Modern Compiler Optimizations
An Annotated Bibliography

Timothy VanSlyke

March 23, 2018

References

[1] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 305–317. [Online]. Available: http://dl.acm.org/citation.cfm?id=3049832.3049865

Proposes method of generating code for software-level cache prefetching in contexts where hardware prefetching is deemed insufficient. Focuses on reducing memory latency in contects where indirect memory accesses typically yield poor cache efficiency (linked-lists, trees, graphs, etc). All code generation is done through static analysis at compile-time (no JIT). Implemented in LLVM - benchmarked agains manually-inserted prefetches and shows comparable performance. Results are very promising, 2.1x - 2.7x speedups.

[2] U. Bondhugula, J. Ramanujam, and P. Sadayappan, "A practical and fully automatic polyhedral program optimization system," 2007.

This older paper was a major advancement for polyhedral compiler technology at the time of its publication. It is refer-

enced by several other papers in this literature review and is included hear for necessary background information.

[3] C. Carruth, "Efficiency with algorithms, performance with data structures," 2014, CppCon 2014. [Online]. Available: https://cppcon2014.sched.com/event/1lYBpr1?iframe=no

Carruth discusses motivations for optimizing software performance in the talk given at CppCon 2014. His discussion focuses on basic, "easy-to-implement" constructs that tre typically missed by C++ programmers. His opening provides significant insight into why software engineers should strive to write fast code.

[4] S. Dissegna, F. Logozzo, and F. Ranzato, "Tracing compilation by abstract interpretation," *SIGPLAN Not.*, vol. 49, no. 1, pp. 47–59, Jan. 2014. [Online]. Available: http://doi.acm.org/10.1145/2578855.2535866

Largely theoretical and abstract. Focuses on optimizing dynamic languages through the usage of a tracing JIT.

[5] Y. Ko, B. Burgstaller, and B. Scholz, "Laminarir: Compile-time queues for structured streams," *SIGPLAN Not.*, vol. 50, no. 6, pp. 121–130, June 2015. [Online]. Available: http://doi.acm.org/10.1145/2813885.2737994

A somewhat specialized optimization, Ko et al. find an effective compile-time transformation for stream programming that results in major performance improvements over typical FIFO structures. The significance of stream programming is important for functional code, particularly non-random-access data traversal.

[6] J. Lifflander and S. Krishnamoorthy, "Cache locality optimization for recursive programs," *SIGPLAN Not.*, vol. 52, no. 6, pp. 1–16, June 2017. [Online]. Available: http://doi.acm.org/10.1145/3140587.3062385

Describes a method for optimizing non-trivial recursive function execution with automatic generation of parallel code. The proposed method involves statically annotating function definitions with desciptions of data access patterns. With these annotations, recursive function execution is transformed into (user-level) threaded code where each thread handles the execution of function call.

[7] S. Mehta and P.-C. Yew, "Improving compiler scalability: Optimizing large programs at small price," *SIGPLAN Not.*, vol. 50, no. 6, pp. 143–152, June 2015. [Online]. Available: http://doi.acm.org/10.1145/2813885.2737954

Proposes alternative to typical compiler dependency model; blocks of statements, rather than individual statements themselves, are viewed in aggregate and modelled as single nodes in a dependency graph. This

[8] V. Paisante, M. Maalej, L. Barbosa, L. Gonnord, and F. M. Quintão Pereira, "Symbolic range analysis of pointers," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16. New York, NY, USA: ACM, 2016, pp. 171–181. [Online]. Available: http://doi.acm.org/10.1145/2854038.2854050

Proposes a method which employs abstract interpretation to conduct global and local alias analysis on pointers. Particularly, the method focuses on deducing whether a subsection of a given pointed-to range aliases a subsection of another. Using symbolic methods, while-program alias analysis can be conducted in linear time.

[9] T. Rompf, A. K. Sujeeth, K. J. Brown, H. Lee, H. Chafi, and K. Olukotun, "Surgical precision jit compilers," *SIGPLAN Not.*, vol. 49, no. 6, pp. 41–52, June 2014. [Online]. Available: http://doi.acm.org/10.1145/2666356.2594316

Addresses the problem of unreliability in modern JIT compilers, like the HotSpot Server compiler (JVM). Proposed is a programmer-visible API to the respective managed language JIT compiler. Lancet, the corresponding implementation of the proposed JIT compiler, makes use of Abstract Interpretation.

[10] K. Stock, M. Kong, T. Grosser, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan, "A framework for enhancing data reuse via associative reordering," *SIGPLAN Not.*, vol. 49, no. 6, pp. 65–76, June 2014. [Online]. Available: http://doi.acm.org/10.1145/2666356.2594342

Proposes optimization techniques for stencil operations (loop -¿ compute -¿ update) with a focus on minimizing usage of register real estate. Leverages associativity and commutativity of addition and multiplication to transform operations in nested loop to develop efficient access patterns that use a minimal number of registers.

[11] K. A. Tran, T. E. Carlson, K. Koukos, M. Sjlander, V. Spiliopoulos, S. Kaxiras, and A. Jimborean, "Clairvoyance: Look-ahead compile-time scheduling," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb 2017, pp. 171–184.

Proposes a method for compile-time reordering of memory-bound instructions to improve performance on non-complex out-of-order CPUs. Method consists of splitting instruction sequences in loop bodies into multiple "Access Phases" and "Execute Phases". Memory-bound instructions are hoisted up into access phases in a safe, dependency-aware manner while following execute phases are simplified to reduce register pressure. Benchmark results show mostly small improvements across different C library recompilations, with a few large improvements and occasional pessimisation.