Investigating Adaptive Compilation using the MIPSpro Compiler

Keith D. Cooper and Todd Waterman

Department of Computer Science
Rice University
Houston, Texas, USA

Abstract

Despite the astonishing increases in processor performance over the last forty years, delivered application performance remains a critical issue for many important problems. Compilers play a critical role in determining that performance. A modern optimizing compiler contains many transformations that attempt to increase application performance. However, the best combination of transformations is an application-specific issue [4, 5]. This paper details an experiment in which we used techniques from our work on adaptive compilation in an attempt to duplicate the application and processor specific optimization attained by the ATLAS system [6]. Our goal was to show that an adaptive compiler can achieve results similar to those achieved by ATLAS by using more computer time and less human effort.

Command-line flags provide the developer with some control over a compiler's optimization process. We selected the MIPSpro compiler on an R10000 processor and used an adaptive feedback system to find the best blocking factors for the matrix-matrix multiply code in Linpack and ATLAS, DGEMM. (We found that the other command line flags either had little impact on DGEMM's performance or had an obvious best setting.) Our experiments show that the adaptive approach: derives good blocking factors for DGEMM at various input sizes; picks better blocking factors, in general, than the MIPSpro compiler; and achieves performance that is competitive with ATLAS (for much less human effort). By automating this kind of tuning, adaptive compilation has the potential to make the custom-tuned performance of ATLAS available for a larger class of applications.

1 Introduction

Computer performance has increased astonishingly over the last 40 years. However, there are still applications that require cutting-edge performance to run effectively or cannot yet be run in reasonable amounts of time. Complex scientific codes dealing with the weather, earthquakes, and nuclear physics push the bounds of computing power and will continue to do so for years to come. Strong compilers are necessary to maximize the performance of these applications on modern processors.

Efficiently compiling a program into a high-quality executable is difficult. Many of the problems that must be solved during compilation have optimal solutions that are NP-complete. During optimization, a compiler makes many decisions that affect the performance of the resulting code. The best choices can depend on many factors including the specific target machine, the source program, and the input data. The best sequence of decisions can usually not be deduced during a single compile; in the worst case, they may not be known until all possible sequences are tried. Because this kind of exhaustive exploration is impractical today, compilers typically use a single preset sequence. They use heuristics or simple estimators to determine reasonable parameters for transformations that require them.

An adaptive compiler changes its behavior in

response to the target machine, the source program, and (perhaps) the input data [4]. Our current prototype systems explore the space of decisions experimentally; they compiler the program and evaluate its performance using either execution or estimation. Using a process of feedback-driven iterative refinement, they discover an effective set of options for that combination of input program, target machine, and input data—options that achieve a high level of performance.

This paper focuses on loop blocking for the MIPS R10000 processor. It harnesses the MIPSpro compiler into an adaptive system that chooses blocking sizes. Two critical factors led us to choose this combination. First, the MIPSpro compiler produces, in general, high-quality code for the R10000. Second, the MIPSpro compiler provides command-line flags that allow the user to control blocking sizes; that capability is critical to our experiment. (Again, other compilers were lacking in this regard.) The combination of a strong base compiler and command-line flags that let the adaptive system control blocking make the MIPSpro compiler a good choice for this study. We used this setup to examine the impact of user-level options on the effectiveness of the MIPSpro compiler. We evaluated performance using linear algebra kernels and compared the results against kernels optimized using the ATLAS system.

Our experiments show that the baseline MIPSpro compiler does a good job competing with the ATLAS system for smaller data sets. (Several other compilers that we examined could not.) As array sizes grow, the effectiveness of the standard compiler's loop blocking deteriorates. Adaptive choice of blocking sizes with the MIPSpro compiler produces results that are close to those of ATLAS even for larger data sets. These results are achieved through the variation of a single option to the compiler – blocking size. Adaptive compilation quickly determines a good blocking size while only exploring a small portion of all reasonable values.

Improving the performance of linear algebra kernels to near ATLAS levels with adaptive compilation is a good start. However, our goal for adaptive compilation is ambitious; to obtain these types of results on a wide variety of input programs. Current industrial compilers, including the MIPSpro, are not sufficient for this; the only parameter to MIPSpro that we found useful is blocking size. For the remaining parameters, the adaptive system could not improve on the compiler's default settings. This suggests that compiler writers must parameterize their systems in new and interesting ways to take advantage of adaptive techniques like those that we propose. In particular, the adaptive system will need parameters that exercise fine-grained control over optimization and impact performance in orthogonal ways.

In this paper, we will show the experiments we performed using the MIPSpro compiler and ATLAS. Section 3 explores the potential of varying optimization decisions using a single target code. The data we gathered drove development of an adaptive technique discussed in section 4. In section 5, we hypothesize about how compiler design must change to generalize the experiment of the previous sections, and in section 6 we discuss potential future work. To begin, we will discuss the related work in the area to ground our research.

2 Related Work

A beginning motivation for our work is the ATLAS system [6]. ATLAS tries to achieve hand-coded performance for linear algebra kernels on different processors without a programmer having to modify the code for each processor. Each kernel is modified and parameterized once for all processors by a programmer. Then, when the system is installed on a particular machine, experiments are run to determine the proper parameters for the kernel. The performance of the ATLAS kernels may sometimes fall short of hand-optimized versions that take advantage of special features on a specific processor. However, ATLAS will often outperform hand-coded kernels since they are rarely written for each possible processor configuration.

Hand-coded programs achieve excellent performance at the expense of a large amount of human time to optimize the program. ATLAS trades

off some of this human time for processor time and still delivers high-quality executables. Our approach attempts to take this tradeoff a step further. ATLAS still requires a developer to examine and optimize each kernel included in the system. Our system uses adaptive compilation to replace both the processor-independent hand tuning and the processor-dependent automatic tuning done in ATLAS. The ATLAS-optimized kernels provide a good comparison point for our work and help us to determine if we can use additional processor time to replace hand tuning without significantly harming performance.

Yotov *et al.* modify the ATLAS system to determine proper parameters using a model-driven approach instead of running experiments [8]. This substantially reduces the CPU time necessary to install ATLAS on a target architecture and provided comparable performance on two of the three machines tested. It does not decrease the amount of hand tuning required for each kernel.

Knijnenburg *et al.* investigate automatic choice of blocking sizes [5]. They examine the interaction of blocking size and unrolling factor to find an ideal combination, using an adaptive compilation scheme Their work uses source to source transformations as opposed to adjusting compiler parameters. We expand upon their work by examining how iteratively determining blocking size compares with the ATLAS system and the default algorithms in the MIPSpro compiler, and by placing it in the context of general adaptive compilation. In our experiments, adaptive selection of unrolling factors in conjunction with blocking size never improved upon the results achieved by letting MIPSpro select the unrolling factor automatically.

Profile-driven optimization also executes code on sample input to gather information and improve performance. However, it uses that knowledge in a very different way than do our adaptive compilers. Profiling instruments code and then executes it in order to provide a more detailed picture of how control flows through the program [2, 3]. This information is then used to provide more accurate analysis to the optimizer. In contrast, adaptive compilation uses the results of execution as feedback for choosing parameters and optimizations in subsequent compilations. Profiling usually executes only a single version of the program to provide information for optimization, while adaptive compilation repeatedly optimizes and executes code to iteratively enhance performance. Profiling uses the results of execution to identify control-flow patterns in the executing code. Adaptive compilation uses the results of execution to measure the effectiveness of particular optimization strategies and parameters.

3 Adjusting Blocking Size

Proving adaptive compilation profitable for the MIPSpro compiler requires demonstrating that adaptively chosen compiler parameters can yield better executable performance than the standard compiler. We evaluated the compiler's adaptability by examining the performance impact of various options on kernels from the basic linear algebra subprograms (BLAS). Since these kernels are scientific programs we focused on parameters that affected high-level loop transformations. Initial experimentation showed that several parameters impacted performance, but only one, blocking size, resulted in better performance than the default compiler when varied. Therefore, we further examined the effects of adjusting blocking size.

Blocking is a memory hierarchy optimization that reorders array accesses to improve temporal and spatial reuse [1, 7]. Blocking can be targeted at any level of the memory hierarchy, but is most frequently targeted at reducing cache misses. This reduction of cache misses allows blocking to vastly reduce the running time of scientific codes. The MIPSpro compiler automatically performs blocking when the loop nest optimizer is invoked. The compiler allows the user to manually specify the blocking size, overriding its own choice.

To determine the potential benefits for adaptively determining blocking size we explored the effects of various block sizes on running time. We exhaustively examined the performance of the

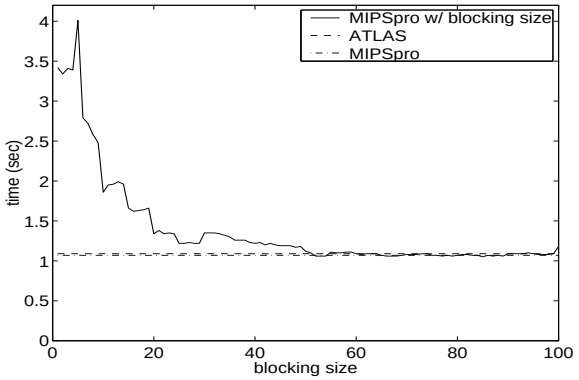

Figure 1: Running times for 500^2 arrays

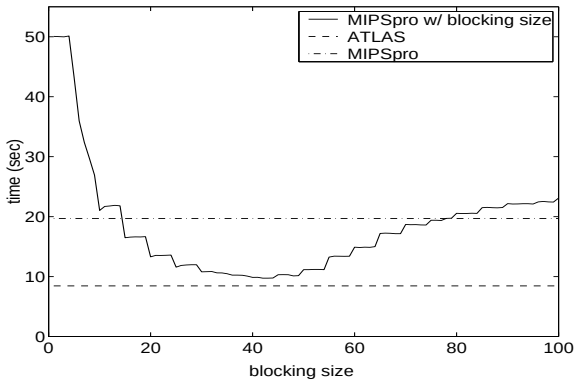

Figure 2: Running times for 1000^2 arrays

BLAS kernel DGEMM, a general matrix multiply routine, for different block sizes and varying array sizes. We first modified the DGEMM kernel by hand to eliminate a single conditional that resided within the multiplication loop nest. The conditional prevented multiplication of rows of the matrix by zero elements. The check was unnecessary for correctness and hurt performance on dense matrices by preventing the compiler from performing any high-level optimizations on the loop nest. Experimentation suggests that this conditional was also removed from the ATLAS DGEMM kernel.

The MIPSpro compiler was run using blocking sizes from one to one hundred squared. We compared the results against the automatically-blocked code as well as a version of DGEMM tuned by the ATLAS system. All tests were executed on a 195 MHz MIPS R10000 with 256MB memory, a 32 KB L1 data cache, and a 1 MB unified L2 cache.

Figures 1, 2, and 3 show the running times of the different methods for various square array sizes. The results for the ATLAS system and the MIPSpro compiler using internally determined blocking are straight lines since they do not vary the blocking size. The results of compiling using the MIPSpro compiler and manually selecting a blocking size vary substantially. Poorly selecting a blocking size can cause substantially worse performance. In addition, as array size increases the range of good blocking sizes decreases and the penalty for selecting a bad blocking size is

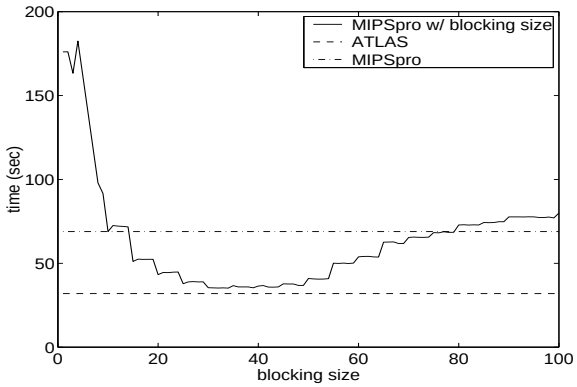

Figure 3: Running times for 1500^2 arrays

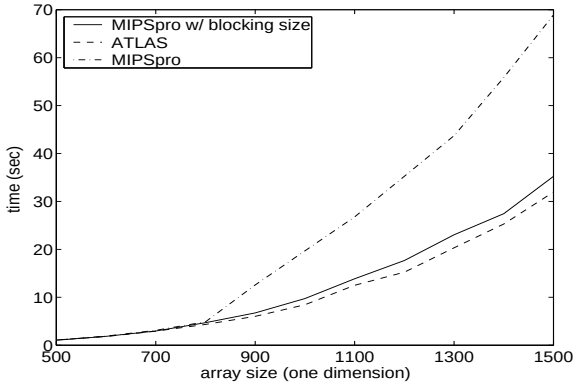

Figure 4: Running times for square matrices

magnified.

The data from these figures also suggests that the performance of the MIPSpro compiler with tuned blocking size remains close to the performance of the ATLAS system as array size increases, while performance of the standard MIPSpro compiler deteriorates. This trend is better shown in figure 4. The performance of the standard MIPSpro compiler diverges from the performance of ATLAS and the MIPSpro compiler with adaptively tuned blocking after the array size reaches 800 squared. This can also be clearly seen in figure 5, which shows the relative running times with respect to the ATLAS system. After the array size reaches 900 squared, compiling using the standard MIPSpro compiler blocking scheme results in more than twice the running time obtained using ATLAS. In contrast, tuning to find an ideal blocking size keeps results within 10 to 15 percent of ATLAS.

The reason behind the poor performance of the standard MIPSpro scheme quickly becomes evident when examining figures 6 and 7. When DGEMM is compiled with the standard MIPSpro compiler it incurs significantly more L1 cache misses than with either ATLAS or a tuned blocking size. The standard MIPSpro compiler uses a rectangular blocking scheme that grows as array size increases. When the array size becomes too large the block size becomes too large for the L1 cache, and there is a drastic increase in the number of cache misses. Cache misses, however, do not explain the performance difference between the MIPSpro compiler with tuned blocking and ATLAS. Tuned blocking has fewer L1 and L2 cache misses than ATLAS, but has worse performance. This difference is probably due to special tuning performed by ATLAS specific to the DGEMM kernel.

We also examined the performance of DGEMM on non-square matrices. DGEMM multiplies an M by K matrix by a K by N matrix to yield an M by N matrix. In our previous experiments M, N and K were always equal, but we also tried holding two of the three dimensions to a value of 1000 and varying the third dimension from 500 to 2000. The results of these additional experiments can be seen in

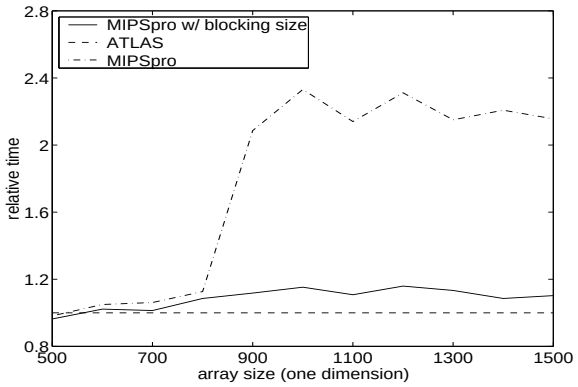

Figure 5: Relative running times for square matrices

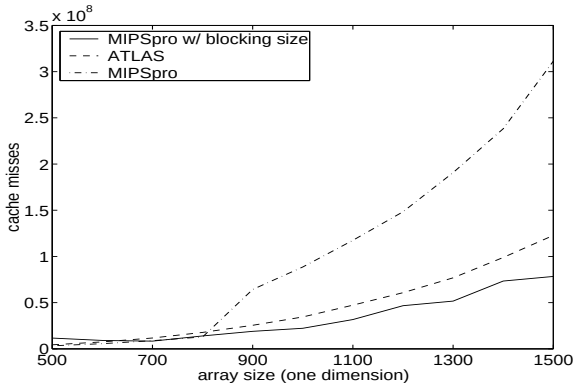

Figure 6: L1 cache misses for square matrices

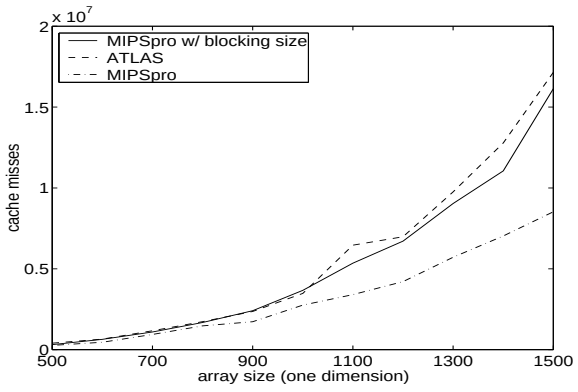

Figure 7: L2 cache misses for square matrices

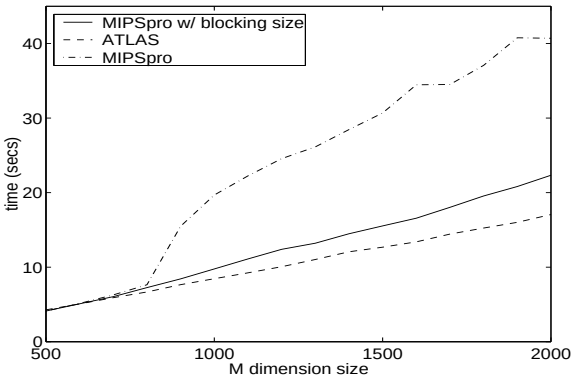

Figure 8: Running times for varying M

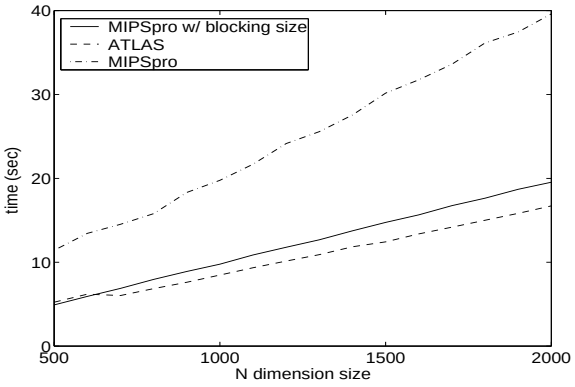

Figure 9: Running times for varying N

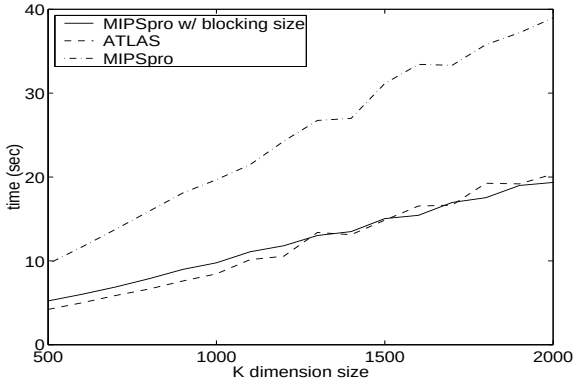

Figure 10: Running times for varying K

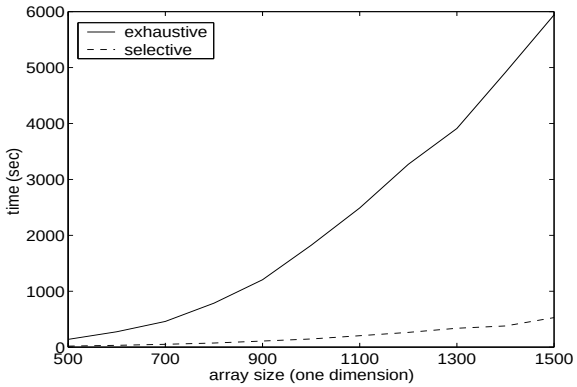

Figure 11: Search time required

figures 8, 9, and 10. There are some minor differences from the results of the square matrices; ATLAS performs worse than expected when the K dimension is larger. The general trend, however, remains the same. The DGEMM executables produced by the MIPSpro compiler are substantially slower than their ATLAS counterparts, while adaptively tuning the blocking size allows executable performance to remain close to ATLAS levels.

4 Determining Blocking Size

Demonstrating the promise of adaptive compilation requires showing increased program performance resulting from adaptively tuning parameters. The previous section shows that correctly setting the parameter for blocking size can lead to better running times for the DGEMM kernel. However, it does not discuss how the blocking size that produces these results can be determined. This section examines how to quickly determine the appropriate blocking size for a program.

Examining every potential blocking size, as done in the previous section, is rarely necessary to find the size that produces the best results. Instead, the space of potential blocking sizes can be explored intelligently. Examining only a few blocking sizes and using those results to choose future sizes to examine allows the best blocking size to be determined in a fraction of the time exhaustive searching would require.

Our approach begins by finding the result for a blocking size of 50. It then begins sampling higher and lower block sizes in increments of ten

as long as the results stay within 10 percent of the best results seen so far. After this stage is complete the area around the best result found is examined in detail: the five block sizes larger than the best result, and the five block sizes smaller than the best result are all examined. Of these eleven block sizes, the one with the best running time is chosen.

When this algorithm was tested on the DGEMM benchmark it always resulted in selecting the best possible blocking size. The amount of time required to determine the blocking size compared to an exhaustive approach can be seen in figure 11. A more aggressive approach could determine a good blocking size in even less time, trading less compile time for slightly more execution time. However, this is contrary to the adaptive compilation approach which believes in using additional compile time to improve executable performance.

Expending CPU time to determine the correct blocking size parameter may also not be necessary for each program compiled. Since the performance of blocking is based on the dimensions of the arrays and not the values contained within, the ideal blocking size needs to be determined only once for each set of dimensions. The first time a program is compiled with a specific set of dimensions the correct blocking size can be determined and stored in a table. Whenever subsequent programs are compiled with the same array dimensions the blocking size parameter can be retrieved from the table.

5 Making Adaptive Compilation General

The results presented in the previous sections show some of the promise of adaptive compilation. However, for adaptive compilation to be truly successful it must improve performance for a wide variety of codes. Our current research only shows how adaptive compilation can find good blocking sizes to improve the performance of linear algebra kernels. Finding the correct blocking size is critical for performance in this situation, but is of little importance for many codes. Achieving our goals for adaptive compilation will require many changes to the workings of current compilers.

Adaptive compilation improves program performance by tuning the behavior of the optimizer. The adaptive system can only control those aspects of optimization that the compiler exposes. Therefore, the effectiveness of adaptive compilation depends directly on the parameterization of the compiler. Our research compiler has been built in a way that lets the adaptive compiler control the ordering of passes, as well as some parameters to those passes [4]. The MIPSpro compiler lets the adaptive compiler control blocking size, along with other parameters that had little effect in the experiments we describe in this paper. Changing the way compilers expose optimization decisions and parameters is a critical issue in the design of effective adaptive compilers.

To make adaptive compilation effective, the underlying compilers must expose a variety of parameters that target distinct aspects of optimization and performance. The interface between adaptive system and compiler must allow for more complex information. The adaptive system must have control over what transformations are applied. In addition, it may need some control over notions of granularity, scope of optimization, and specific parameter settings. The MIPSpro experiments were possible because the compiler allowed command-line control of the blocking size, rather than just allowing the user to enable or disable blocking.

Discovering the appropriate parameterizations is one focus of our work on adaptive compilation. While we do not yet know the answers, we can identify some kinds of controls that should be explored. In blocking, for example, the adaptive system might specify different blocking sizes for individual arrays or individual loop nests. With inline substitution, the adaptive system might specify specific call sites that should (or should not) be inlined. With redundancy elimination, the adaptive system might specify threshold levels of register pressure that locally disable replacement. With instruction scheduling, the adaptive system might select priority heuristics

and scheduling disciplines.

This paper demonstrates the effectiveness of command-line control of a single optimization. As our program of research on adaptive compilation progresses, we will learn more about the effectiveness of different parameterization schemes. These, in turn, will affect the way that we design and implement optimizations.

6 Future Work

The previous section described some of the work that we anticipate pursuing in our research program on adaptive compilation. Future research concerning adaptive computation in general was discussed in the previous section, but future We also intend to continue working with the MIPSpro compiler in an attempt to achieve performance even closer to the ATLAS optimized kernels. We expect that these experiments will also provide us with knowledge that will help to advance our more general goals for adaptive compilation.

The experiments in this paper are based on the DGEMM kernel of the BLAS library. This kernel was a good starting point since it is a linear algebra routine that uses dense matrices where blocking is important and results can be compared to the DGEMM kernel optimized by the ATLAS system. Obviously, manipulating the blocking size will have little effect on non-scientific codes, but it is worth examining how adaptively adjusting the blocking size would effect a wide variety of scientific codes. This would also allow us to see how ideal blocking sizes vary between benchmarks.

The MIPSpro compiler claims to allow manual blocking at two different levels. This allows the compiler to individually optimize blocking for both the first and second level caches. However, in experimenting with the compiler it was unable to manually block for a second level. Adaptively determining two levels of blocking is an important future experiment. It should result in better performance for the adaptive approach, but will also require more advanced search algorithms to deal with the significantly larger search space. These algorithms might prove applicable to adaptive compilation beyond just determining blocking sizes.

7 Conclusion

Our investigation of adaptive compilation using the MIPSpro compiler shows the potential benefits of adaptively tuning parameters. Experiments revealed that adaptively selecting the appropriate blocking size for the DGEMM kernel provides performance near the level of the ATLAS system. In comparison, the performance of the standard compiler drops off considerably for larger array sizes. A good blocking size can also be found quickly by intelligently exploring only a small portion of the search space once for each array size.

The MIPSpro compiler, however, is not sufficient to make adaptive compilation a generally applicable technique. Many heuristic decisions are made during optimization, but these decisions cannot be affected by the parameters of current compilers. The success of adaptive compilation on a wide range of applications will require the design of compilers that expose a carefully selected set of parameters that can significantly alter performance in different ways.

Acknowledgements

This work was supported with funds from the Los Alamos Computer Science Institute and the National Science Foundation ITR Program. Anshuman Dasgupta, Jack Dongarra, Alex Grosul, Tim Harvey, Ken Kennedy, Steve Reeves, Devika Subramanian, and Linda Torczon have all contributed to this work through extended discussion of the ideas and experiments. To all these people go our sincere thanks.

References

[1] Walid Abu-Sufah. *Improving the Performance of Virtual Memory Computers*. PhD thesis, University of Illinois at Urbana-Champaign, 1979.

[2] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *Proceedings of the Nineteenth Annual ACM Simposium on Principles of Programming Languages*, pages 59–70, January 1992.

[3] Pohua P. Chang, Scott A. Mahlke, and Wen mei W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21(12):1301–1321, December 1991.

[4] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21^{st} century. In *Proceedings of the 2001 LACSI Symposium*. Los Alamos Computer Science Institute, Santa Fe, NM, October 2001.

[5] Peter M.W. Knijnenburg, Toru Kisuki, and Michael F.P. O'Boyle. Iterative compilation. In *Embedded Processor Design Challenges*, volume 2268, pages 171–187, 2002.

[6] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1–2):3–25, 2001.

[7] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, June 1991.

[8] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulkis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A comparison of empirical and model-driven optimization. In Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation, June 2003.