RETROSPECTIVE:

Improving Register Allocation for Subscripted Variables

David Callahan
Cray, Inc.
411 First Avenue S., Suite 600
Seattle, WA 98104-2860
david@cray.com

Steve Carr
Department of Computer Science
Michigan Technological University
Houghton MI 49931-1295
carr@mtu.edu

Ken Kennedy
Department of Computer Science
Rice University
Houston, TX 77005-1892
ken@rice.edu

1. INTRODUCTION

By the late 1980s, memory system performance and CPU performance had already begun to diverge. This trend made effective use of the register file imperative for excellent performance. Although most compilers at that time allocated scalar variables to registers using graph coloring with marked success [12, 13, 14, 6], allocation of array values to registers only occurred in rare circumstances because standard data-flow analysis techniques could not uncover the available reuse of array memory locations. This deficiency was especially problematic for scientific codes since a majority of the computation involves array references.

Our original paper addressed this problem by presenting an algorithm and experiment for a loop transformation, called *scalar replacement*, that exposed the reuse available in array references in an innermost loop. It also demonstrated experimentally how another loop transformation, called *unroll-and-jam* [2], could expose more opportunities for scalar replacement by moving reuse occurring across an outer loop into the innermost loop. The key contribution of this work was to demonstrate that the benefits of the very successful register allocation strategies developed for RISC processors could be extended to subscripted variables through the use of array dependence analysis. This approach and its descendants have led to substantive, and in some cases dramatic, improvements in the performance of scientific programs on machines with long memory latencies.

In the remainder of this retrospective, we review the major influences that resulted in the development of scalar replacement and unroll-and-jam and influence that our paper had on later work, including commercial compiler implementations.

2. BACKGROUND

In 1987, John Cocke, on behalf of IBM, approached Rice to carry out a research project on register allocation focused on a new RISC machine that would eventually become the RS-6000. He knew that we had a source-to-source program transformation system that had a strong data dependence analyzer built in. He believed that this would be needed to implement unroll-and-jam and similar loop restructuring transformations to improve memory hierarchy performance on the new system. Such strategies were critical because the RS-6000 would have memory latencies as large as 25 cycles, which was staggering in those days.

As a part of this effort, we developed the concept of scalar replacement, mostly as a way of achieving register allocation at the source-to-source level. Most compilers at that time, and certainly the RS-6000 compiler, did a very good job of allocating scalars to registers. Our thinking was that by copying array elements into scalars and then operating on the scalar quantities, we would encourage the compiler to keep the array elements in registers between uses. Dependence analysis was the key to identifying the uses that were candidates for this transformation.

Dependence analysis was developed beginning in the early 1970s to support automatic parallelization and vectorization. The earliest papers referring to the underlying ideas were by Kuck, Muraoka and Chen [19] and Lamport [20]. Kuck, et al. [18, 17, 22, 27], and Allen, et al. [3, 4], formalized dependence and applied it to parallelization and vectorization.

In parallelization and vectorization, dependences restrict the parallelism that can be extracted from a loop. Hence, many loop transformations, such as loop interchange [27, 3, 4], loop skewing [28], node splitting [18] and loop distribution [3, 4], attempt to modify or rearrange dependences so that parallelism can be extracted from a loop nest. In essence, the compiler restructures a loop nest so that some loop within the nest has no dependence between successive iterations, and hence can be parallelized.

Unlike vectorization and parallelization, scalar replacement and unroll-and-jam do not rearrange dependences to allow loop parallelism. Instead these transformations use dependences to identify data reuse. If a data element can be kept in a register between the instruction at the source of the dependence and its sink, a memory access can be avoided. In addition, dependence can be used to improve instruction-level parallelism through the unroll-and-jam transformation. Hence, in this paradigm, dependences represent opportunities rather than constraints.

Perhaps the first to take advantage of the reuse that dependences encapsulate was Abu-Sufah [1], who used dependence to improve the performance of virtual memory. Vector register allocation [4, 5] used dependence to determine when vectors of data were reused and could be kept in vector registers. In this context, scalar replacement is vector register allocation for vectors of size one.

As we indicated earlier, unroll-and-jam was not new. Allen and Cocke [2] defined it in their famous catalog. However, scalar replacement represented a new strategy, albeit one that was frequently employed in hand coding. Our first concept paper, by Callahan, Cocke and Kennedy [7], presented scalar replacement and unroll-and-jam as strategies to reduce pipeline interlock and improve the balance between memory accesses and floating-point computation. The paper in this volume reported on the implementation and experimental validation of these techniques.

20 Years of the ACM/SIGPLAN Conference on Programming Language Design and Implementation (1979-1999): A Selection, 2003.

3. INFLUENCE

Since our paper appeared in SIGPLAN PLDI '90, many extensions have been added to both scalar replacement and unroll-and-jam. In this section, we outline a number of papers that are most closely related and influenced by our work. This list is only a sample of papers and is by no means exhaustive.

Carr and Kennedy [11] extended the original algorithm to apply scalar replacement to loops that contain conditional control statements by using a combination of dependence analysis and partial redundancy elimination [15]. Deusterwald, Gupta and Soffa [16] developed a data-flow analysis framework to apply scalar replacement in loops with control statements.

Wolf and Lam [25] used unroll-and-jam in the context of their data locality optimizations; they referred to this as "register tiling." Carr and Kennedy [10] showed how to compute the unroll factors for unroll-and-jam from the dependence graph with the objective of improving loop balance. Carr [8], Wolf, Maydan and Chen [26], and Carr and Guan [9] combined optimization for instruction-level parallelism and cache using unroll-and-jam and scalar replacement. Sarkar [24] used a local instruction scheduling based model to compute unroll-and-jam amounts. Qian, Carr and Sweany [23] developed a performance model based upon software pipelining that included intercluster register copies for clustered VLIW architectures.

Since reuse of array values in registers can be captured by dependence information, it is only natural to expand the use of dependence to improve performance of the data cache. McKinley, Carr and Tseng [21] use dependence analysis directly to determine the cache behavior of loops and apply loop permutation and loop fusion to improve locality.

Many commercial compilers today incorporate both scalar replacement and unroll-and-jam in some form. We are aware of implementations of these optimizations in compilers for the following architectures: Texas Instruments TMS320C6x, Compaq Alpha, MIPS R10000, and Intel IA-64.

ACKNOWLEDGMENTS

The authors would like to express our sincerest gratitude to the late John Cocke, who inspired and funded this work while at IBM. His accomplishments made him a pioneer not only of computer architecture but also in compiler optimization.

REFERENCES

[1] W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers*. PhD thesis, University of Illinois, 1978.

[2] F. E. Allen and J. Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1972.

[3] J. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Conference Record of the Fourteenth ACM Symposium on the Principles of Programming Languages*, Munich, West Germany, Jan. 1987.

[4] J. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, Oct. 1987.

[5] J. Allen and K. Kennedy. Vector register allocation. *IEEE Transactions on Computers*, 41(10):1290 – 1317, Oct. 1992.

[6] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 275–284, Portland, OR, July 1989.

[7] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5:334–358, 1988.

[8] S. Carr. Combining optimization for cache and instruction-level parallelism. In *Proceedings of the 1996 Conference on Parallel Architectures and Compiler Techniques*, pages 238–247, Boston, MA, Oct. 1996.

[9] S. Carr and Y. Guan. Unroll-and-jam using uniformly generated sets. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30)*, Research Triangle Park, NC, Dec. 1997.

[10] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, 1994.

[11] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. *Software – Practice & Experience*, 24(1):51–77, Jan. 1994.

[12] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages*, 6:45–57, Jan. 1981.

[13] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105, Boston, MA, June 1982.

[14] F. C. Chow and J. L. Hennessy. Register allocation by priority-based coloring. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 222–232, Montreal, Quebec, June 1984.

[15] K.-H. Drechsler and M. P. Stadel. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, Oct. 1988.

[16] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical data flow framework for array reference analysis and its use in optimizations. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 68–77, Albuquerque, NM, June 1993.

[17] D. Kuck, R. Kuhn, B. Leasure, and M. Wolfe. The structure of an advanced retargetable vectorizer. In *Supercomputers: Design and Applications*, pages 163–178. IEEE Computer Society Press, Silver Spring, MD., 1984.

[18] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eight ACM Symposium on the Principles of Programming Languages*, 1981.

[19] D. Kuck, Y. Muraoka, and S. Chen. On the number of operations simultaneously executable in fortran-like programs and their resulting speedup. *IEEE Transactions on Computers*, C-21(12):1293–1310, Dec. 1972.

[20] L. Lamport. The parallel execution of DO-loops. *Communications of the ACM*, 17(2):83–93, 1974.

[21] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, 1996.

[22] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, Dec. 1986.

[23] Y. Qian, S. Carr, and P. Sweany. Optimizing loop performance for clustered vliw architectures. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 271–280, Charlottesville, VA, Sept. 2002.

[24] V. Sarkar. Optimized unrolling of nested loops. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 153–166, Sante Fe, NM, May 2000.

[25] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ontario, June 1991.

[26] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Twenty-Ninth Annual Symposium on Micorarchitecture (MICRO-29)*, Dec. 1996.

[27] M. Wolfe. Advanced loop interchange. In *Proceedings of the 1986 International Conference on Parallel Processing*, Aug. 1986.

[28] M. Wolfe. Loop skewing: The wavefront method revisited. *Journal of Parallel Programming*, 15(4):279–293, Aug. 1986.