Using Compiler Technology to Drive Advanced Microprocessors

Keith D. Cooper

Department of Computer Science
Rice University
Houston, Texas 77251-1892

Abstract

Recent years have seen the introduction of a series of ever faster, ever more complex microprocessors. These advanced microprocessors have found widespread application in machines that range from personal computers to engineering workstations to massively parallel multicomputers. Unfortunately, many of the features used to endow these processors with high peak performance numbers are difficult for either human programmers or compilers to manage. This paper looks at broad trends in microprocessor architecture, relates them back to the basic problems that they present to a compiler, and examines the kind of compiler infrastructure that will be required to address them.

1 Overview

Developments in the design of microprocessors will shape tomorrow's computing systems. Microprocessor-based personal computers and workstations dominate the desktop; today's fastest supercomputers are actually large collections of microprocessors linked together in some regular way. Recent years have seen rapid changes in the design of microprocessors and systems built around them. These developments have shifted a larger share of the burden for achieving a machine's peak performance from the hardware to the software. That is, the programmer and the compiler must take more of the responsibility for achieving high performance on these systems.

In their quest for ever higher peak performance, the architects of modern microprocessors have turned to a collection of features that are difficult for conventional compilers to exploit. A typical advanced architecture microprocessor

- has several pipelined functional units,
- issues several instructions per cycle, and
- has a short cycle time relative to its surrounding RAM chips.

Independently, multiple pipelines, wide instructions, and relatively slow memories pose challenges to the compiler. Taken together, they present a daunting challenge. (Fortunately, not all of the features in modern microprocessors pose problems. These machines have large register sets, uniform addressing modes, and RISC-style integer units, each of which simplifies some aspect of compilation.)

To make interesting systems out of microprocessors, architects embed them inside memory hierarchies. Single level caches are the rule; multiple level caches are beginning to appear. To achieve higher aggregate performance, a group of microprocessors is connected to a private communications network to form a multiprocessor. Memory can be either local to a processor or shared among multiple processors; the extent of sharing varies widely. (One likely side effect of sixty-four bit addressing is that users will expect to be able to address any byte in the machine, even when the memories are *very* large.)

Taken together, these features have twin effects: raising the peak performance of a system and shifting a larger share of the burden for achieving that performance onto the compiler. Unfortunately, users buy systems with an expectation that they will attain a reasonable fraction of peak performance. To an ever increasing extent, it takes a good compiler to achieve this level of performance.

This paper appeared in the Proceedings of the DARPA Software Technology Conference, April 1992, on pages 42–49. It has been reformatted to use a single column on a wider page. The text itself has not changed.

Thus, microprocessor and systems architects have become driving forces behind research and development in code optimization. Modern machines present new challenges to the compiler; at the same time, the relative importance of older problems is changing. While advances in compilation and changes in architecture are difficult to predict, we can pinpoint some of the important issues by looking at broad trends in both areas. This paper looks at several well known trends in the architecture of microprocessor-based systems, relates them back to the basic problems that they will present to the compiler, and examines the kind of compiler infrastructure that will be required to address them.

2 Microprocessor system trends

While the design details vary widely from microprocessor to microprocessor, a number of trends are clear. To increase peak performance, designers have built machines that have an ever larger number of instructions executing concurrently. To supply operands to all those instructions, they have embedded the microprocessors inside multiple layers of cache memory. Finally, to achieve aggregate performance that far exceeds uniprocessor performance, they have built large-scale multiprocessors.

Instruction level parallelism. To achieve higher uniprocessor performance, architects have increased the amount of instruction-level parallelism. This appears in two distinct ways: increasing the use of pipelining and increasing the number of instructions issued in each cycle.

Pipelining can increase the number of concurrently executing instructions; after starting a pipelined instruction, another instruction can be issued on the next cycle, provided that an instruction is ready. If the next instruction needs the result of the current instruction, then that potential is lost. If the processor does not have scoreboarding hardware, the compiler must analyze the code and insert `nop` instructions when overlapped execution is not safe. Reordering can often eliminate these conflicts.

Pipelining instruction execution can also increase demand for registers. In some designs, an instruction that takes three cycles ties down its registers for the duration (others only require the result register during the final cycle). Additionally, widespread use of pipelining increases the amount of state that must be managed.

The ability to issue multiple instructions in each cycle creates two principal problems for the compiler. The compiler must determine when to issue each instruction; modern machines have extremely complex scheduling problems. Successful scheduling can drastically increase the rate of computation; unfortunately, this also increases the rate at which operands must be supplied. If more instructions are issued at each cycle, more operands must be ready, requiring more memory bandwidth.

Deeper memory hierarchies. To meet the demands that an advanced microprocessor places on its memory system, designers use cache memories. This strategy only works if a program exhibits reasonable locality – that is, the program reuses data that is already cached at some level of the hierarchy. Some common operations have nice locality properties; for example, much work has been done on improving the locality of dense linear algebra codes. Others, like a character copy loop in C or a vector addition, have no reuse. Thus, the only improvement from caching comes from faster access to the second and subsequent elements in a cache line.

As the gap between processor and RAM chips widens, multiple-level caches will become more common. Each level of the memory hierarchy has a different latency; scheduling loads and stores to cover latency becomes more complex. To make matters worse, the cost of memory access may actually be dependent on the manner in which it is used. For example, on the i860XP, the double precision floating-point load instruction takes the same number of cycles as the quad-word floating-point load [20]. Thus, accessing adjacent, quad-word aligned items can halve the cost of loading data. Of course, using this feature increases the demand for registers and introduces a pairwise allocation problem.

This also introduces a somewhat subtle but pervasive problem: tracking alignment. Within a single procedure, the compiler can understand the alignment of the various data objects. Thus, on a series of consecutive double-word references, the compiler may be able to use a feature like the quad-word load mentioned earlier. If the data object is a call-by-reference formal parameter, however, the compiler cannot determine whether or not the first element begins on the quad-word boundary.

Multiprocessor parallelism. From the design, programming, and performance perspectives, the most complex microprocessor-based computers are multiprocessors. Machines like the Intel Paragon machines, the Thinking Machines CM-5, and the Alliant Campus all rely on large-scale multiprocessor parallelism to provide levels of aggregate performance that exceed those available with more traditional vector supercomputers. Using these machines to their potential is difficult; several major tasks confront the user and compiler.

finding and expressing parallelism: The program must contain enough parallelism to make use of all of the distinct processors that the machine presents. Furthermore, the parallelism must be exposed in a way that the compiler recognizes.

granularity: The individual parallel tasks must contain enough work to offset the overhead costs of creating and synchronizing them. If this is not true, the parallel code will run more slowly than a sequential version, because more time is spent getting in and out of the parallel region than would be required to perform the computation on a single processor.

data placement and communication: Once the parallelism is recognized and understood, the compiler must assign each task to a processor, place each data item on one or more processors, and orchestrate the interprocessor communication to ensure that data is available when and where it is needed. Changing data placement and communication schemes can radically change the performance obtained on a multiprocessor machine.

For these reasons, along with others, writing a program that makes effective use of a large-scale parallel machine is a difficult engineering task.

Because the individual CPUs in such a system are all microprocessors, each of them has the problems detailed earlier. Thus, these machines will have many distinct memory hierarchies. If there is sharing between the address spaces of the individual processors, another ugly problem arises: maintaining consistency between caches on different processors.

3 Compiler-based solutions

Fortunately, researchers in compilation are as active as those in architecture. As microprocessor-based systems have evolved, so has the focus of work in compiling. This section sketches some of the tools being developed to address the problems described in Section 2.

Instruction level parallelism. Processors like the i860 and the RS/6000 achieve much of their performance through pipelining and issuing several instructions in a single cycle. This basic idea has been around for a long time; early high-performance machines like the IBM 360/91 and the CDC 6600 had facilities for overlapping execution of instructions. Thus, a long and deep literature has developed on compiler-based techniques to capitalize on features like pipelining and superscalar instruction issue. In general, these techniques are referred to as *instruction scheduling*. This work was quickly adopted into compilers for microprocessors [15, 19]. Most of this work dealt with ordering the instructions within a single basic block.

Recently, research in this area has focused on aggressive techniques that reorder entire loops or loop nests. These techniques, like Fisher's trace scheduling [13], Aiken and Nicolau's perfect pipelining [1], and Lam's software pipelining [22], create an order for the instructions in which pieces of several different iterations of the loop may be executing concurrently. Work continues on this set of problems at a furious pace; much of

that work aims to extend the earlier techniques to handle loops that contain conditional control flow and to deal with real constraints on resources like instruction issue slots and machine registers.

Deeper memory hierarchies. In many microprocessor-based systems, managing the memory hierarchy is as important as properly scheduling instructions. For example, on the iPSC/860 – a hypercube-connected multiprocessor based on the i860XR – changing DRAM pages[1] incurs a ten-cycle stall [26]. In many real loops, stalls from the memory controller will dominate the cost of the actual computation. Thus, there is broad interest in techniques for improving the locality of programs.

Like scheduling, this is not a new problem. In the earliest days of compiling, researchers worked on techniques to pack memory in order to run practical problems on machines with tiny memories [23]. With the advent of virtual memory, the focus shifted to improving page locality – that is, decreasing the number of page faults during a program's execution. Work in operating systems concentrated on statistical measures of program behavior, in large part because the operating system cannot look inside a program to ascertain its possible future behavior. In the compiler construction community, work has looked at packing memory for locality [12, 27] and transforming loops to improve their cache behavior [14, 29].

The tools to improve locality exist. Open areas for future work include improved decision algorithms to drive the transformation process, techniques that consider multiple levels of cache, and mechanisms to parameterize the entire process around a few simple characteristics of the cache.

Multiprocessor parallelism. Multiprocessor systems present their own set of challenges to the compiler. A large and active research community is attacking the problem of compiling code that makes effective use of these machines. Over the past decade, progress has been made in detecting parallelism and in improving performance on parallel programs [2, 3, 21].

The key tool that compilers use to improve performance on multiprocessor systems is static program analysis. Restructuring compilers perform data-flow analysis, data-dependence analysis, and control-dependence analysis to understand the sequence of memory references and computation that can occur during a program's execution [16, 24]. These compilers examine the program, on a loop nest by loop nest basis, trying to discover loop nests whose iterations can be run in parallel. If the compiler can prove that parallel execution does not reorder the memory references and computations in a way that changes the results, then it transforms the loop into a parallel loop. Of course, not all loop nests are provably parallel. Thus, restructuring compilers have a repertoire of transformations that they use in an attempt to reformulate loops into parallel form.

Of course, just finding parallelism is not enough. The compiler must worry about the amount of work inside a parallel region, about the placement of data, and about the structure and efficiency of compiler-inserted communication and synchronization. All of these issues are the subject of on-going research in the area.

4 Meeting the challenge

Crafting a compiler that can make effective use of a modern microprocessor-based system is quite a challenge. The compiler must deal with all of the problems described in Section 2. Additionally, it must do an outstanding job of classical optimization. Finally, it must be reasonably efficient itself.

The common thread to all of the techniques mentioned in Section 3 is compile-time knowledge about program behavior. If we are to build compilers that do a substantially better job of driving microprocessor-based systems, we must design into them the deepest analysis that is practical. The compiler must break

[1] A DRAM page on the iPSC/860 is 4k bytes.

down the barriers to analysis that are imposed by artificial boundaries in the code, like procedures and compilation units. It must analyze the entire program to derive information that is as precise as possible. It must use the results of this analysis to transform the program into a form where it is more amenable to efficient execution on the target system.

Interprocedural analysis. Algorithms for performing data-flow analysis across whole programs are both readily available and well understood [5, 7, 8, 10]. These techniques are beginning to appear in commercial systems [25]. Perhaps less well appreciated are the limitations of this kind of analysis, the role that it can play in code optimization, and the rational consequences of that role.

Interprocedural data-flow analysis is a technique for estimating, at compile time, sets of facts that will hold at run-time. In particular, an interprocedural analyzer estimates the set of variables that may be modified or referenced by a call to some procedure, the set of variables that have known constant values on entry to each procedure, and the set of variables that may occupy overlapping storage on entry to a procedure. The analysis is approximate; it assumes that all paths through the code can actually execute. A second source of approximation comes from the treatment of arrays as single units – that is, if any element of an array is modified, the entire array is deemed to have been modified.

Unfortunately, experience with interprocedural analyzers suggests that more precise information is needed to support dependence analysis [18]. The precision can come in two ways: deeper forms of analysis, and transformations on the program. To perform deeper analysis, we must reformulate the basic problems that are solved. As an example, Havlak has implemented a more precise form of side-effect analysis called *regular section analysis* [18]. It uses a simple lattice formulation to track more complex information about reference patterns within an array.

A more subtle form of imprecision arises because the compiler analyzes the program as written. The compiler computes, for each procedure, the strongest set of facts that it can prove for *all* the paths that reach that point. If we clone a copy of some procedure and redistribute the calls to it, we can eliminate the dilution that happens when two paths with different information meet. By modifying the program, we can create a "nearby" program that produces the same answers but is more amenable to analysis and optimization.

Interprocedural optimization. The compiler can apply interprocedural transformations for two very different reasons: to directly improve the code's performance and to change the program's structure in a way that enables some other transformation. The former class of optimizations are relatively straightforward. They include inline substitution, cross-procedural register allocation [9, 28], and limited forms of interprocedural code motion [17].

The latter situation is more complex; the difficulty here is deciding when and where to apply an optimization. Inline substitution and procedure cloning can fall in this category. Procedure cloning is an unusual case. It replaces a single copy of the procedure with two or more copies and assigns some subset of the calls to each of them. It has two principal effects. First, it can expose more interprocedural facts by segregating paths through the call graph that contribute significantly different information. Second, because it singles out a path through the call graph, cloning can create a point where specialized code can be placed. In effect, it creates a different path through the code for the special cases that can be effectively optimized. For example, cloning based on parameter alignment can create an opportunity for using the i860XP's quad-word load instruction.

We have been exploring the problem of deciding how to use inlining and cloning to create opportunities for other transformations. Our study of inline substitution (using commercial FORTRAN compilers) showed that secondary effects in the compilers often overshadowed *any* benefit from inlining [11]. Thus, our strategy is to use interprocedural transformations in a *goal-directed* way – that is, we identify a high-payoff transformation that can be helped by some combination of procedure cloning and inlining and use them to enable the high-

payoff transformation [4]. We have used this goal-directed strategy in several experiments; we have been pleased with the results.

Special case code generation. A natural extension of our work with procedure cloning is to be more aggressive about generating conditional code. In our inlining study, both the vectorizing compilers had cases where they incorrectly assumed that parallel execution of a loop was profitable [11]. The compilers should have inserted a run-time test on the number of iterations and generated both a sequential and a parallel version of the loop. This would have led to better behavior on small data sets while keeping the parallelism for the larger cases where the granularity actually covered the overhead of startup and synchronization.

When the compiler cannot determine the value of some important constant at compile time, it should consider generating a run-time check and special case code. There is a problem with this strategy; duplicating code increases object code size. The tradeoff is simple: code space against performance consistency.

Program repositories. Of course, implementing all of these ideas in a production compiler requires careful attention to efficiency. In particular, a less-than-careful implementation of interprocedural analysis and interprocedural optimization can examine every procedure in the program to compile just one. A compiler that uses interprocedural techniques needs a *program repository* to store the results of analysis and optimization.

The repository will contain information about individual procedures and about whole programs. For example, the set of variables modified directly in a procedure p is independent of the rest of the program while the set of variables modified by a call inside p must be a function of the program in which p is included. Object code for p can be stored with the procedure if no interprocedural information was used to produce it. On the other hand, if it is the product of interprocedural facts derived from a program that includes p, then it must be stored with that program.

Use of a repository makes interprocedural analysis and optimization more practical. In particular, it can reduce the amount of local analysis required – p is only re-analyzed when it has been edited rather than on every compilation. By retaining information across compilations, the compiler can also limit the amount of recompilation that must be performed in response to an editing change to one procedure [6].

5 Conclusions

Microprocessor-based systems present many challenges to the compiler writer. Given the proliferation of systems designed around microprocessors, it is imperative that we continue to attack the problems that we already see and that we examine the trends in microprocessor design to try and predict future developments. In particular, there are a number of concrete steps that we can take to improve our ability to compile effective code for these machines:

- closer cooperation between compiler writers and microprocessor architects
- build better infrastructure to provide an information rich environment for the compiler
- look for linguistic solutions to hard problems, like a mechanism for expressing locality

Advanced architecture microprocessors are here to stay. Careful application of compilation technology appears to be the only way to narrow the gap between the architect's dream and the user's experience.

References

[1] A. Aiken and A. Nicolau. Perfect pipelining: A new loop parallelization technique. In *Proceedings of the 1988 European Symposium on Programming*. Springer Verlag Lecture Notes in Computer Science, Mar. 1988.

[2] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, June 1987.

[3] J. R. Allen and K. Kennedy. PFC: A program to convert Fortran to parallel form. In K. Hwang, editor, *Supercomputers: Design and Applications*, pages 186–203. IEEE Computer Society Press, Silver Spring, MD, 1984.

[4] P. Briggs, K. D. Cooper, M. W. Hall, and L. Torczon. Goal-directed interprocedural optimization. Technical report, Rice University, Nov. 1990.

[5] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*. ACM, June 1986.

[6] M. Burke and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. To appear in *ACM Transactions on Programming Languages and Systems*.

[7] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 23(7), pages 47–56. ACM, July 1988.

[8] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction,* SIGPLAN Notices 21(7), pages 152–161, July 1986.

[9] F. C. Chow. Minimizing register use penalty at procedure call. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 23(7), pages 85–94, July 1988.

[10] K. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 49–59. ACM, Jan. 1989.

[11] K. D. Cooper, M. W. Hall, and L. Torczon. An experiment with inline substitution. *Software – Practice and Experience*, June 1991.

[12] J. Fabri. Automatic storage optimization. In *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction,* SIGPLAN Notices 14(8), pages 83–91, Aug. 1979.

[13] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: A smart compiler and a dumb machine. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction,* SIGPLAN Notices 19(6), pages 37–47, June 1984.

[14] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[15] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction,* SIGPLAN Notices 21(7), pages 11–16, July 1986.

[16] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.

[17] M. W. Hall, K. Kennedy, and K. McKinley. Interprocedural transformation for parallel code generation. In *Proceedings of Supercomputing '91*, Albuquerque, NM, Nov. 1991.

[18] P. Havlak and K. Kennedy. Experience with interprocedural analysis of array side effects. In *Proceedings of Supercomputing 90*, pages 952–961. IEEE Computer Society Press, Nov. 1990.

[19] J. L. Hennessy and T. Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[20] Intel Corporation. *i860™ XP Microprocessor*, 1991.

[21] D. Kuck, R. Kuhn, B. Leasure, and M. J. Wolfe. Analysis and transformation of programs for parallel computation. In *Proceedings of COMPSAC 80, the 4th International Computer Software and Applications Conference*, pages 709–715, Chicago, IL, Oct. 1980.

[22] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 23(7), pages 318–328, July 1988.

[23] S. S. Lavrov. Store economy in closed operator schemes. *Journal of Computational Mathematics and Mathematical Physics*, 1(4):687–701, 1961. English translation in *U.S.S.R. Computational Mathematics and Mathematical Physics* 3:810-828, 1962.

[24] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 26(6), pages 1–14, June 1991.

[25] R. Metzger and P. Smith. The CONVEX application compiler. *Fortran Journal*, 3(1):8–10, 1991.

[26] D. S. Scott and G. R. Withers. Performance and assembly language programming of the iPSC/860 system. In *The Sixth Distributed Memory Computer Conference Proceedings,* Portland, OR, pages 534–541, Apr. 1991.

[27] K. O. Thabit. *Cache Management by the Compiler.* PhD thesis, Rice University, May 1982.

[28] D. Wall. Register windows vs. register allocation. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 23(7), pages 67–78. ACM, July 1988.

[29] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 26(6), pages 30–44, June 1991.