   $5,o_r3($27)
  subl  $5,1,$5
  cmpeq $5,0,$9
  bne   $9,branch_taken
  stl   $5,o_r3($27)
  b     fallthrough_target
branch_taken:
  b     somewhere_target
```

The other circumstance where `bintrans` uses liveness information is when new liveness information becomes available for a block which has already been translated. This happens when a direct jump is first executed. As explained above, a direct jump is translated to a call to the dispatcher. Should the target block not be available, it is translated. Then, the out set of the target block is used to eliminate register stores in the already generated code. The register stores always directly precede the call to the dispatcher, so that removing some of the stores is an easy process. We simply go backwards one instruction at a time and if the instruction is a register store, we examine which register it stores and delete the store if the stored register is not alive. A register store instruction can be easily identified by examining the base register and the offset. If the base register is the register pointing to the register save area, the store is a register store. In that case, the offset determines which register it stores.

3 Register Liveness Analysis

Global register liveness analysis [ASU86] is a prerequisite of `bintrans` for producing efficient code. Since the binary translator has very hard time constraints, it is apparent that exhaustive data flow analysis[1] techniques will not deliver an analysis result in a short period of time. Therefore, we propose a dynamic approach that works fast and still obtains satisfactory results.

[1]An exhaustive data-flow analysis considers the effect of all basic blocks and their branching structure.

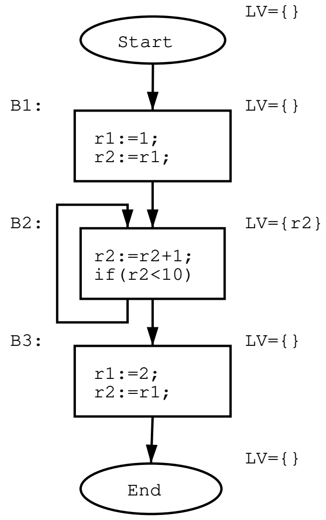

Figure 1. Liveness Analysis Example

Dynamic liveness analysis is only performed when basic blocks are translated. The liveness information is kept and at the end of execution it is stored in a file. A further run of a program reads the file and refines the information from previous runs. In this setting the propagation of liveness information is performed over several runs of a program. Due to the restricted propagation of information (along the execution path and over several runs of a program) considerations about correctness and precision of the analysis result are fundamental for our approach.

Liveness analysis statically determines whether a register is *alive* or *dead* at some program point. A register is alive at some program point if the value of the register is used in a later point in time. In contrast a register is *dead* if there are no further uses of the value. In our framework the analysis result can be weakened in terms of precision. A register which is dead and marked as alive cannot harm program semantics when optimizations are applied – the analysis result is *safe*.

For dynamic analysis a trade-off between precision and runtime is essential. The most precise analysis result does not pay off if the analysis time is significantly higher than the achievable benefit of the optimizations. Moreover, dynamic liveness analysis also has to cope with the problem that only fragments of the control flow exists when the program is translated. For not translated basic blocks the analysis has to assume that all CPU registers are alive which definitely worsens the analysis results.

The example in Figure 1 depicts the control flow of a program that consists of a simple loop. For sake of simplicity we have two registers (i.e. r_1 and r_2). Inside the loop, register r_2 is incremented and the loop is terminated if the

Node u	use(u)	def(u)	*LV*(u)	*MFP*(u)	*LFP*(u)
start	$\emptyset$	$\emptyset$	*LV*(B_1)	$\emptyset$	$\emptyset$
B_1	$\emptyset$	$\{r_1, r_2\}$	*LV*(B_2)-$\{r_1, r_2\}$	$\emptyset$	$\emptyset$
B_2	$\{r_2\}$	$\{r_2\}$	*LV*(B_3) $\cup$ *LV*(B_2) $\cup$ $\{r_2\}$	$\{r_2\}$	$\{r_1, r_2\}$
B_3	$\emptyset$	$\{r_1, r_2\}$	*LV*(*end*)-$\{r_1, r_2\}$	$\emptyset$	$\emptyset$
end	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Table 1. Dataflow Analysis of Example in Figure 1

value of r_2 is greater than or equal to 10. Before and after the loop we assign register r_1 a constant value and register r_2 is assigned the value of r_1. For the analysis we need the liveness information of registers r_1 and r_2 at the entry of the basic blocks. This information can be simply deduced by looking at all paths ending in the end node. For example basic block B_3 has only one path to the end node and at the entry we do not need any previous computation of registers r_1 and r_2 since register r_1 is assigned a constant and register r_2 is assigned the constant value of r_1. Therefore, both registers are dead at the entry of B_3. In the figure the liveness information is given by set *LV* which is empty for B_3. Note that a register is alive at the entry of a basic block if the register is a member of set *LV*. In the example basic block B_2 is more complex. Register r_2 is alive at the entry of B_2 since register r_2 is incremented and the value of r_2 might be used for a further iteration of the loop. Similar to basic block B_3 both registers are dead at the entry of B_1 (i.e. $LV = \emptyset$) since registers are defined in B_1. For the start node we have no definitions and no uses of registers r_1 and r_2. Therefore, we have identical analysis results as given for basic block B_1.

In the example we provide the most precise solution. However, an approximation is still acceptable as long as the result is safe and the optimizing transformation does not destroy program semantics.

3.1 Background of Liveness Analysis

Liveness analysis problem can be optimally solved in polynomial time and the data flow analysis problem is characterized as a backward any path gen/kill problem. The data flow analysis framework computes liveness sets at the entry of basic blocks. The liveness sets are subsets of the power set $LV \in 2^R$ where R are the registers. Moreover, the power set of R induces a partial order: $LV_a \leq LV_b$ if $LV_b \subseteq LV_a$. The partial order corresponds to the degree of information of two liveness sets. A liveness set LV_b which is a true subset of LV_a has fewer registers, that are alive, than liveness set LV_a. Therefore, the degree of information of LV_b is higher than LV_a, e.g. more optimizing transformation can be applied.

In order to obtain a solution for register liveness we need the control flow graph as an underlying program representation. The control flow graph is a tuple $G =< N, E, s, e >$ with a set of nodes N, a set of edges E, a start node s and an end node e. The set of successors is given by $succs(u) = \{v | (u, v) \in E\}$. Since liveness analysis is a backward problem we need to consider reverse paths, i.e. a reverse path $\pi = [n_1, n_2, \ldots, n_k]$ is a finite sequence of nodes where $(n_{i+1}, n_i) \in E$ for all i, $1 \leq i < k$. Note that the sequence can be empty as well. The set of reverse paths $Path(u, v)$ denotes the set of all reverse paths from u to v. As usual a gen/kill framework is represented as tuple $< 2^R, \cup, F, G, M >$ where

- 2^R is a lattice with meet operator $\cup$ where R is the set of registers,
- $F \subseteq 2^R \to 2^R$ is a monotone function space,
- $G =< N, E, s, e >$ is a control flow graph,
- $M : N \to F$ is a map from the nodes of the control flow graph to data flow to functions in F.

The transition function of node u is defined by two constant sets $use(u)$ and $def(u)$. Set $use(u)$ contains all registers which are used. Set $def(u)$ are all registers which are defined in u. Note that if a register r is in $use(u)$ there must not be a definition from the use of the register to the entry of the basic block.

In Table 1 the *def* and *use* sets are given for the example in Figure 1. For start node and end node the sets are empty since no registers are either used or defined. For basic block B_1 and basic block B_3 the *def* set contains registers r_1 and r_2 since both are defined. Although register r_1 is used in the second statement of basic block B_1 and B_3 the preceding definition of r_1 kills the use and r_1 is not a member of the *use* set. For basic block B_2 register r_2 is in the use set since r_2 is not killed by a preceding definition. In addition r_2 is in the *def* set since it is also defined.

Based on sets $use(u)$ and $def(u)$ the transition function of node u is expressed as follows:

$$M(u)(x) = (x - def(u)) \cup use(u)$$

We extend function M to path $\pi = [n_1, \ldots, n_{k-1}, n_k]$ where $M(\pi)$ is defined as $M(\pi) = M(n_k) \circ M(n_{k-1}) \circ \ldots \circ M(n_1)$. If path π is empty ($\pi = []$), then $M([])$ is

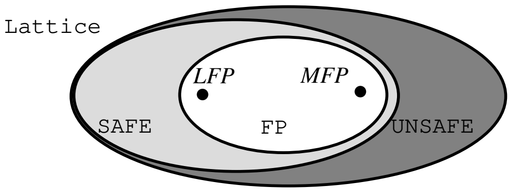

Figure 2. Solutions of Liveness Analysis

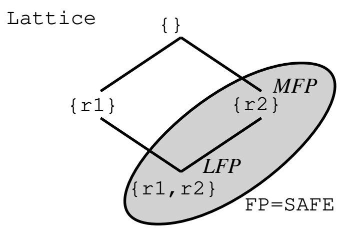

Figure 3. Solutions of Node B_2

the identity function, i.e. $f(x) = x$. Given a reverse path π from end node e to a node u, we define *state*(u) to be $M(\pi)(\emptyset)$.

The most precise solution of a framework is given by the *meet over all path* (MOP) solution which is defined as

$$MOP(u) = \bigcup_{\pi \in Path(u)} state(\pi).$$

The MOP solution for a node u is the meet operation applied over the effect of all reverse paths which end in node u. The formula reflects the desired behavior of the liveness analysis. If there is only one register used on one path to the exit node and the register use is not hidden by a prior definition of the register, the register is alive in u. Note that the definition of the MOP solution is not constructive since an infinite number of paths can occur in the presence of loops. Therefore, an alternative method of computing the MOP solution is necessary.

Since liveness analysis is a monotone and distributive problem the MOP-solution is identical to the *maximum fix-point* (MFP) of the following equation system:

$$LV(u) = \left[\bigcup_{s \in succs(u)} LV(s) - def(u) \right] \cup use(u) \qquad (1)$$

In Table 1 the column $LV(u)$ gives the equations, whereby the equations are constructed by Formula 1. For example the equation of the end node is the empty set $LV(end) = \emptyset$ since the node has no successors and therefore all registers are dead at that point. For basic block B_1 and basic block B_3 the right-hand side of the equations is given by the set difference of the successors and registers r_1 and r_2 since both registers are defined. Only for B_2 have we two successors (i.e., the block itself and block B_3) which are joined by the set union. Register r_2 must be added as well since there is a use of r_2 in B_2. The definition of register r_2 in B_2 was not displayed on right-hand side due to the fact that the use of r_2 cancels the set difference of the *def* set. The equation of the start node is quite simple. There are no uses and definitions in the start node and therefore the solution of this node entirely depends on the solution of its successor B_1.

The solution of the equation system can be solved iteratively. A simple solver initializes the registers $LV(u)$ with the empty set which is the greatest element in the lattice 2^R and iterates until the equations are stable. Note that the iterations starts with an analysis result that is not always a safe solution. The first solution with which the equations hold is called the *maximum fix-point(MFP)*. Feeding the *MFP* solution into the equation system will keep it stable for an arbitrary number of iterations. Note that a fix-point solution is a solution where all equations hold and the maximum fix-point might not be the only fix-point of the equation system. In contrast to the maximum fix-point there exists a *least fix-point(LFP)* which is the smallest fix-point. All other fix-points $FP(u)$ of the equation system must be smaller than the *MFP* and greater than the *LFP*:

$$\forall u \in N : LFP(u) \leq FP(u) \leq MFP(u) \qquad (2)$$

The least fix-point is computed by initializing all registers of the equation system by the smallest element of the lattice 2^R which is R. Then, the least fix-point is obtained by iterating the equations as long as the result is stable.

In Table 1 the maximum fix-point (column $MFP(u)$) and the least fix-point (column $LFP(u)$) are also given. Both fix-point results are the same except for node B_2. The result depends on the initialization of the node. If node B_2 is initialized by R no further iteration of the equation can make the result better since the equation depends on itself.

For our approach it is important to state which solution is safe. An analysis result can be adequate even if it is not the MOP solution. The only requirement is that it must not produce incorrect code when it is used for optimizations. Since we know that the most precise solution is the MOP solution every solution which is below the most precise solution is a safe solution

$$\forall u \in N : SAFE(u) \leq MOP(u)$$

If a solution was unsafe i.e. it is not smaller than or equal to the MOP solution, some registers would not be marked as alive and therefore would give a wrong analysis result. Note that the smallest solution, i.e. all registers are alive, is always a safe solution — but it is not always the most precise solution.

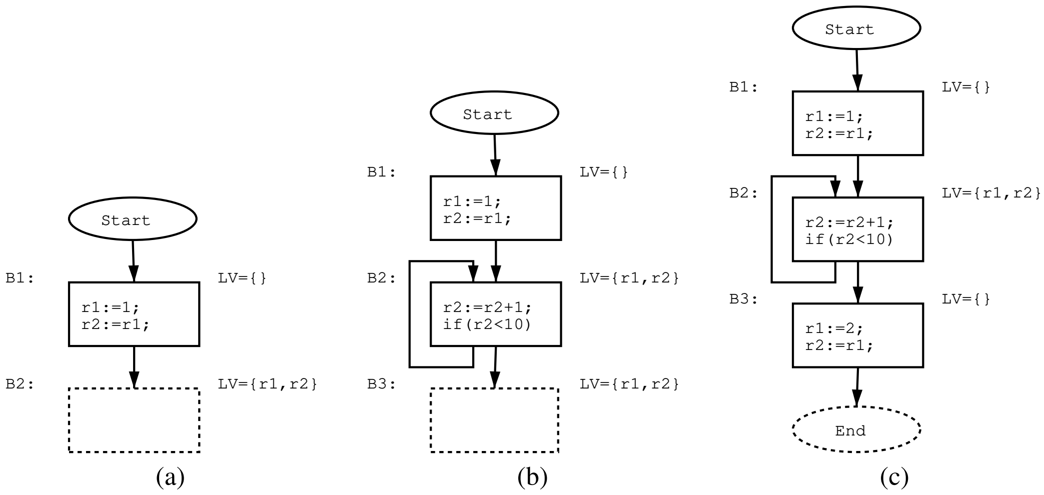

Figure 4. Dynamic Liveness Analysis Example

In Figure 2 possible solutions of a node are given. The analysis must provide a solution which is safe and must not be contained in the unsafe set. Fix-points of the equations must be safe since all fix-points are smaller than the maximum fix-point (cf. Equation 2). As described above we have two dedicated fix-points: The maximum fix-point which represents the MOP solution and the least fix-point which is the smallest fix-point in the set of fix-points.

Recall solution of node B_2. The lattice is formed by the power set of $2^{\{r_1,r_2\}}$. The lattice induces a partial order as depicted as Hasse diagram in Figure 3. Two elements connected by a line means that the element on the higher level is greater than the element on the lower level. Since the relation is transitive, transitive relations can be constructed as paths between two elements (e.g. $\{\}$ is greater than $\{r_1, r_2\}$). In addition safe solutions are highlighted. For solution of B_2 the set of safe solutions is identical with the set of fix-points. In the figure the maximum fix-point and the least fix-point are indicated whereby the maximum fix-point is on a higher level as the least fix-point.

3.2 Dynamic Liveness Analysis

`bintrans` performs the register liveness analysis for a basic block when the basic blocks is translated. The liveness information of the newly translated block is not propagated — it is only kept for further runs of the program. To perform the analysis we have three sets, i.e. *LV*, *def* and *use* for each basic block. Set *LV* is computed as given in Equation 1. Note that for one run of the program we would hardly get a precise solution since the problem is a backward data flow analysis problem. However, by keeping the liveness information the liveness information becomes more precise over several runs since the propagation of information is achieved by a feedback loop.

In general we cannot obtain a fix-point after few runs since the propagation is only performed when a block is translated. Moreover, if no liveness information of a not translated successor block is available, we have to assume that all registers of the not translated block are alive. This approach ensures that all analysis results are safe (i.e. equivalent to an initialization of R) but in the best case we can only achieve the least fix-point of the liveness equations. If the least fix-points differs from the maximum fix-point, we will never get the most precise result (i.e. MOP solution) even when the program is executed several times until a fix-point is obtained. Therefore, our approach does not seem to be viable in terms of precision and convergence. However, it has an excellent performance during translation since the translator just needs to store three sets[2] for a block and the analysis time for computing one equation of the newly translated block is negligible.

In Figure 4 we perform dynamic liveness analysis of the example in Figure 1 when the blocks are translated, and assume that we have no information from a previous run. In the first step (a) we translate block B_1. The block has one not translated successor (B_2) with an unknown behavior. Therefore, we assume that both registers (i.e. r_1 and r_2) are alive. By applying Equation 1 we obtain an empty liveness set for B_1 because both registers are defined in B_1. In the second step (b) we translate block B_2. Again, we assume that in B_3 all registers are alive. According to the liveness equation we mark both registers as alive at the entry of B_2. In the last step (c) we translate B_3. Block B_3 defines both registers and therefore the liveness set is empty. In the example we obtain the least fix-point in the first execution of the program. As given in Table 1 the maximum fix-point

[2] Even the *use* and *def* set can be omitted since it can be computed on the fly.

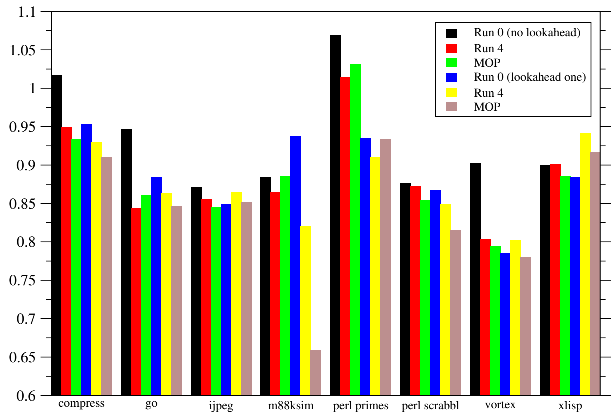

Figure 5. Relative run time (21264, 500MHz)

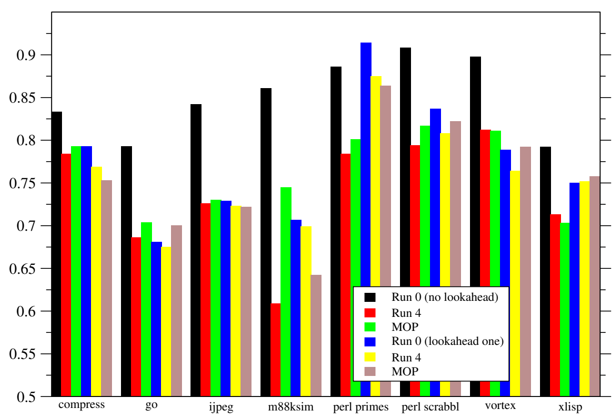

Figure 6. Relative run time (21164PC, 533MHz)

differs from the least fix-point. Therefore, we will never obtain the most precise solution (i.e. MOP solution) for our example. Moreover, the least fix-point was computed after the first execution of the run. In general we cannot expect this fast convergence of the equations.

3.3 Refinement

To overcome the obstacles of locality we can improve the liveness analysis of `bintrans` by looking ahead at not translated basic blocks. Therefore, a translation of one basic block could cause an additional overhead for the analysis of not translated successors of the block. In the worst case there are only two not translated successors and therefore the analysis is still very cheap. The improvement can be substantial since the convergence of the analysis accelerates and for not translated blocks we have a detailed information about their register liveness.

4 Experiments

The main question of our experiments was: Can we come close to the MOP solution with our dynamic liveness analysis and if so, how many iterations and/or lookahead do we need?

To that end we have run several SpecInt95 benchmarks with various configurations of the PowerPC to Alpha binary translator. All run-times given are arithmetic averages over five samples on lightly loaded machines (sums of user and system time). The PowerPC executables were compiled with the GNU C Compiler and statically linked.

What we were mainly interested in was overall run-time and, more specifically, the number of loads and stores executed.

All benchmarks have been run independently with no lookahead and with a lookahead of one block.

One should expect the liveness optimizations to speed the binary translator up compared to running it without those optimizations. To get a picture of this improvement, we ran the benchmarks without any liveness information. All other data in the figures are given relative to this value.

In order to compare the dynamic liveness analysis with the MOP solution, we made one run of the benchmarks and collected gen/kill sets. We did this independently without lookahead and with a lookahead of one block. The results were gen/kill sets of all dynamically translated blocks for each benchmark, and, for the runs with lookahead, gen/kill sets of all successors of all translated blocks.

We then used those gen/kill sets to calculate the MOP solution using a traditional data-flow analysis. The MOP solution was then used by `bintrans` instead of a dynamic solution. The results for these runs are given in the columns labeled "MOP". We should not expect to perform better than that, but we would like to come close.

The results for the regular runs are given in the columns labeled "Run 0" and "Run 4", for the first run, and the fifth, respectively. The columns "Loss" give the differences between the MOP numbers and the fifth run numbers in percent (of the MOP numbers).

Figures 5 and 6 give relative execution times for the benchmarks for an 21264 (500 MHz) and a 21164PC (533 MHz), respectively. The impact on performance for the 21264 is clearly not as big as for the 21164PC. This can be explained by the fact that the former is an out-of-order machine, while the latter operates in-order. The 21264 seems to be able to schedule a large number of stores in parallel to the instructions doing the "real" work to avoid slowing down execution considerably.

Figures 7 and 8 give the relative numbers of executed register loads and register stores. These results show conclusively across all benchmarks that using good liveness in-

Benchmark	Blocks	Source insns	Gen insns	Gen loads	Gen stores	Gen CRF0	Gen CRFx	Billion loads	Billion stores
compress	845	5314	14561	2401	9473	1836	174	22.694	56.555
go	9839	72411	191258	32771	127020	25072	2208	15.291	37.587
m88ksim	2695	16761	45765	7645	29485	5636	585	28.872	81.200
ijpeg	2599	18817	46561	8429	31474	5886	397	7.879	18.171
perl primes	3106	17709	52130	8311	33355	7008	776	7.590	20.531
perl scrabbl	3941	22158	65658	10353	41804	8560	1044	14.074	35.349
vortex	11812	77039	192542	33982	144924	27449	842	41.193	105.927
xlisp	2228	11773	31752	5765	21156	3560	362	33.971	94.462

Table 2. Various benchmark information (no liveness)

Benchmark	Blocks	Source insns	Gen insns	Gen loads	Gen stores	Removed stores	Gen CRF0	Gen CRFx	Billion loads	Billion stores
compress	845	5314	13826	2259	6594	293	1401	174	20.278	30.226
go	9839	72411	176707	28970	72923	3931	13368	2208	13.933	19.168
m88ksim	2695	16761	43419	7141	20363	966	4090	585	25.323	34.562
ijpeg	2599	18817	44467	7987	21837	825	4513	397	7.265	8.146
perl primes	3106	17709	48931	7510	21174	1110	4561	776	6.883	11.916
perl scrabbl	3941	22158	61578	9327	26527	1382	5421	1044	12.848	20.031
vortex	11812	77039	176469	29975	91014	3461	15293	842	35.574	53.133
xlisp	2228	11773	30677	5503	15737	569	2750	362	31.319	49.508

Table 3. Various benchmark information (liveness with lookahead one, first iteration)

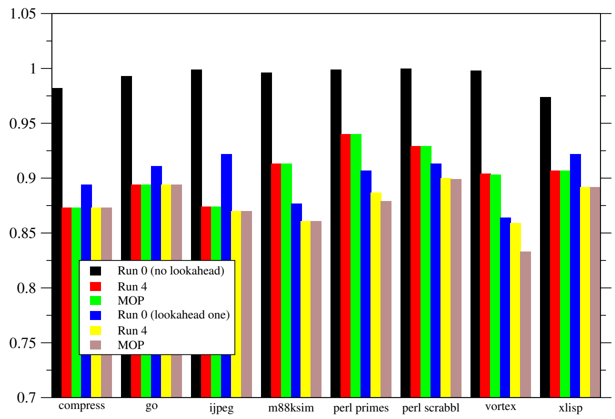

Figure 7. Dynamic register loads

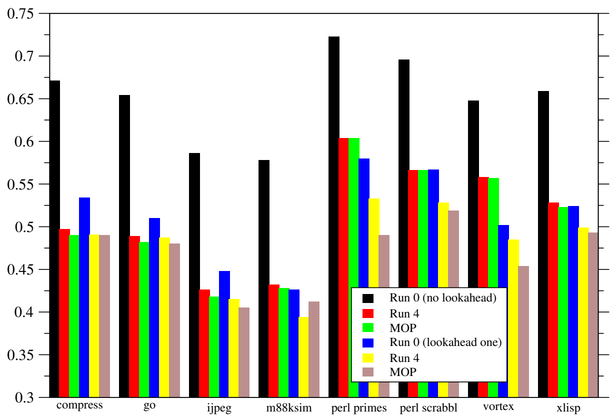

Figure 8. Dynamic register stores

formation (i.e. the MOP solution), about 10% of executed register loads and about 50% of executed register stores can be avoided.

Also, the MOP solution results do not substantially differ between using no lookahead and a lookahead of one block.

The results for the fifth iterated runs show that the dynamic liveness analysis comes very close to the MOP solution after enough iterations.

The result most relevant to our work is the fact that with a lookahead of one block, the iterative liveness analysis comes to within about 10% of the MOP solution for the register stores. This means that it makes perfect sense to do this sort of liveness optimization even if only a single run is done.

Figure 9 gives relative numbers of executed generations of bits in the first field of the condition register. Each generated bit counts as one generation. The difference between the first iteration and the MOP solution is bigger here than for the register stores. This is partly due to the fact that the instructions generating condition bits are not removed

Benchmark	# Registers	no lookahead		lookahead one	
		Dynamic	MOP	Dynamic	MOP
compress	74169	93.3%	93.1%	91.5%	91.0%
go	696331	87.8%	86.5%	87.2%	85.8%
m88ksim	228269	92.5%	92.2%	90.6%	89.9%
ijpeg	221234	92.7%	92.3%	91.0%	90.3%
perl primes	271283	92.7%	92.6%	90.6%	90.0%
perl scrabbl	333191	92.2%	92.1%	90.3%	89.7%
vortex	951132	90.4%	90.3%	87.8%	86.9%
xlisp	177349	92.9%	92.8%	91.4%	91.1%

Table 4. Number of alive registers—Comparison between dynamic analysis and MOP solution

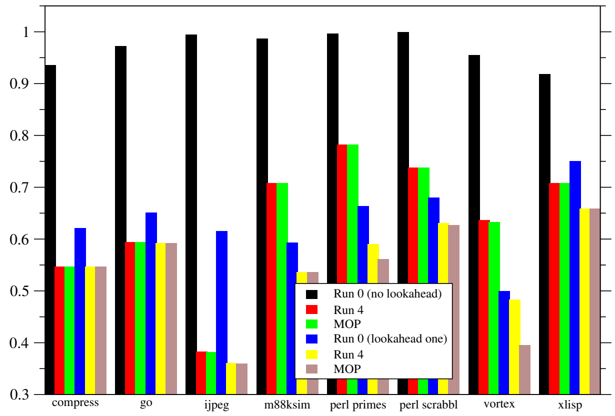

Figure 9. Dynamically generated CR field 0 bits

when new liveness information becomes available. Doing so would require either expensive bookkeeping or a more involved removal algorithm.

Also, the difference between MOP solutions with and without lookahead is significant. However, we are not overly concerned about condition bit generation, since each generated bit costs only one (cheap) compare instruction.

Tables 2 and 3 give various additional information about the benchmarks for a run without liveness analysis and for the first iteration of a run with liveness analysis with lookahead. The meanings of the columns are: number of translated blocks, number of translated instructions, number of generated target instructions (not counting register loads and stores), number of generated register loads, number of generated register stores, number of removed (patched) stores (only in Table 3), number of generated condition field 0 bits, number of generated condition bits in other fields, number of executed register loads and stores in billions.

Even with liveness information, `bintrans` still generates about one register store instruction for each translated source instruction. This makes obvious the need for register allocation between blocks, which we will investigate in the future.

Finally, Table 4 directly compares the solution to the liveness problem of the fifth run of the dynamic algorithm with the MOP solution. The second column gives the number of all registers over all translated blocks. The other columns present the relative portion of these found to be alive by the two analyses, again without and with lookahead. Surprisingly, the differences between the dynamic and the MOP solutions are minimal.

5 Related Work

Iterative data flow analysis including liveness analysis is well described in the book of Aho et al. [ASU86]. Our dynamic liveness analysis can be seen as a variation of classical algorithm. Each run of the program is comparable with an iteration in an iterative data flow analysis framework. Since a backward data flow problem is solved in the wrong direction the number of runs for obtaining a fix-point might be close to the worst case complexity. In comparison with classical approaches we initialize our liveness sets with safe solutions and step-wise improve this solution. In the best case we achieve the least fix-point which can differ from the most precise solution (i.e. MOP solution).

In the eighties when computers were slow several attempts were undertaken to reuse data flow information from previous compilations if the changes in the programs were small. Pollock and Soffa [PS89] presented an iterative incremental data flow analysis algorithm for use in programming environments. They present two different algorithms with different complexity which solve the any path update and all path update problem. Marlowe and Ryder [MR90] present a hybrid algorithm for incremental data flow analysis based on iteration and elimination techniques. All these incremental algorithms are too complex to be executed at run time in a dynamic binary translator. The analysis overhead would spoil the speed-up of the optimization.

Computing the liveness information needed for register allocation is usually done using an iterative work list algorithm [App98]. This information has to be computed repeatedly after insertion of spill code. Since only a few iterations of spill code insertions are necessary and many basic blocks are affected, liveness analysis starts from scratch. Kim and Leung [KL00] very briefly describe an incremental liveness analysis algorithm when liveness information is changed due to live range splitting.

To the best knowledge of the authors none of the available dynamic binary translators or optimizers such as UQDBT [UC00], Dynamo [BDB00], Mojo [CLCG00] and the simulator by Zhu [ZG99] compute liveness information at run time. Static binary translators such as FX!32 [CHH+98] are not restricted in the resource usage and can use iterative data flow analysis algorithms.

6 Conclusion and Future Work

We have demonstrated an effective optimization for a dynamic binary translator based on dynamic register liveness analysis. The optimization mainly targets the reduction of register store operations.

We introduced a new approach to liveness analysis in dynamic binary translation. `bintrans` performs liveness analysis for basic blocks when they are translated. The analysis information is propagated by a feed-back loop between several runs of the program. Our approach has a negligable runtime overhead in comparison with incremental or exhaustive data-flow analysis frameworks. Although our dynamic liveness analysis always produces safe analysis information, we showed that in general the most precise solution cannot be obtained. To improve the analysis result the binary translator also analyzes the successors of a newly translated block.

We conducted experiments with the SpecInt95 benchmark suite using our PowerPC to Alpha translator. The optimization reduced the number of stores by about 50 percent. This resulted in a speed-up of 10 to 30 percent depending on the target machine. In our experiments the dynamic liveness analysis results are very close to the most precise solution (i.e. MOP solution). The analysis difference between the most precise solution and our dynamic analysis is below 2 percent. By analyzing successors of a newly translated block the propagation of analysis information is accelerated. Even for the first run of the program the number of reduced stores is within 10 percent of the number obtained with the most precise analysis.

Our future work in this area will be twofold. First, we want to apply more aggressive optimization techniques to further reduce the number of store operations, which are still a considerable overhead. Although global register allocation is a challenging problem for dynamic binary translation, it would further decrease load and store operations. Second, we want to investigate other dynamic analyses such as alignment analysis.

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[CHH+98] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. FX!32: A profile-directed binary translator. *IEEE Micro*, 18(2):56–64, March/April 1998. Presented at Hot Chips IX, Stanford University, Stanford, California, August 24–26, 1997.

[CLCG00] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David Gillies. Mojo: A dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*. ACM, December 2000.

[KL00] Hansoo Kim and Allen Leung. Frequency-based live range splitting and rematerialization. *KSEA Letters*, 29(1), December 2000.

[MR90] Thomas J. Marlowe and Barbara G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 184–196. ACM SIGACT and SIGPLAN, ACM Press, 1990.

[PS89] Lori L. Pollock and Mary Lou Soffa. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering*, 15(12):1537–1549, December 1989.

[UC00] David Ung and Cristina Cifuentes. Machine-adaptable dynamic binary translation. In *Proceedings of the ACM Sigplan Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO-00)*, volume 35.7 of *ACM SIGPLAN NOTICES*, pages 41–51, N.Y., January 18–18 2000. ACM Press.

[ZG99] Jianwen Zhu and Daniel D. Gajski. A retargetable, ultra-fast instruction set simulator. In *Proceedings of the Design Automation and Test Conference In Europe*, 1999.