Register Allocation Aware Instruction Selection

David Ryan Koes

October 2009
CMU-CS-09-169

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

In existing optimization frameworks, compiler passes are not tightly integrated and often work at cross purposes. In this report we describe an integration framework for the key backend compiler optimizations of register allocation and instruction selection: Register Allocation Aware Instruction Selection (RA^{2}ISE). We discover that the fundamental building block of the RA^{2}ISE framework, register allocation aware tiles (RAATs), introduce significant complexity into the network flow model of register allocation. It is unlikely that efficient and effective solution techniques exist when RAATs are incorporated into the model. We also explore the merits of another component of the RA^{2}ISE framework, feedback driven instruction selection and find that the expected benefits are far outweighed by the necessary costs.

This research was sponsored by the National Science Foundation under grant numbers CCF-0702640, CCR-0205523, EIA-0220214, and IIS-0117658; and Hewlett Packard under grant number 1010162.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Compilers, Register Allocation, Instruction Selection, Backend Optimization

In existing optimization frameworks, compiler passes are not tightly integrated and often work at cross purposes. The absence of integration has a negative effect on code quality as is highlighted by studies in adaptive and iterative compilation [1, 2, 4, 9, 12, 13]. These studies exhaustively search for better orderings and combinations of existing compiler optimizations. Average performance improvements of 5%–20% are found and some numerical kernels increase in performance by as much as 4x. In an optimization framework with tightly integrated passes, the pass execution order and optimization behavior would be dependent on both the input code and the context and feedback of the integrated optimization passes.

In this report we describe an attempt to integrate instruction selection and register allocation into a single feedback-driven optimization framework. We consider the state-of-the-art NOLTIS instruction selector [6, 7] and a progressive register allocator that is based on a highly expressive global MCNF model of register allocation [7, 8]. Instruction selection and register allocation are essential components of the compiler backend. Together they are responsible for the translation of the compiler's target-independent intermediate representation into target-specific assembly. However, it is clear that instruction selection and register allocation do not function in isolation; indeed, they are tightly intertwined and, ideally, a principled compiler backend would seek to integrate these two problems. This report proposes the *Register Allocation Aware Instruction SElection (RA^{2}ISE)* optimization framework and explores the fundamental limitations of this framework.

Instruction selection and register allocation are interdependent. Register allocation is constrained by the instruction sequence generated by instruction selection. Conversely, the quality of the instruction selection directly depends upon the accuracy of tile costs. These tile costs are inherently inaccurate since spills, register preferences, and move coalescing may change the instructions corresponding to a tile. For example, although the instruction sequence of Figure 1(a) contains fewer instructions, it may actually end up with more instructions than that of Figure 1(c) if the register allocator can successfully coalesce the move instructions, as shown in Figure 1. In the RA^{2}ISE framework, instruction selection and register allocation are incorporated into a feedback loop in which each phase can influence the other phase as shown in Figure 2.

```
movl  (p),t1                    movl  (ecx),ebx
leal  (x,t1),t2        ⇒        leal  (edx,ebx),edx
leal  1(y),t3                   leal  1(eax),eax
leal  (t2,t3),r                 leal  (edx,eax),eax
```
(a) (b)

```
movl  x,t1                      movl  edx,edx
addl  t1,(p)                    addl  edx,(ecx)
movl  y,t2             ⇒        movl  eax,eax
incl  t2                        incl  eax
movl  t2,r                      movl  eax,eax
addl  r,t1                      addl  eax,edx
```
(c) (d)

Figure 1: An example of the interaction between instruction selection and register allocation. Sequence (a) is shorter than sequence (c). However, register allocation can coalesce the move instructions in (c) resulting in a shorter post-allocation sequence. (d).

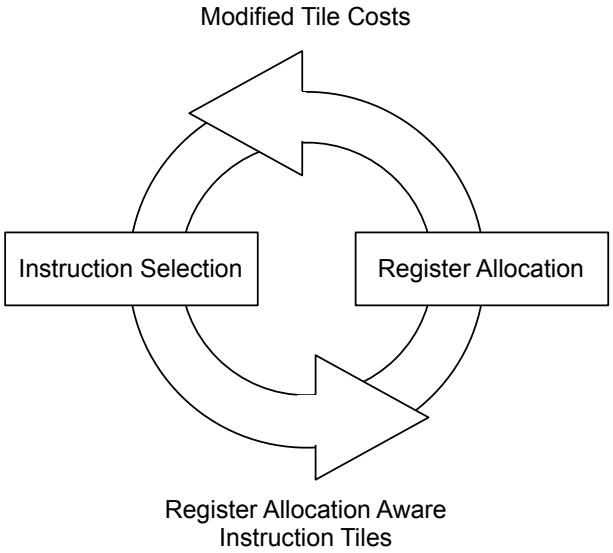
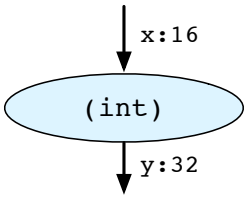


Figure 2: The feedback loop at the heart of RA^{2}ISE. The instruction selector selects Register Allocation Aware Tiles (RAATs), which enable the register allocator to make some instruction selection decisions at allocation time. Conversely, the results of register allocation are sent back to the instruction selector in the form of modified tile costs.



(a)

sign_extend $x \rightarrow y$

x\y	eax	edx	···	mem
eax	cwtl(1)	movsx(4)		
edx	movsx(4)	movsx(4)		
⋮				
mem	movsx(5)	movsx(5)		

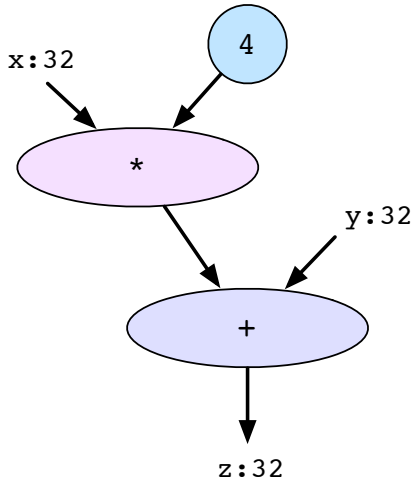
(b)

Figure 3: A register allocation aware tile for sign extension. The tile (b), matches the IR operation (a). The cost of the tile (here determined by code size) depends on the eventual allocation of the input (x) and output (y) of the operation. In some cases multiple instructions might be necessary. For example, if both x and y are in memory, a store instruction has to be generated. The register allocator is assumed capable of generating this store, and so the tile does not need to represent this case.

In order to enable communication between instruction selection and register allocation, the RA^{2}ISE framework utilizes Register Allocation Aware Tiles (RAATs). RAATs are more flexible than traditional instruction tiles:

- A RAAT does not necessarily correspond directly to a single instruction sequence. Instead, a RAAT represents several functionally equivalent potential instruction sequences and the final instruction sequence is chosen by the register allocator.
- A RAAT does not have a single fixed cost. Instead, its cost is modified based on the results of register allocation.

Like a traditional instruction tile, a RAAT represents a mapping from an expression tree intermediate representation to an assembly sequence. Unlike traditional tiles, this mapping is not one-to-one and there is no single fixed cost for the tile. Instead, the final assembly sequence and cost are determined by the results of register allocation. This dependancy is represented by a table relating the possible allocation class assignments of the inputs and outputs of the RAAT to the final assembly sequence and cost. An example of a sign extension RAAT is shown in Figure 3. This RAAT explicitly encodes the benefit of both operands being in `eax` (a smaller instruction can be used) and the cost of the input operand being in memory (an additional byte is necessary



(a)

plus (mult x,4), $y \rightarrow z$

z:eax

x\y	eax	edx	⋯	mem
eax		leal(3)		sall;addl(6)
edx	leal(3)			
⋮				
mem				

plus (mult x,4), $y \rightarrow z$

z:edx

x\y	eax	edx	⋯	mem
eax		leal(3)		
edx	leal(3)			sall;addl(6)
⋮				
mem				

⋮

(b)

Figure 4: A register allocation aware tile (a) for a more complicated expression tree (b). In this case it is impossible for x and y to be allocated to the same register since their live ranges overlap, but if y is in memory and x and z are allocated to the same register then y can be directly accessed with the `addl` instruction. Although this code sequence is no smaller than loading y into a register and using the `leal` instruction, it does require one less register. In RA^{2}ISE, the register allocator makes the final decision as to which sequence to generate.

to store the stack offset). Larger RAATs that potentially resolve to multiple instructions can also be used, as shown in Figure 4.

In the RA^{2}ISE framework instruction selection is first done using RAATs with optimistic costs. This result is then used by the register allocator to find an allocation. Given this allocation, each RAAT can then be transformed into a final instruction sequence which could potentially exceed the initial optimistic cost assumptions. Violations of the optimistic cost model are then relayed back to the instruction selector in the form of tile cost modifications and the feedback loop is repeated as necessary.

The success of the RA^{2}ISE framework depends on the expressiveness and flexibility of the supported RAATs and the ability of the register allocator to effectively incorporate RAATs into its model. The next section describes some limitations of the global MCNF model of register allocation when modeling truly expressive RAATs. Then an implementation of tile cost modification based on the results of register allocation is described and examined. Finally, overall results for the implemented RA^{2}ISE framework are detailed.

1 The Problem with RAATs

Ideally, the model used by the register allocator would be able to support arbitrary RAATs, such as those shown in Figures 3 and 4. That is, the relationship between RAAT operand allocations and the respective costs could be completely nonuniform with no particular structure. The global MCNF model of register allocation naturally supports tiles where the cost of an allocation decision is determined solely by an individual variable's allocation. For example, in the tile 3(b) the cost of allocating x to memory can be accurately represented. However, the cost of allocating x to `eax` can not be modeled exactly since this cost is determined by the allocation of y. In order to support more expressive RAATs, additional constraints must be added to the global MCNF model. Unfortunately, as we shall see, adding even a relatively simple constraint that ties the allocation of two variables together significantly reduces the solvability of the model.

move $x \rightarrow y$

x\y	eax	edx	⋯
eax	(0)	movl(2)	
edx	movl(2)	(0)	
⋮			

(a)

plus $x, y \rightarrow z$

z :eax

x\y	eax	edx	⋯
eax	addl(2)	addl(2)	
edx	addl(2)	leal(3)	
⋮			

z :edx

x\y	eax	edx	⋯
eax	leal(3)	addl(2)	
edx	addl(2)	addl(2)	
⋮			

(b)

Figure 5: Two RAATs representable in the global MCNF model using simple tied constraints. In these examples the cost of each instruction sequence is its size.

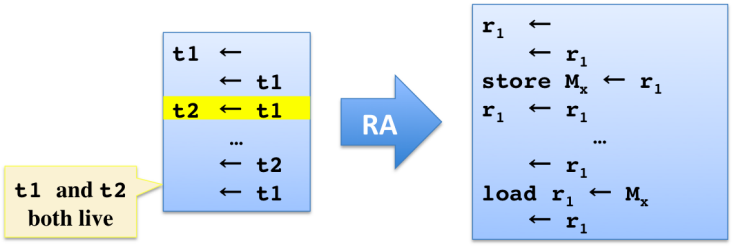


Figure 6: Two quantities may be tied together even if their live ranges conflict. In this example, `t1` and `t2` are both live after the highlighted instruction. However, if `t1` has been previously evicted to memory, it is still possible to allocate `t1` and `t2` into the same register at the highlighted instruction thus providing a benefit from a tied constraint.

1.1 Modeling Tied Constraints

A *tied constraint* simply states that there is a cost benefit if two variables (one an input operand and the other an output operand) are allocated to the same register. Two examples of RAATs that can be fully modeled in the global MCNF model through the use of tied constraints are shown

in Figure 5. In the case of a move instruction, Figure 5(a), if both the input and output operands of the move can be allocated to the same register than the instruction can be eliminated. When choosing an instruction selection for a simple addition, Figure 5(b), the smaller `addl` instruction can be used if the output operand (z) is allocated to the same register as either one of the input operands (x or y). Note that in both these cases, the tied constraint may still be satisfiable even when the tied input operand is live out of the instruction if the value in that operand has been previously stored to memory as shown in Figure 6. In order to effectively optimize for tied constraints, the full context of register allocation is needed. Although implementing tied constraints is not sufficient to support arbitrary RAATs, implementing these simple constraints is the first step towards supporting more complex, expressive constraints, and an evaluation of the consequences of implementing these simple constraints is useful in evaluating the advisability of implementing more complex, expressive constraints.

Tied constraints are incorporated into the global MCNF data structure by adding an annotation to an instruction group node that describes the benefit of allocating two operands to that same node. More formally, the term $-tc^j_{benefit}tc^j$ is added to the objective function and the following constraints are added to the integer linear programming model:

$$tc^j \leq x^{q_1}_{ij}$$

$$tc^j \leq x^{q_2}_{jk}$$

Here q_1 and q_2 are the two variables (commodities) being tied together at instruction node j; q_1 is the input operand, traversing edge x_{ij}, and q_2 is the output operand, traversing x_{jk}. The new quantity tc^j can only be one if both q_1 and q_2 are allocated to the same node j. The benefit of q_1 and q_2 being tied together is represented by $tc_{benefit}$ in the objective function. Since the goal is to minimize the objective function, tc^j will be one whenever q_1 and q_2 have the same allocation.

The addition of tied constraints to the global MCNF model of register allocation presents several problems. Since these constraints are not hard constraints (all solutions that are feasible in the original model are still feasible in the expanded model), there is no need to relax them. Although this simplifies the task of finding feasible solutions, it also means that Lagrangian relaxation cannot be used to push the found solutions closer to optimal, as is the case with

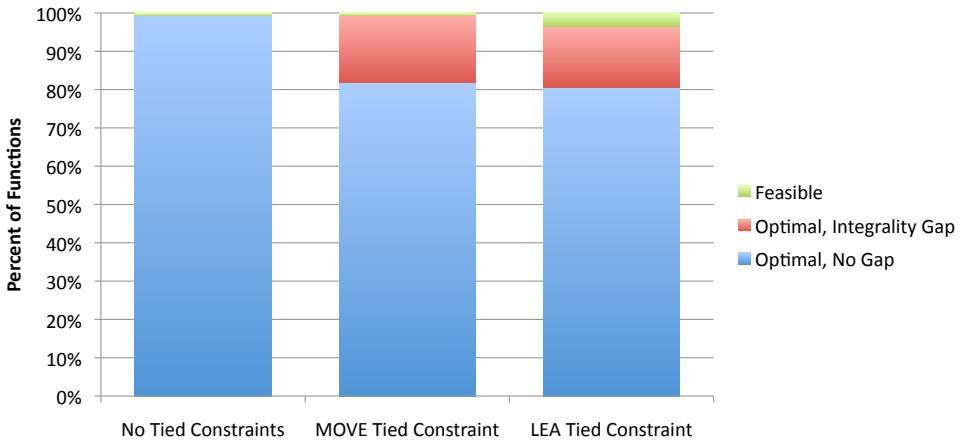


Figure 7: Optimality gap introduced when tied constraints are added to the model. For all the functions in a small benchmark suite the optimal solution is calculated for the original global MCNF model, the model with tied constraints implementing the move RAAT of Figure 5(a), and the model with tied constraints implementing the `lea`/`add` RAAT of Figure 5(b). Approximately 20% of the functions had an integrality gap. In some cases, likely due to the presence of an integrality gap and the need to perform branch and bound search, only feasible, not provably optimal, solutions were found by the CPLEX solver within the hour time limit when tied constraints were present.

the bundle constraints in the model. More importantly, even if a highly effective heuristic for satisfying these constraints could be developed, the existence of these constraints fundamentally alters the nature of the model.

1.2 Solving Tied Constraints

One of the advantages of the global MCNF model of register allocation coupled with progressive solution techniques is that a meaningful upper bound on the optimality of a solution can be computed. This is possible because, empirically, there is rarely an integrality gap; that is, the value of the linear relaxation is equal to the integer solution. Unfortunately, when tied constraints are added to the model this property disappears. This is demonstrated empirically in Figure 7. We consider a subset of the MediaBench [10] benchmark suite consisting of the benchmarks dijkstra, g721, mpeg2, patricia, qsort, sha, and stringsearch. The reduced size of this benchmark set allows us to use ILOG CPLEX [5] to find optimal solutions to the register allocation problem when targeting 32-bit x86 and optimizing for code size. Using the original global MCNF model,

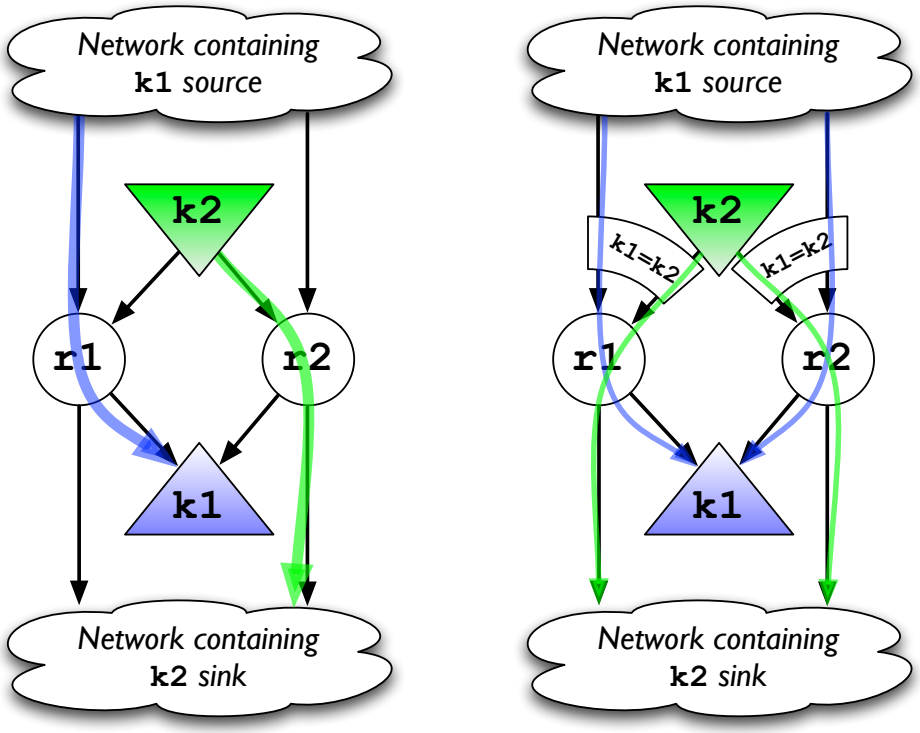


Figure 8: Tied constraints introduce an integrality gap. In this example an optimal solution for the flows of `k1` and `k2` can be achieved using only integral flows unless tied constraints are added to the problem. The existence of a tied constraint decreases the optimal value of the solution by the benefit of the tied constraint and necessitates that fractional flows be used in the optimal solution.

none of the functions compiled had a definite integrality gap (although in one case the solver time limit expired before a provably optimal solution was found). However, when tied constraints are added to the model approximately 20% of the functions compiled have an integrality gap. The introduction of these integrality gaps results in an order of magnitude increase in the amount of time it takes to find an optimal solution using CPLEX. The progressive solution techniques used by the register allocator are also negatively affected by the presence of integrality gaps since tight lower bounds cannot be computed.

Unfortunately, the introduction of integrality gaps is fundamental to the nature of the constraint. Consider the case where in the optimal (integer) solution it is undesirable to satisfy the tied constraint. This means that the cost of assigning the two variables of the constraint to the same register is greater than the benefit. However, in the linear relaxation of the problem, it is

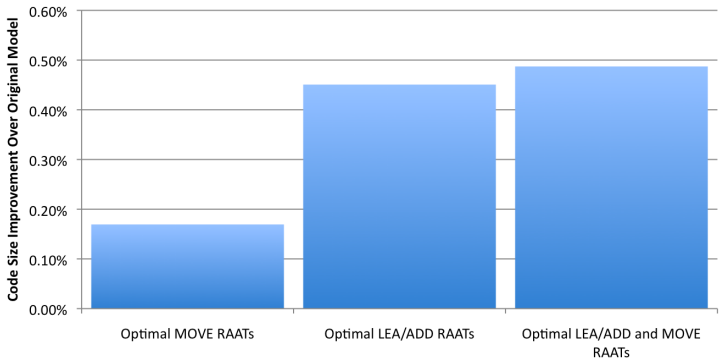


Figure 9: Code size improvement of models with tied constraints. Adding tied constraints to the register allocation model to represent both move RAAT of Figure 5(a) and the `lea`/`add` RAAT of Figure 5(b) results in code size improvements when the model is solved to optimality across a small benchmark suite. The baseline model always implements simple additions with an `add` instruction since this results in better average code size than when a `lea` instruction is always used.

possible to modify the integer optimal solution to satisfy the tied constraint without incurring any additional costs. As an example, consider the following scenario, which is illustrated in Figure 8. Assume that in the integer optimal solution, variables k_1 and k_2 are allocated to distinct registers r_1 and r_2 respectively. If k_1 and k_2 are part of a tied constraint tc^j, then the cost of the linear relaxation can be improved by $tc^j_{benefit}$ simply by dividing the flows of k_1 and k_2 equally between r_1 and r_2. That is, k_1 is allocated to half of r_1 and half of r_2. Unless there is a strong preference in the model for k_1 to be in r_1 or k_2 to be in r_2, which is uncommon, this will improve the cost of the linear relaxation resulting in an integrality gap.

The integrality gaps resulting from introducing tied constraints into the model have a definite negative impact on the solvability of the model. However, the addition of tied constraints does have a small, but not insignificant, impact on the quality of the solution. As shown in Figure 9, the addition of each RAAT type to the model results in a fraction of a percent improvement in code size although the combined effect of the different RAAT types is subadditive. This suggests that a model that can explicitly and exactly represent fully expressive RAATs would have advantages over the unmodified global MCNF model. Unfortunately, supporting fully expressive RAATs drastically impacts the solvability of the global MCNF model. Therefore, an entirely

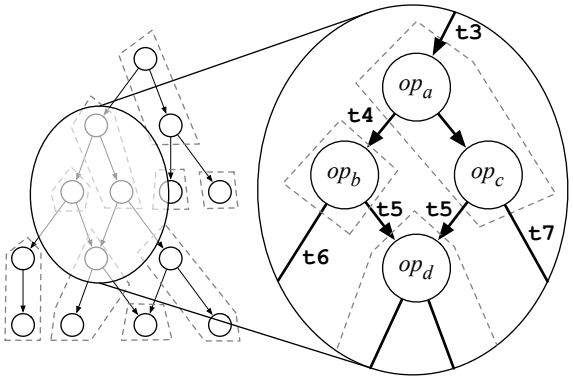
new modeling framework must be developed, less principled ad-hoc approaches have to be used, or heavy-weight solution techniques, such as branch and bound, must be used.

The global MCNF model of register allocation is a natural and intuitive representation that is amenable to effective progressive solution techniques. Although tied constraints do not integrate well into the model, its many other advantages outweigh this limitation. It is worth noting that this limitation is not unique to the global MCNF model. No other register allocation framework fully and expressively integrates the benefits modeled by tied constraints. Instead, ad-hoc heuristic techniques are used. An ad-hoc heuristic approach is equally applicable to the global MCNF model when using a heuristic solver. Allocation decisions that satisfy a tied constraint are simply given a preference over similarly priced alternative allocation decisions. This is a not a principled approach since the benefits of the tied constraints are not explicitly modeled and the cost/benefit tradeoff of satisfying the tied constraint is not explicit. Therefore, it is not surprising that this approach results in decidedly mixed results. In fact, when evaluated on the benchmark suite of Figure 9 there is an overall slight degradation of code quality (0.03% larger). As an alternative to a heuristic approach, heavy-weight solution techniques, such as branch and bound, could be used. These approaches, which are all variants on exhaustive search, dramatically increase compile time and are only appropriate when code quality and optimality guarantees are of utmost importance.

We conclude that a practical principled framework for incorporating fully expressive RAATs into register allocation requires a significant advancement beyond the current state-of-the-art. However, it is possible that some of the benefit of the proposed RA^2ISE framework can be gained by providing feedback from the register allocator to the instruction selector. This is done by having the register allocator modify the instruction tile costs.

2 Modifying Tile Costs

The NOLTIS algorithm performs near-optimal linear-time instruction selection. Given a set of tiles that represent machine instructions, the NOLTIS algorithm finds an optimal or close to optimal tiling of an expression DAG. In order for an optimal tiling to correspond to an optimal

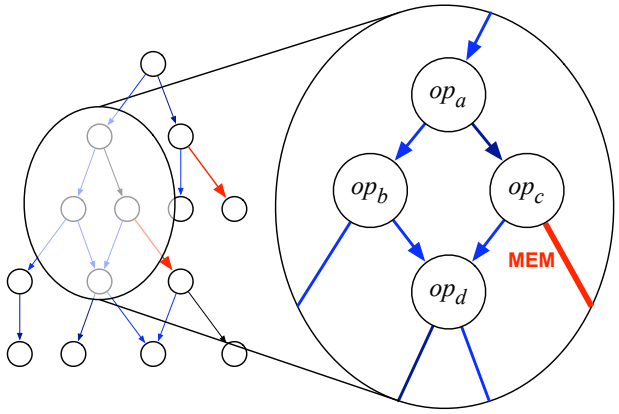



```
...
op_d  t5 ←
op_b  t4 ← t5,t6
op_ac t3 ← t4,t5,t7
...
```

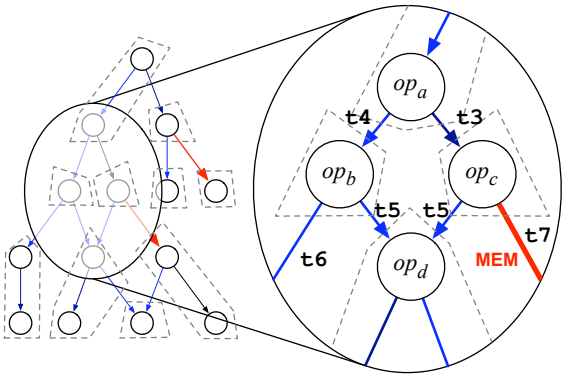
(a)

```
...
op_d  r1 ←
op_b  r2 ← r1,r2
load  r3 ← M7
op_ac r1 ← r2,r1,r3
...
```

(b)



(c)




```
...
op_d  t5 ←
op_b  t4 ← t5,t6
op_c  t3 ← t5,t7
...
```

(d)

```
...
op_d  r1 ←
op_b  r2 ← r1,r2
op_c  r2 ← r2,M7
...
```

(e)

Figure 10: Feedback directed instruction selection. (a) Instruction selection and (b) register allocation are performed as usual in the first phase. (c) The results of register allocation are then back annotated onto the expression DAG indicating what edges are likely to carry values that are in memory. (d) In the second phase, instruction selection is performed on this modified expression DAG using tile costs that more accurately reflect the likely results of register allocation. (e) Register allocation is then performed using the new register-allocation aware instruction sequence.

instruction selection, the tile costs must accurately reflect the realities of the final code quality metric. Unfortunately, when viewed in a larger context, the tile costs used by the instruction selector are inherently inaccurate. NOLTIS, and other instruction selection algorithms, assume an idealized machine model with an infinite register set. In actuality, the cost of instruction tiles should include costs related to register allocation, such as spill costs and the cost of memory operands. The RA^2ISE framework attempts to resolve this discrepancy by allowing for feedback from the register allocator to the instruction selector. In this section we investigate the practicality and effectiveness of register allocation provided feedback on instruction selection.

In order to implement a feedback driven system it is necessary to execute at least two passes of instruction selection and register allocation. The first pass performs a preliminary selection and then the register allocator determines the register or memory assignment of variables. These locations are then mapped back onto the corresponding edges in the original expression DAG. If there is no location assigned to an edge in the expression DAG (because the edge is internal to a first phase instruction tile) then the value for this edge is extrapolated from the connected edges. These edge annotations are then used to refine the costs of tiles. For example, if a value is in memory and an instruction cannot directly access memory, the cost of a load is included in the total cost of the tile. Alternatively, if the instruction can directly access memory, then only the cost of incorporating a memory operand into the instruction is included in the tile cost. A second pass of instruction selection and register allocation is then performed using these modified tiles. Since the second phase of register allocation may have a different result than the first phase, the modified tile costs are not perfectly accurate. However, it seems reasonable to assume that these feedback derived costs will be more accurate than the unmodified costs. The complete tool-flow is shown in Figure 10.

We have implemented a prototype feedback-directed instruction selection system within LLVM 2.1 [11] using the NOLTIS instruction selector. In our prototype system we limit ourselves to considering only costs relative to values being allocated to memory. Because of its CISC instruction set and limited register set we target the 32-bit x86 architecture. We optimize for code size since improved instruction selection can reduce code size significantly. The prototype system was used to compile the Mibench [3] benchmark suite. Despite the improved tile

costs derived from the register allocation provided feedback, the resulting code size improvement of our system was a microscopic 0.0015% with only three of 6147 functions showing any improvement. Additional iterations of the feedback loop yielded zero additional benefit. Clearly, despite the presence of more accurate tile costs, feedback-directed instruction selection as implemented in our system provides essentially zero benefit.

The cause of this disappointing result becomes clear when the behavior of the instruction selector is examined. Although the tile costs are properly modified, in almost all cases the modified costs do not change the resulting tiling. This is in large part due to the relatively simple single-instruction tiles used and the design of the x86 ISA. For most tile decisions either none of the tiles can directly access memory, or the best tile is already capable of accessing memory. In these cases, modifying the tile costs to reflect the cost of an operand being allocated to memory does not change the outcome of the tiling. It is possible that if more complex tiles, such as the RAATs described in Figure 4, were used then feedback-direct instruction selection would be more successful. Unfortunately, as described in Section 1, such RAATs are not practical within our register allocation framework. Considering the extra compile-time overhead of executing two full passes, feedback-directed instruction selection is unlikely to have much practical value unless a RAAT-friendly framework can developed.

3 Summary

In this report we explored the feasibility of tightly integrating instruction selection and register allocation. We described an integration framework, RA^2ISE. Unfortunately, we discover that the fundamental building block of the RA^2ISE framework, register allocation aware tiles (RAATs), introduce significant complexity into the network flow model of register allocation. It is unlikely that efficient and effective solution techniques exist when RAATs are incorporated into the model. We also explore the merits of another component of the RA^2ISE framework, feedback driven instruction selection and find that the expected benefits are far outweighed by the necessary costs.

References

[1] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 231–239, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-806-7. doi: http://doi.acm.org/10.1145/997163.997196. (document)

[2] G.G. Fursin, M.F.P. O'Boyle, and P.M.W. Knijnenburg. Evaluating iterative compilation. In *Proc. Languages and Compilers for Parallel Computers (LCPC)*, pages 305–315, 2002. (document)

[3] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. *IEEE International Workshop on Workload Characterization*, pages 3–14, December 2001. 2

[4] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 123–132, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2429-X. doi: http://dx.doi.org/10.1109/PACT.2005.9. (document)

[5] ILOG. ILOG CPLEX. `http://www.ilog.com/products/cplex`. 1.2

[6] David Koes and Seth Copen Goldstein. Near-optimal instruction selection on dags. In *CGO '08: Proceedings of the International Symposium on Code Generation and Optimization (CGO'08)*, 2008. (document)

[7] David Ryan Koes. *Towards a More Principled Compiler: Register Allocation and Instruction Selection Revisited*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, October 2009. (document)

[8] David Ryan Koes and Seth Copen Goldstein. A global progressive register allocator. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 204–215, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-320-4. doi: http://doi.acm.org/10.1145/1133981.1134006. (document)

[9] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 12–23, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-647-1. doi: http://doi.acm.org/10.1145/780732.780735. (document)

[10] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *ACM/IEEE*

International Symposium on Microarchitecture, pages 330–335, 1997. URL `http://www.icsl.ucla.edu/~billms/Publications/mediabench.ps`. 1.2

[11] LLVM. The LLVM compiler infrastructure. `http://llvm.org`. 2

[12] Andy Nisbet. Gaps: A compiler framework for genetic algorithm (ga) optimised parallelisation. In *HPCN Europe*, pages 987–989, 1998. (document)

[13] Spyridon Triantafyllis, Manish Vachharajani, and David I. August. Compiler optimization-space exploration. *The Journal of Instruction-Level Parallelism*, 7:1–25, January 2005. URL `http://www.jilp.org/vol7`. (document)