Literature review: Superoptimizers

Tom Hume, T.Hume@sussex.ac.uk
November 2011

Contents

Background

Superoptimization is a term coined by Massalin (1986) which embodies the Ken Thompson saying, "*When in doubt, use brute force*"; where conventional compilers often define optimization as improving the code produced, superoptimization looks for the literally optimal output. Broadly speaking superoptimization involves the following phases:

1. Gathering input for the superoptimizer;
2. Pruning the search space of potentially optimal output candidates;
3. Generating these candidates;
4. Testing these candidates to determine their equivalence to desired code;
5. Applying a cost function to determine a candidates optimality.

Of this list, the second and fourth are typically seen as key challenges.

The technique has so far been applied to optimization of small but important sections of code and in efficiently automating translations of binaries between architectures. It suggests the possibility of truly optimal implementations of algorithms, through a pruned brute-force search of candidate code sequences.

Literature

To gather literature on superoptimization I searched on Google Scholar for the terms "superoptimiser" and "superoptimizer", in "Engineering, Computer Science, and Mathematics" articles only. These terms returned 140 and 152 matches respectively. Of these I read 6 of the top full papers across both searches, and followed up 1 further

reference found in these papers. In the interests of consistency, I have resignedly used the US spelling of “superoptimizer” throughout this document.

Findings

When preparing his first paper on the topic, Massalin (1986) produced a demonstration superoptimizer which takes 68000 machine code as input. His implementation exhaustively generates code sequences of increasing length. These sequences are then filtered to remove those which could not occur in an optimal code sequence (using n-dimensional bit tables to implement the filter). Sequences which pass this stage are tested for equivalence to the original code using a probabilistic test - i.e. a few manually chosen values designed to quickly exclude poor matches. Those that pass the probabilistic test are manually examined for correctness. To cut down the search space Massalin limits himself to register-to-register problems, avoiding code sequences involving pointers as they “*can point anywhere in memory and so to model a pointer in terms of boolean expressions one needs to take all of memory into account*“ - a complexity he sought to avoid.

5 years later, Granlund built on Massalin’s work and presented the GNU Superoptimizer# (Granlund, 1992), which applied superoptimization techniques to code generation for conditional jumps in the GNU C compiler, working on specific goal functions: a more focused use of the technique than Massalin proposed, but very practical. Granlund cut down the search space of possible code sequences by using a "*recursive iterative-deepening method, and avoids generating clearly useless instructions that would otherwise have to be pruned later*” (Granlund 1992, p.9). Granlund echoed Massalin's desire to keep the search space small, in order to allow it to generate longer code sequences.

In 2002 Joshi, Nelson and Randall proposed Denali, a superoptimizer which sought to prune the search space radically by using a refutation-based automatic theorem prover. They considered such a prover to be “*a general-purpose goal-directed search engine, which can perform a goal-directed search for any- thing that can be specified in its declarative input language. Successful proofs correspond to unsuccessful searches, and vice-versa*”. A side-effect of this approach ought to be the production of nearly mathematically optimal machine code. Using Denali the authors saw large gains in efficiency, generating a near-optimal 31-instruction sequence in 4 hours, and avoided the need for manual checking of output code sequences.
Massalin had previously made the observation that “*the shortest programs are surprising, often containing sequences of instructions that one would not expect to see side by side*” (Massalin 1986, p.124), and Granlund similarly found that even the architects of his target processor, the RS/6000, were surprised by the results his superoptimizer produced. The need for Denali to be pre-populated with axioms which it uses to goal-direct its search limits the range of its potential output: a key weakness of the approach, summarised nicely by Bansal and Aiken in their 2008 paper: “*we didn’t want to rule out optimizations simply because we hadn’t thought of them*” (Bansal and Aiken 2008, p.9).

Bansal and Aiken have produced a couple of works on the topic. In 2006 they looked at applying superoptimization to peephole optimizers, in a paper which also proposed a novel architecture for compilers: that they effectively use large-scale look-up tables to search for replacement rules which had previously been calculated, mapping input code

sequences to their optimal versions. This led to their envisaging a system "*much closer to a search engine database than to a traditional optimizer*" (Bansal and Aiken, 2006). Their 2006 work took a less far-sighted and more pragmatic approach, superoptimizing many target sequences first, canonicalising potential target sequences to ensure that they are matched quickly against any appropriate optimization (thus reducing the search space), and producing as their output a set of replacement rules. Their paper is not clear on the length of target instruction sequences they considered for superoptimization (all of which were harvested from a representative set of real-world binaries), and their experience with finding good heuristics for testing candidate optimizations directly contradicted Massalin's (they found many sequences that could pass an execution test without being correct) (Bansal and Aiken 2006, p.6) but nonetheless their technique led to speed-ups of a factor between 1.7 and 10 between original and superoptimized code sequences. They also made progress with the pointer problem Massalin identified, by sand-boxing memory accesses and redirecting them into known and tracked memory locations (Bansal and Aiken 2006, p.4) - thus increasing the corpus of code ripe for superoptimization.

In 2008 they followed up with a second paper, this time applying superoptimization to a new area: automating binary translation between different machine architectures (Bansal and Aiken, 2008). Here they claimed results equivalent to, or in excess of, other binary translation products like the open source QEmu and Apple's Rosetta, though their approach is one of static translation up-front. Rosetta and QEmu are dynamic in nature, giving their optimizers the advantage of knowing machine state at the time of optimization, whilst static translators can carry out more far-reaching optimizations at compile time (potentially over the whole program). Nevertheless, they observed performance improvements even on already-optimized binaries, suggesting that "*even mature optimizing compilers today are not producing the best possible code*" (Bansal and Aiken 2008, p.11). It's interesting to note that their analysis of binaries showed that 99.99% of execution time was spent within around 2% of the instructions in a given binary - empirical evidence in favour of the case for concentrating superoptimization effort on key routines.

Also in 2008, Sayle presented a more limited use case for superoptimization: efficient code generation for multi-way branching (e.g. switch-statements) (Sayle, 2008). His approach to this situation was a very specific implementation, based around considering control flow graphs as binary trees. Sayle (2008, p.111) notes that techniques used by modern processors like advanced branch prediction and on-processor caching make effective benchmarking of this kind of code (which frequently uses jump tables which might end up stored in caches) challenging, and the results of such benchmarking therefore potentially skewed or biased.

Superoptimization has thus far received little attention beyond the core literature of the papers described above. When considering cost functions (so as to determine what to superoptimize for), Sayle references Tiwari (1994) in discussing the possibility of energy-efficiency as a possible considering for one such function.

The table below summarises different approaches to the key problems pruning of search space and equivalence testing, taken by various authors:

Author	Year	Pruning technique	Equivalence technique

Massalin	1987	Filter with n-dimensional bit tables, either manually or automatically populated	Probabilistic execution test with known sample inputs, manual testing
Granlund	1992	Prune operands without inputs to the sequence or generated by previous instructions, look for use of carry bit	Probabilistic execution test with known sample inputs, with manual testing recommended
Joshi/Nelson/Randall	2002	Binary search problem space using prover	Refutation-based automatic prover, fed with axioms
Bansal and Aiken	2006	Canonicalisation of candidate sequences	Probabilistic execution test with known sample inputs, then SAT solver of boolean expression of test
Bansal and Aiken	2008	(refers back to 2006 paper)	Probabilistic execution test with known sample inputs, then SAT solver of boolean expression of test
Sayle	2008	Restricts to optimising control flows represented as binary trees	Sayle is silent on his test function, concentrating on cost functions instead

Conclusions

Despite having demonstrated real-world applications in two areas (peephole optimization in compilers and efficient automatic binary translation), and the increasingly available of large-scale computing resources in the decades since it was first proposed, superoptimization still feels like a “fringe” topic receiving sporadic attention.

Much effort in superoptimization has been “in the small”: looking at specific types of compiler output (multi-way branches in the case of Sayle, 2008, and branch elimination in the case of Granlund, 1992).

The prospect of using formal proofs to validate superoptimized code (or, as Denali does, limit the search space) is attractive, but axiom-based approaches to this deprive superoptimization of one of its advantages, the discovery of unexpected code sequences.

Further research

A number of directions for further research would naturally follow from the work done to date:

1. Identifying approaches for formally proving equivalence of candidate code sequences, such that testing of larger sequences or greater numbers of sequences can be carried out without manual intervention. Ideally these approaches would address the issues introduced by being informed by a limited set of axioms;

2. Identifying better approaches for pruning the search space of possible code sequences, without restricting the techniques these code sequences may employ to those envisaged by human designers;

3. More sophisticated cost functions would be useful. Energy-efficient cost functions, as suggested by Sayle, 2008 could deliver value either in small-scale embedded systems where energy supply is naturally limited, or in large-scale computing infrastructure where small efficiency gains can deliver large aggregate effects.

4. Cost functions based on the actual performance of target hardware may be more realistic than those to date, which are typically based on either minimising the number of output instructions, or on deriving the speed of individual instructions data from technical manuals (thus losing the often-noted ability of the superoptimizer to surprise);

5. More work on realising the Bansal & Aiken 2006 vision of a compiler modelled around a "search engine" metaphor;

6. Work to ensure that equivalence tests of candidate code sequences fail faster: perhaps certain types of test, in certain situations, would be more useful to prioritise, and a superoptimizer might vary its approach to testing to aim for earlier failure in every case.

Acknowledgements

I'd like to thank Des Watson for his assistance in preparing this review, and for introducing me to the topic.

References

BANSAL, S. and AIKEN, A. (2006) Automatic generation of peephole superoptimizers. *ACM SIGPLAN Notices*

BANSAL, S. and AIKEN, A. (2008) Binary translation using peephole superoptimizers. In: *OSDI*

’08: Proceedings of the 8th conference on Symposium on Operating Systems Design & Implementation pp 177-192

GRANLUND, T. (1992) Eliminating branches using a superoptimizer and the GNU C compiler. *ACM SIGPLAN Notices*

JOSHI, R., Nelson, G., and Randall, K. (2002) Denali: a goal-directed superoptimizer. In: *PLDI '02 Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation* New York: ACM

MASSALIN, H. (1987) Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News*, pp. 122-126

SAYLE, R.A. (2008) A Superoptimizer Analysis of Multiway Branch Code Generation. In: *GCC Developers' Summit, 2008* Ottowa pp 103-116

TIWARI, V., MALIK, S. and WOLFE, A. (1994) Power Analysis of Embedded Software: A
First Step Towards Software Power Minimization. In: I*EEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 2, No. 4* pp. 437–445