International Journal of Parallel Programming, Vol. 33, Nos. 2/3, June 2005 (© 2005)
DOI: 10.1007/s10766-005-3581-7

Empirical Optimization for a Sparse Linear Solver: A Case Study

Yoon-Ju Lee,[1] Pedro C. Diniz,[1] Mary W. Hall,[1] and Robert Lucas[1]

This paper describes initial experiences with semi-automated performance tuning of a sparse linear solver in LS-DYNA, a large, widely used engineering application. Through a collection of tools supporting empirical optimization, we alleviate the burden of performance tuning for mapping today's sophisticated engineering software to increasingly complex hardware platforms. We describe a tool that automatically isolates code segments to create benchmark subsets for the purposes of performance tuning. We present a collection of automatically generated empirical results that demonstrate the sensitivity of the application's performance to optimization parameters. Through this case study, we demonstrate the importance of developing automatic performance tuning support for performance-sensitive applications.

KEY WORDS: Memory hierarchy optimization; performance tuning.

1. INTRODUCTION

Developers of today's scientific and engineering applications for high-end computing platforms spend an inordinate amount of their time tuning the performance of their application, often far more time than is required to achieve an initial, working implementation. Further, performance tuning must be repeated each time the code is ported to a new architecture. The process of manual performance tuning involves focusing in on a few key

[1]Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Suite 1001, Marina del Rey, CA 90292-6695, USA. E-mail: {yoonju, pedro, mhall, rflucas}@isi.edu

0885-7458/05/0600-0165/0 © 2005 Springer Science+Business Media, Inc.

computational components of an application. For each component, the programmer derives a sequence of different implementations for performing the same computation. Each variant is first debugged, and then its performance characteristics are evaluated. This lengthy process continues until the programmer either arrives at a suitable variant or, as is often the case, decides to give up on further performance tuning. New variants may be required for different input data sets, or as the application is ported to other platforms.

We illustrate the complexity of this process with an example taken from LS-DYNA. The LS-DYNA application is a general-purpose, nonlinear finite element program capable of solving a vast array of engineering and design problems ranging from bioprosthetic heart valve operation to automotive crash and earthquake engineering.(1) The LS-DYNA application is a commercial derivative of DYNA originally developed at Lawrence Livermore National Laboratory. It was originally designed to take explicit time steps, allowing it to model strong shocks and contact. Recently it has been enhanced to take implicit time steps as well. This improves both accuracy and time to solution in problems like springback. The computational bottleneck of implicit LS-DYNA is the solution of a large sparse system of symmetric indefinite linear equations. The default multifrontal sparse solver used in LS-DYNA was developed by one of the authors,(2) and is the computation that is the focus of this paper.

The current implementation has been ported to four different parallel platforms, requiring substantial performance tuning at each port. The code is primarily written in FORTRAN, but different parallel versions use OpenMP, MPI and architecture-specific language extensions. Certain parameters of the algorithm are known by the author to be important to the application's sequential and parallel performance, and certain algorithms are more appropriate, depending on execution context. For this reason, implementation variants of the solver's sub-computations have been developed to improve performance for particular problem sizes and architectures, corresponding to different algorithms and different parameter values. The author would have benefitted greatly from tools to support the generation and selection among these variants.

The goal of this paper is to examine in detail the performance tuning process of the application developer for this solver and use this as a guide towards developing effective tool support to enhance programmer productivity and improve the quality of the result. A key observation is that much of what the developer did could be systematized and automated. This work is being performed in the context of the ECO project, which is based on the notion of *model-guided empirical optimization*, whereby code variants are generated and executed with representative input data sets so that performance can be measured and compare.(3,4) We begin by describ-

ing LS-DYNA's solver in more detail in Section 2. Section 3 describes a *code isolator* tool, which generates executable sub-programs from large applications, preserving the behavior of the sub-program when run in the context of the full program. We describe code generation strategies to derive variants and their parameter values in Section 4, to accomplish different optimization goals. In Section 5, we focus in on a collection of results based on the selection of a particular performance-oriented parameter. Subsequently, we present related work and a conclusion.

2. LS-DYNA SOLVER

A computational bottleneck of LS-DYNA consists of finding the solution of a very large, sparse, symmetric, indefinite matrix. Depending on the specific size and structure of the input data set, the factorization of this sparse matrix or multiple forward and backward solutions will dominate the run time of the application.

Figure 1(a) depicts a multifrontal elimination tree, which represents how a large sparse matrix has been transformed into a tree of smaller, dense problems (the individual triangles). These small dense problems are called "frontal matrices". The equations in the children of a given node of the tree are eliminated before the equations in their parent. After eliminating the equations in the child nodes, the computation will update (in a sparse fashion) the entries in the parent node and then proceed to eliminate the equations in the parent node.

Because the frontal matrices are dense, they can be factored using highly optimized arithmetic routines such as those found in the BLAS3. This is illustrated in Fig. 1(b), in which a dense symmetric frontal matrix is subdivided into panels, with triangular diagonal blocks. Once the

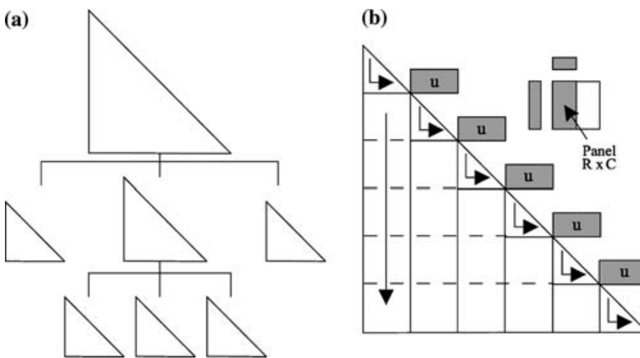

Fig. 1. LS-DYNA solver.

equations in the leading panel have been eliminated, the remaining panels can be updated with dense matrix–matrix multiplication kernels. These matrix–matrix multiplies are naturally written in Fortan to use SAXPY inner loops. Selection of panel size (height and width) significantly impacts register usage, instruction-level parallelism and memory hierarchy behavior, and thus was an optimization parameter that was evaluated empirically by the application developer. The computation within a panel is fully unrolled. Thus, too small a panel leads to under-utilization of resources, while too large a panel can lead to stalls waiting on overcommitted resources. Finding the appropriate panel size is therefore architecture specific. Tools to automate selection of panel size for the SAXPY implementation would greatly accelerate the process of performance tuning for a new architecture.

Alternatively, the matrix–matrix multiplies can be implemented using dot products (SDOT), which can be used in place of the SAXPY kernels. For the SDOT-based kernels, the parameters include the number of eliminated equations and the block size for the updated columns. These parameters affect memory hierarchy performance, and the appropriate values are also architecture specific. Tool support to derive optimal values of these parameters would also be useful.

Whether SAXPY or SDOT will perform better is a complex function of the size of the frontal matrix, the numerical properties of the matrix, the precision of the arithmetic, and architectural features of the computer that performs the calculations. For example, SAXPY loops tuned for vector processors will in general not perform well when factoring large frontal matrices on cache-based machines, as the latter typically do not have sufficient main memory bandwidth to feed the arithmetic units. Further, even for a fixed architecture, neither the SAXPY nor the SDOT matrix–matrix multiply kernels is expected to be universally optimal even on a specific architecture, as the frontal matrices are of varying sizes. It would be highly desirable to have a tool that could identify where each of the many possible implementations is optimal, and dynamically switch between them during the application's execution. At a higher level, the ordering of equations in the factorization is another parameter that can significantly impact performance, as is discussed elsewhere.[(5)]

Ordering the equations in the factorization, deciding between SAXPY and SDOT, and selecting parameter values for panel sizes and block sizes have all been done manually to date. In the remainder of the paper, we describe tools to support this process. We also present a collection of empirical results, obtained automatically, that measure the performance impact of varying panel size for SAXPY and block size for SDOT.

3. ISOLATING CODE FRAGMENTS

Large application codes can consist of millions of lines of code; the current LS-DYNA is almost half million lines of code. Typically, the key computations in such a code comprise only a small fraction of the overall size.[2] Further, the overall program may execute for hours, or sometimes even days. To tune application performance, it would be far less cumbersome to run just the key computation, isolated from the rest of the program. This permits bypassing the irrelevant computation and facilitating quick compiles and runs of the variants.

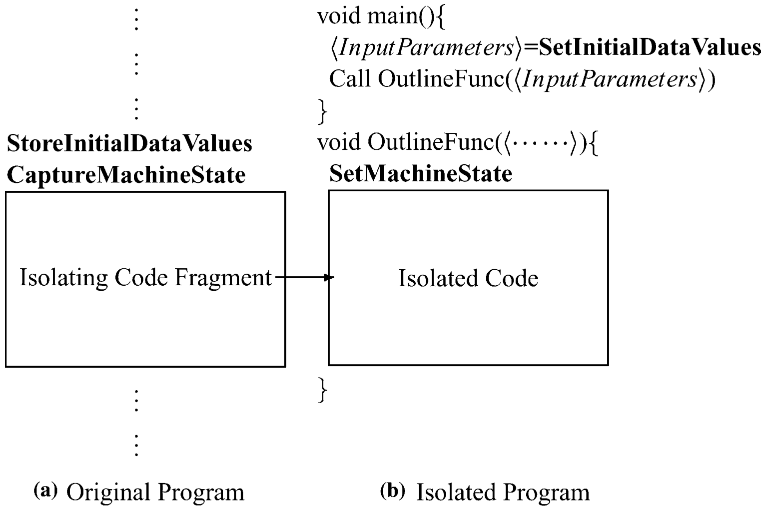

Fig. 2. A code isolator.

For this purpose, we have developed a tool called a *code isolator*.[6] The goal of the code isolator is to produce an isolated version of the code that can be *compiled* and *executed* with inputs representative of the original program, and initializing the *machine state* such that execution of the isolated code segment mimics the performance behavior of the computation when executing in the full program.

Figure 2 presents an overview of how the code isolator achieves these results. Let us assume that a core computation has been selected for isolation. Identifying code that merits isolation is beyond the scope of this

[2] This is often referred to as the 90-10 rule of thumb: a program typically spends 90% of its time in 10% of the code.

paper, but a number of automatic or user-directed techniques can be used; as examples, the user could select and annotate the computation, it could be selected by the compiler as a result of profiling, or it could be triggered by code that failed to meet a user's performance expectations.[7]

As is shown in Fig. 2(a), the code isolator instruments the original application to capture initial machine state and representative input data values, just prior to executing the code to be isolated. The tool generates a version of the isolated code, as shown in Fig. 2(b), with the data set and machine state initialized. The isolated code is encapsulated in a function, which is invoked by a main program.

We have implemented the code isolator in the Stanford SUIF compiler as part of the ECO project,[3,8] with some modest manual intervention, and we describe its features in the remainder of this section. We first describe how to generate isolated code that can be compiled, followed by code that can be executed and subsequently describe how to initialize machine state of the isolated code.

3.1. Compilable Isolated Program

The first step of the code isolator is to automatically extract from application programs key code segments, such as a loop nest computation. In Fig. 2, a code segment is selected from the original program to create the isolated procedure called *OutlineFunc*($\langle InputParameters \rangle$). The isolated procedure is generated by the outline transformation based on the SUIF compiler library.

Outlining takes a body of code and forms a function; input data that is live on entry to the code is passed as a parameter. The isolated program consists of two parts: the main procedure and the outlined procedure (see Fig. 2(b)). The main procedure calls the outlined procedure and includes all parameters in its local symbol tables.

3.2. Executable Isolated Program

In the second step, the code isolator creates an executable program. The set $\langle InputParameters \rangle$ must be initialized prior to invoking the outline procedure in the isolated program. First of all, if the isolated code includes undefined sizes of arrays, then the lower and upper bounds of array sizes are determined through instrumenting the original program. Second of all, data values are extracted from the original program, and are initialized in the isolated program. In Fig. 2(a), the *StoreInitialDataValues* module saves input parameter data from the original program into a file.

In Fig. 2(b), the *SetInitialDataValues* module assigns initial data values to the $\langle InputParameters \rangle$ before setting up the call to the outlined procedure in the main procedure of the isolated program, by reading the input data from a file.

3.3. Setting the Machine State

In the final step of the code isolator, the state of the machine from the original program is captured through instrumentation, and is set during initialization in the isolated code. The machine state describes all relevant state of the target architecture, including register, cache memory, TLB, and so on, that will impact the performance of the isolated code. In Fig. 2(a), the *CaptureMachineState* module executes just prior to the isolated code fragment in the original program. As a starting point of this research, we focus on achieving comparable cache behavior from the original code.

The tool must identify what data accessed by the isolated code is already in cache, and ideally, it is also important to know where the data is located in cache to predict when replacement will occur. For these purposes, we capture an address trace for a small portion of the code that executes immediately prior to the isolated code fragment. Using a cache simulator, we determine which of these addresses remain in cache at the entry of the isolated code. We also capture an address trace for the isolated code, and determine what data accessed by the isolated code will be in cache as a result of executing its preceding code. To reduce the amount of preceding code that must be traced, we use analysis to identify how much code must be executed to achieve the comparable cache state. Conceptually, we examine the code prior to the isolated code fragment in a reverse traversal. Using an adaptation of the region-based interprocedural array data-flow analysis described in,[9] we determine the minimum data footprint of each code region. When analysis can prove that the data footprint of preceding code exceeds the cache capacity, there is no need to traverse the code further. This is the point in the preceding code where address tracing should begin.

The module *SetMachineState* initializes the cache to include the cache blocks identified by the analysis to capture the machine state, previously described. To set the machine state in the isolated code, after initializing the data values, we flush the cache. Subsequently, we prefetch into cache the desired cache blocks. Using the array index of the first element of the cache line, we insert prefetch instructions into the source code. On the SGI Origin, our target architecture, we insert a set of *#pragma prefetch_ref* directives, which are supported by the MIPSpro compiler.[10]

4. GENERATING AND SEARCHING VARIANTS

Once a core computation has been isolated, the compiler needs to generate a collection of program variants to be tested and compared. If we consider the LS-DYNA solver in Section 2, the variants include different optimization parameters (unroll factors for SAXPY's panels, unroll factors and tile size for SDOT's blocks), different algorithm choices (SAXPY versus SDOT and different equation orders), and the composition of different variants.

For specific optimization parameters such as unroll factor selection or tile size, it is somewhat straightforward to generate the variants, although it is more difficult to determine which variants are of interest. Compiler analyses and models such as reuse analysis and modeling of cache behavior can suggest appropriate loop ordering options, which loops should be unrolled, which loops should be tiled, and other optimizations such as copying and prefetching for cache, as discussed in previous work.[(4)] Selecting a particular parameter value requires searching the space of possible values for that parameter. That search space can be quite large, and searching the entire space can be prohibitively expensive in the general case. Thus, it is important to exploit compiler knowledge of optimization properties to prune from the search unprofitable parameter values. For example, a tile size whose footprint exceeds the cache size is too large. Once we consider multiple levels of the memory hierarchy, our preliminary experience suggests enough constraints can be derived on variants and parameters to limit the search to something manageable, a subject of current and future work.[(3,4)]

For algorithm variants, user guidance is required, both to implement the different variants, and to indicate that the two are equivalent. A linguistic mechanism called a *selector* for specifying variants and dynamic criteria for selection of variants is described elsewhere.[(5)] In the absence of user-specified selection criteria, a tool could automatically search the space of alternative implementations under different run-time conditions.

5. EXPERIMENTS

We now examine the benefits of an empirical approach to derive the selection of a parameter value for a specific target high-performance machine and for a selected phase of the solution of a sparse linear system of equations—the factorization phase as described in Section 2. We present preliminary experiments for two variants of the elimination step in the factorization phase. One step uses a SAXPY-based kernel computation for which we vary the panel size. Another step uses a SDOT-based kernel for

which we vary the block size. Our target single processor platform consists of a 195 Mz MIPS R10K CPU with a MIPS R10010 FPU, a main memory of 3 GB, L1 data and instruction caches of 32 KB each and a unified instruction/data L2 cache of 4 MB. The entire application was compiled with the native F90 compiler with the –03 flags.

5.1. Input Data Set

For this experiment we used a data set derived from an automotive crash problem, named *Hood*, which simulates the mechanical deformation of a hood of a vehicle in a frontal impact scenario as depicted in Fig. 3. This *Hood* data set input consists of a sparse-matrix for a linear system with 235 K equations using approximately 59 MB of primary storage. The symbolic decomposition of the system yields an elimination tree with 14,663 *super-nodes*. The distribution of the size of the resulting frontal matrices is depicted in Fig. 4(a). There are a large number of small nodes in the tree and relatively few large ones, near the root. The largest frontal matrix has approximately 2100 equations. For a specific implementation of the algorithm using SDOT kernels, Fig. 4(b) shows the factorization time on average for each sub-matrix size, while Fig. 4(c) presents the cumulative execution time distribution based on sub-matrix size. As can be seen from the figures, despite the distribution heavily biased towards smaller frontal matrices, the bulk of the work is concentrated in the larger, less numerous matrices, as the amount of work per sub-matrix is cubic.

Fig. 3. Hood frame deformation in a frontal impact scenario.

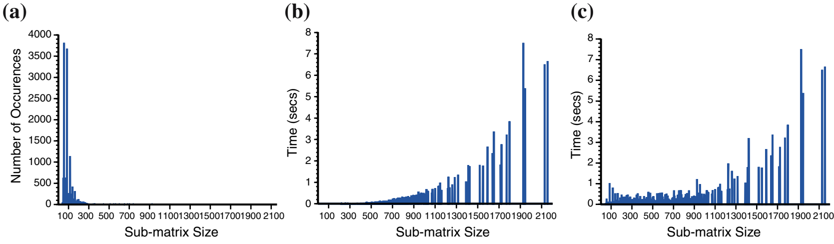

Fig. 4. Sub-matrix size distribution (a), factorization time (b), and cumulative factorization time (c) for the Hood input data set.

5.2. Panel Size for SAXPY Kernels

In this experiment, we compute the performance of the SAXPY kernels as a function of panel shape. For each run, we execute the solver holding the panel shape fixed for all sub-matrix sizes. We have focused our attention on the selection of the panel size and ratio of its height and width, i.e., the number of equations that are eliminated from a panel of columns of the matrix during an elimination.

Panel shape is designated by *xRyC*, with varying designations of *x* and *y* corresponding to the number of rows and columns in a panel, respectively. For each panel shape, we perform within an inner loop body the entire execution of the SAXPY operation resulting in the elimination of the pivot block from a panel of columns of the triangular matrix. The code transformation applied is *unroll-and-jam*;[11] two outer loops of a 3-deep loop nest are unrolled and the resulting inner loop bodies are fused together.

To support this experiment for a wide variety of panel shapes, we have developed a set of FORTRAN code generation functions specific to the triangular nature of the storage of the matrix. These code generators create a partial calling tree that generates the subroutines that carry out the elimination. It also includes the sophisticated code that handles the boundary condition when the number of equations being updated does not evenly divide the number of equations in the panel. The code generation functions also modify the code that invokes these subroutines and are responsible for sweeping through the entire sub-matrix of each node in the elimination tree as described in Section 2. For each panel selection, the script that supports the experiment generates the appropriate file with the corresponding functions and recompiles and links the application generating a new executable image. This approach aims at reducing the size of the executable since accumulating all of the binaries in a single executable could lead to instruction cache misses. For each panel size,

the scripts generate three new subroutines. These vary in size according to the panel shape as their internal loops are explicitly unrolled. Given the already large executable size of the selected target application, the added image size is very negligible. Lastly, the script and code generation function can also include library calls to *PAPI*[12] to monitor the execution behavior of the code for each panel through the elimination process.

Figure 5 presents the speedup results for several of the panel shapes. The baseline execution time for the performance comparisons is the factorization time in which individual equations are eliminated from all the remaining equations using Gaussian-Elimination (GE) within each frontal matrix. The results do not include the panel shapes of $1R1C$ as the current code generation scheme would generate too many subroutine calls, negatively biasing the performance of the generated code. In the future we plan to address this issue by modifying the call-chain appropriately. For all of the results reported here we collected the data corresponding to five executions and used the average of the three median runs, i.e., we have ignored the best and worst performing runs.

Overall the results show a maximum speedup of 2.6 for the panels with shapes $10R2C$ and $10R6C$ over the single equation GE implementation. (Note that these speedups are less than previously reported speedups in[8] as we have manually improved the performance of the baseline by the application of scalar replacement.) We see from the results that it is more beneficial to unroll rows versus columns. For example, the result for $4R2C$ is 1.17 times faster than that of $2R4C$, presumably due to more data reuse for data in different rows within a column. The results also reveal a saturation effect reflecting diminishing returns for very substantial

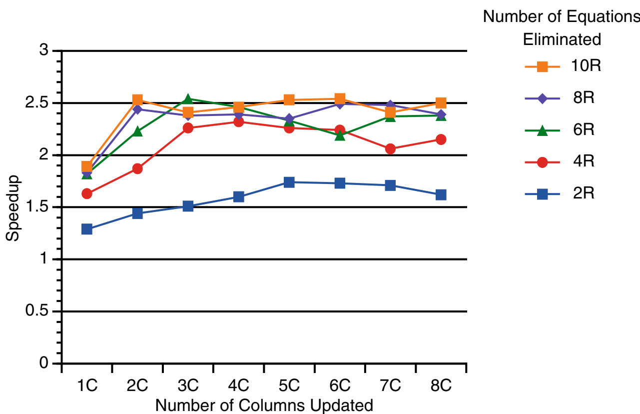

Fig. 5. Hood speedup on MIPS R10K target machine.

programming and verification efforts. This is due to register pressure and floating-point stalls for large unroll factors.

We are currently investigating automatic ways of understanding the implication of architecture features that directly contribute to this behavior. For example, larger panel selection and subsequent unrolling can lead to increased required bandwidth and register pressure. The inability of a compiler to adequately reschedule FP operations in the presence of pipeline hazards can also lead to further performance degradation.

5.3. Block Size for SDOT Kernels

We also present preliminary results for varying the block size for the alternative SDOT-based kernels. The block size for the SDOT kernels determines the length of the inner product loops. Unlike the previous experiment, the unroll factors for the two innermost loops are held fixed at unroll factors of four each. As in the previous experiment, we have instrumented this implementation using *PAPI*, to measure both wall-clock time and values of various performance monitors.

Figure 6 depicts the performance of block sizes for distinct sub-matrix sizes. Each curve represents a particular block size. As each sub-matrix occurs multiple times during a given factorization, the results reflect the average MFLOPS for each sub-matrix size.[3] The results reveal the optimal selection of the block size for three regions of the space of sub-matrix sizes for this input data. A block size of 128 yields the best results for the small sub-matrices of *Region*1 (less than 255). In *Region*2 (between 255 and 265), the block size of 48 works best. Finally, the block size of 64 produces the best performance for the large sub-matrices in *Region*3 (greater than 265). Using these regions in an adaptive strategy, we measured the performance of an adaptive implementation that dynamically selects one of these three block sizes depending on the region of the sub-matrix size. The additional curve, called *Adaptive*, presents the results of this dynamic strategy, which leads to the best overall performance.

Table I provides insight into how the block size impacts performance as a function of sub-matrix size. The table depicts the performance results in terms of both FLOPS and Cache Misses for L1 and L2 caches for the five different block sizes and the *Adaptive* algorithm, given fixed unroll factors. Among the five fixed block sizes, the best results are obtained for a block size of 64, revealing that the L1 cache behavior is best for the smallest block size, whereas L2 behavior improves as the block size increases. The best result has neither the smallest number of L1 cache

[3] The x-axis in this plot bins the sub-matrix size in bins of size 5.

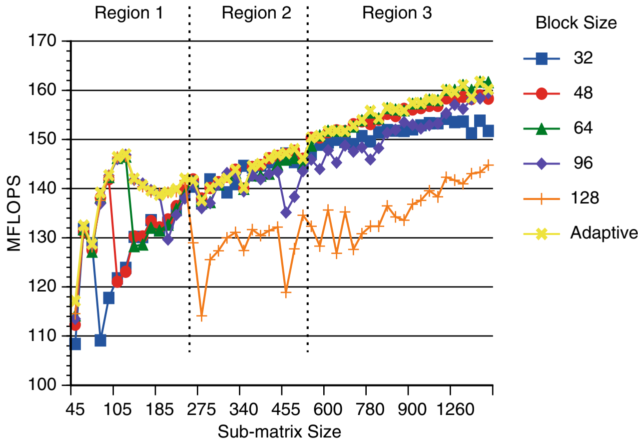

Fig. 6. MFLOPS distribution with different block sizes.

Table I. Performance Results for Different Block Sizes, Using SDOT

Block size	FLOPS	L1 Cache misses	L2 Cache misses
32	135.33M	339M	9.1M
48	139.65M	380M	8.6M
64	140.83M	400M	7.6M
96	139.02M	671M	7.4M
128	127.68M	1599M	7.2M
Adaptive	141.25M	395M	7.9M

misses nor the smallest number of L2 cache misses, but rather represents the “sweet spot" in simultaneously optimizing for both levels of the memory hierarchy. When we use the *Adaptive* block size selection, the number of FLOPS increases by around $0.42M$ beyond that of a block size of 64. The results for a block size of 64 are close to that of the *Adaptive* algorithm, since from Fig. 4(c), roughly 84% of the execution time is spent on sub-matrix sizes in Region 3, while Region 1 and 2 account for less than 7% and 10% of the execution time, respectively.

5.4. Discussion

Taken together, these results show that automating the search for optimization parameters is within the capabilities of current compilation systems and can identify parameters yielding the optimal performance for a particular architecture and input data set. Interestingly, the optimal

parameter values identified through this automatic approach were also found by the developer using a manual strategy, but required a substantial increase in effort over our approach. There is much work to be done to compose a collection of variants and parameter values, but this is the subject of current work in the ECO project.[4]

6. RELATED WORK

There are several on-going research projects in empirical optimization of scientific libraries, ATLAS,[13] PhiPAC[14] and domain-specific libraries UHFFT,[15] FFTW,[16] and SPIRAL.[17] The ATLAS system generates high performance implementations of the Basic Linear Algebra Subroutines (BLAS) based on empirical performance data. The PHiPAC system generates highly tuned implementations of matrix multiply, with performance comparable to hand-tuned vendor libraries, by searching a large space of potential optimizations. The FFTW system uses a combination of static models and empirical techniques to optimize FFTs. The SPIRAL system generates optimized digital signal processing (DSP) libraries by searching a large space of implementation choices and evaluating their performance empirically.

There has been extensive research on improving the cache performance of scientific applications (e.g., Refs. 11, 18 and 19). Loop optimizations for improving data locality, such as tiling, interchanging and skewing, focus on reducing cache capacity misses as well as conflict misses which play an important factor on the performance of tiled loop nests (see e.g., Refs. 20–23). Most of the work in this area has focused on the development of analytical, static models, rather than relying on empirical performance measures. As such researchers have developed a wide range of models for cache miss analysis with varying degree of accuracy and cost,[24,25] as a way to guide the application of data transformations (e.g., tile copying[26]).

Other researchers have also recognized the lack of adequate programming language support for applications that must react to changing environments. Currently, programmers must intermix their application code with run-time system calls that implement the desired adaptation or optimization policies. The resulting code is complex and virtually impossible to port and maintain. In ADAPT,[27] researchers have defined new languages used exclusively to specify adaptation policies, triggering events and performance metrics. Programmers develop their base application textually independently from their adaptation specification. An alternative approach extends existing languages to provide hints or directives to

the compiler about the dynamic nature of the application. Adve *et al.*[28] describe an extension to the class hierarchy of an objected-oriented model of computation that defines three basic concepts for dynamic adaptation—adaptors, metrics and events.

7. CONCLUSIONS

This paper has presented a case study illustrating how software tools can be used to greatly accelerate the process of performance tuning, leading to better application performance as well as increasing productivity of programmers of high-end systems. Examining issues in optimizing LS-DYNA has influenced the technology we are developing in the ECO project.[3] In this paper, we have described a tool for isolating code segments from large programs, such that the isolated code can be used in place of running the full application. We have described automatic code generation of parameterized implementation variants. We have also presented a set of results where parameterized variants are generated and compared for a sparse solver in LS-DYNA. Our long-term goal is to develop tools and methodologies that allow a programmer to discover performance bottle-necks by a mixture of static analysis and profiling, automatically generating variants using known compiler-based optimizations, and empirically determining the best. This will dramatically reduce the human time and cost associated with performance optimization that inevitably crops up when codes are ported to new systems or extended to address new problems.

ACKNOWLEDGMENT

Work sponsored by the National Science Foundation (NSF) under award ACI-0204040.

REFERENCES

1. LS-DYNA User's Manual V. 960, Livermore Software Technology Corporation, http://www.lstc.com (March 2001).
2. C. Ashcraft and R. F Lucas, A Stackless Multifrontal Method, in *Proc. 10th SIAM Conference on Parallel Processing for Scientific Computing* (March 2001).
3. N. Baradaran, J. Chame, C. Chen, P. Diniz, M. Hall, Y. Lee, B. Liu, and R. Lucas, ECO: An Empirical-based Compilation and Optimization System, in *Proc. of the Workshop on Next Generation Software, held in conjunction with IPDPS'03* (April 2003).

4. C. Chen, J. Chame, and M. Hall, Combining Models and Guided Empirical Search to Optimize for Multiple Levels of the Memory Hierarchy, in *Int. Symposium on Code Generation and Optimization (CGO'05)* (March, 2005).
5. P. Diniz and B. Liu, Selector: An Effective Technique for Adaptive Computing, in *Proc. of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC'02)* (July, 2002).
6. Y. Lee and M. Hall, A Code Isolator: Isolating Code Fragments from Large programs, in *Proc. of the 17th Workshop on Languages and Compilers for Parallel Computing (LCPC'04)* (September, 2004).
7. J. S. Vetter and P. Worley, Asserting Performance Expectations, in *Proc. of Supercomputing'02* (November, 2002).
8. P. Diniz, Y. Lee, M. Hall, and R. Lucas, A Case Study Using Empirical Optimization for a Large, Engineering Application, in *Proc. of the Workshop on Next Generation Software, held in Conjunction with IPDPS'04* (April, 2003).
9. M. Hall, S. Amarasinghe, B. Murphy, S. Liao, and M. Lam, and M. Lam, Interprocedural Parallelization Analysis in SUIF, in *ACM Trans. on Programming Languages and Systems* (2005).
10. MIPSpro C and C++ Pragmas, Document Number 007-3587-003, 1998, 1999 Silicon Graphics, Inc.
11. S. Carr and K. Kennedy, Improving the Ratio of Memory Operations to Floating-Point Operations in Loops, in *ACM Trans. on Programming Languages and Systems* (TOPLAS), 15(3), pp. 400–462 (July, 1994).
12. K. London, J. Dongarra, S. Moore, P Mucci, K. Seymour, and T. Spencer, End-user Tools for Application Performance Analysis, Using Hardware Counters, *Intl. Conference on Parallel and Distributed Computing Systems* (August, 2001).
13. C. Whaley and J. Dongarra, Automatically tuned linear algebra software, in. *Proc. of Super-computing '98* (1998).
14. J. Bilmes, K. Asanovic, C.-W. Chen, and J. Demmel, Optimizing Matrix Multiply using PHiPAC: Portable High-Performance ANSI-C Coding Methodology, in *Proc. of the ACM International Conference on Supercomputing '97* (1997).
15. D. Mirkovic and S.L. Johnsson, Automatic Performance Tuning in the UHFFT Library, in *Proc. of the International conference on Computational Science (ICCS'01)* (May, 2001).
16. M. Frigo, A Fast Fourier Transform Compiler, in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)* (June, 1999).
17. J. Xiong, J. Johnson, R. Johnson, and D. Padua, SPL: A Language and Compiler for DSP Algorithms, in *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI'01)* (June, 2001).
18. M. Wolf and M. Lam, A Data Locality Optimization Algorithm, in *Proc. of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)* (June, 1991).
19. M. Wolfe, More iteration space tiling, in *Proc. of Supercomputing '89* (November, 1989).
20. J. Chame and S. Moon, A Title Selection Algorithm for Data Locality and Cache Interference, in *Proc. of the 1999 ACM International Conference on Supercomputing' 99* (June, 1999).
21. S. Coleman and K. McKinley, Tile Size Selection Using Cache Organization and Data Layout, in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)* (June, 1995).

22. G. Rivera and C.-W. Tseng, Data Transformations for Eliminating Conflict Misses, in *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI'98)* (June, 1998).
23. M. Lam, E. Rothberg, and M. Wolf, The Cache Performance and Optimization of Blocked Algorithms, in *Proc. of the 4th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'91)* (April, 1991).
24. S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck, Exact Analysis of the Cache Behavior of Nested Loops, in *Proc. of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)* (June, 2001).
25. S. Ghosh, M. Martonosi, and S. Malik, Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity, in *Proc. of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)* (October, 1998).
26. O. Temam, E. Granston, and W. Jalby, To Copy or not to Copy: A Compile-time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts, in *Proc. of Supercomputing '93* (November, 1993).
27. M. Voss and R. Eigenmann, High-Level Adaptive Program Optimization with ADAPT, in *Proc. of the ACM SIGPLAN Conference on Principles and Practice of Parallel Processing (PPoPP'01)* (June, 2001).
28. V. Adve, V. Lam, and B. Ensink, Language and Compiler Support for Adaptive Distributed Applications, in *Proc. of the ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM'01)* (June, 2001).