Profile Based Register Allocation

Robert Shih, Paul Sastrasinh

1 Introduction

Register allocation in modern compilers are commonly based on graph coloring. Each node represents a virtual register, and edges are drawn between nodes that are live simultaneously at some program point. Neighbors in the graph must not be assigned to the physical register. When a valid assignment cannot be found using certain heuristics, some virtual registers are spilled, i.e. assigned to memory slots, and removed from the graph.

Virtual registers are assigned weights based on heuristics that model the extra memory latency. Candidates with smaller weights are preferred spill candidates. Traditional weight assignment heuristics are static and deterministic. The LLVM allocator for example assigns higher weights to induction variables and deeper nested variables, and lower weights to definitions that are trivial rematerialize. This document describes a weight assignment strategy that is both dynamic and non-deterministic.

The proposed allocator uses profile information as a first pass approximation of spill weights. Weights are then scaled and shifted by random coefficients drawn from a normal distribution. Assignment histories are persisted across compilation runs to guide future allocations.

2 Related Work

Wall described the use of profile information in his link time global register allocator[5]. Commercial compilers from Microsoft[2] and Intel[1] both feature profile guided allocators. Chen et al. proposed hardware based profiling in [3]. Hardware based methods are less intrusive and allows existing projects to be built and run as is; on the downside hardware profilers only *`sample`* the target program, resulting in less complete and less accurate statistics. We do not know of existing register allocation approaches that explores non-deterministic spill weight assignments.

3 Design

We decided to implement the proposed allocation scheme, which we would refer to later as `prof`, as an extension to the LLVM project.

First `prof` instruments the target program with profile code. After a sample run basic block execution counts are used to bootstrap the initial weight assignments. Profiling is done using LLVM's builtin `-insert-edge-profiling` class, which happens at the bitcode level. During the initial spill weight assignment phase, profile output is read back,

and basic block execution counts are aggregated from each incoming control flow edge. For each live interval, the `prof` allocator collects the native blocks spanned within the definition and use. `prof` then maps each native block back to its bitcode counterpart and aggregates the execution counts. The aggregated count is used as the initial spill weight.

`prof`then performs a series of recompilations. In each round of compilation, it scales the original spill weight with a factor drawn randomly from a normal distribution. The new factors are persisted to aid future allocations. Basic blocks in LLVM are identified by memory addresses, e.g. in map based lookups. To provide consistent basic block identification across multiple compiler runs, `prof` added a new pass that associates each basic block with a persistable unique identifier. Two new passes are inserted before and after register allocation respectively to load and store weight assignments to the file system.

4 Evaluation

To evaluate our profile based allocation, we test our profile based allocator compared to the baseline allocators on each of LLVM's built in register allocators. Note that by default, LLVM approximates the weight of live ranges using a heuristic based on loop depth. These allocators include the basic allocator, greedy allocator and the partitioned boolean quadratic programming (PBQP) based register allocators. We keep the weighting scheme constant for each of the profile allocation schemes and run our tests on a variety of small benchark problems as well as some larger more complex code.

The basic allocator is a simple allocator which sorts live ranges by decreasing spill weight and spills registers in that order. One downside to the basic allocator is that it does not differentiate between small and large live ranges, so small ranges may be spilled first, requiring more spilling than necessary. The greedy allocator is similar to the basic allocator except it allocates larger live ranges first. Finally, the PBQP allocator is an implementation of the algorithm described in [4].

We ran each of our benchmarks ten times and averaged the results. Our benchmark results are summarized in the figure 1 and table 1 below.

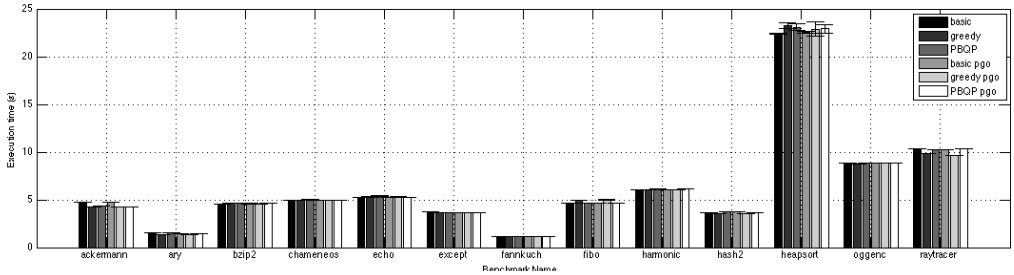

Figure 1: Average benchmark times and their variance (over 10 runs) for each of the allocators.

Our results show a small improvement over the baseline allocators which do not use profile information (about .31% improvement in time). While this might seem like a very small improvement and is probably subject to the set of benchmarks chosen, it is in line

	Basic	Greedy	PBQP
ackerman	0.2470	-0.2305	0.7565
ary	1.2629	-0.5525	1.5291
bzip2	-0.7236	1.1066	-0.3888
chameneos	0.2951	0.2004	0.8185
echo	0.6506	0.2853	2.1688
except	0.5394	0.2407	-0.6425
fannkuch	-1.6252	0.8330	2.1421
fibo	-0.1474	-0.9891	-0.7108
harmonic	-0.4711	-0.2126	0.0348
hash2	-0.2338	-0.4756	1.4722
heapsort	-0.4785	1.6145	0.6931
oggenc	0.0811	-0.5608	0.1734
raytracer	0.8283	1.7374	-0.2737
mean	0.0173	0.3155	0.5979

Table 1: Percentage improvement of our profile based allocators over baseline (default LLVM allocators).

with improvements offered by the different LLVM allocators. For example, comparing the non profile guided basic to PBQP allocators, we would expect that the PBQP would perform better, since it attempts to smartly optimize the method in which registers are allocated - unlike the basic allocator. This however, turns out to not be the case (see table 2), and in fact, performance averaged over all our benchmarks show there is almost no gain at all.

ackerman	ary	bzip2	chameneos	echo	except	fannkuch
8.6682	0.4301	-1.2253	-0.8169	-2.2989	1.1708	-3.3358
fibo	harmonic	hash2	heapsort	oggenc	raytracer	**mean**
-0.0313	-0.8552	-1.5861	-2.9737	-0.0664	0.6993	-0.1709

Table 2: Percentage improvement of non PGO based PBQP allocation versus basic allocation.

We also tested are randomized weight allocator, which attempts to find the best weight values given profile information (by randomly perturbing the live range weights. The results of our semi randomized scheme on LLVM's basic allocator can be seen in table 3 below.

The margin of improvement in our randomized allocation scheme is on the same order as the improvement we saw in the normal profile guided allocation scheme versus the non profile guided scheme. Again, the margin of improvement is rather small, although in this case, most of the programs seem to run a marginally faster.

	ackerman	ary	bzip2	chameneos	echo	except	fannkuch
no pgo	4.6960	1.4806	4.4953	4.8979	5.1955	3.6566	1.0917
random pgo	4.6700	1.4660	4.4886	4.9235	5.1461	3.6521	1.0914
% change	0.3611	0.9835	0.1479	-0.5229	0.9503	0.1236	0.0291
	fibo	harmonic	hash2	heapsort	oggenc	raytracer	**mean**
no pgo	4.5843	6.0109	3.6439	22.1228	8.7337	10.2589	
random pgo	4.5652	6.0030	3.6179	21.9901	8.7383	10.1150	
% change	0.4158	0.1308	0.7128	0.6000	-0.0532	1.4026	0.4063

Table 3: Time taken to complete for LLVM's basic allocator. The time reported is the minimum time in seconds over ten seperate runs.

5 Discussion

From our experiments we saw very small improvements over the basline non profile guided allocators. While the gains were small, the extremely small variance between different runs leads us to believe that the gains were significant. In fact, it is not surprsing that the gains are only on the order of tenths of a second for many of our benchmarks. As described in the experiments, simply comparing LLVM's basic allocator to the state of the art (but not profile based) PBQP allocator showed practically no improvement - PBQP was in fact, on average worse than the basic allocator for our benchmarks. Furthermore, spilling a register is stil a relatively cheap operation, and in order to realize any user (human) noticable gains in speed would probably require complex programs running for long periods of time. Our benchmarks span several different complexities to try and account for this - some of the more complex benchmarks include bzip2 (a compression algorithm), and oggenc (the ogg audio encoder), however it is still difficult to say whether these really affect the results significantly.

The minor gains resulting from profile based register allocation make it difficult to prefer over other non profile based allocation methods. This is especially true since the time taken to instrument the program for data can be quite long (depending on the program), leading to much longer compilation pipelines. It is also difficult to tell in advance whether or not profile based allocation will offer any benefits at all to a program (although on average this seems to be the case, but it is a very small gain). Furthermore, it seems that even the most complex non profile allocator (PBQP) provides little noticable benefit over less complex allocators, such as the basic allocator at the cost of higher computation time.

6 Future Work

Currently the process of identifying the best allocation weight across multiple compilation runs is manual. Ideally the compile-log-profile cycle should be automated to converge towards a local optimum. A plausible next step is to model the relation between basic block weights and program execution time as a linear program, and solve for the optimal assignments. One can naturally follow with experiments on more sophisticated mathematical models. It would also beinteresting to try and evalute the allocation schems on longer and more complex benchmarks which might reveal more significant differences

between the allocation schemes.

One of the major problems of profile based optimizations is that it requires profiling information, which can be difficult to collect. Furthermore, if the program being profiled depends on the input data, changes to input data in the future might cause the profiling information to be incorrect, potentially causing programs to run slower. In these cases it might be better to make no assumptions about the data (ie. profiling information). A more interesting idea, however, might be to try and find places which should be optimized and spilled subject to the assumption of changing input data - maybe by randomizing the input data and exmining the difference in edge visit and basic block visit counts.

Another direction of future work involes developing a less intrusive profiling build system. The current design requires us to manually modify the build scripts and integrate the modified tool chain. Manual modification is not scalable in experiment setups and unlikely to be adopted by existing software projects.

References

[1] Intel C++ Compiler XE 13.1. Profile-guided optimizations overview. `http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/GUID-42C5C93F-5F77-4B14-8A27-95798BC30CE3.htm`, January 2013.

[2] Visual Studio 2012. Profile-guided optimizations. `http://msdn.microsoft.com/en-us/library/vstudio/e7k32f4k.aspx`, April 2013.

[3] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for fdo compilation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 42–52, New York, NY, USA, 2010. ACM.

[4] Lang Hames and Bernhard Scholz. Nearly optimal register allocation with pbqp. In *In Proceedings of the 7th Joint Modular Languages Conference (JMLC'06). LNCS*, pages 346–361. Springer, 2006.

[5] David W. Wall. Global register allocation at link time. In *Fast Printed Circuit Board Routing. WRL Research Report 86/3*, pages 264–275, 1986.

7 Work Distribution

The approximate work distribution was 50%-50%, with Paul working on the profile guided register allocation and benchmarks, Robert working on the randomized allocation and both contributing to the reports.