AESOP: The Autoparallelizing Compiler for Shared Memory Computers

Aparna Kotha, Timothy Creech and Rajeev Barua
Department of Electrical and Computer Engineering
University of Maryland, College Park, 20742
Email: {akotha, tcreech, barua}@umd.edu

***Abstract*—An automatic parallelizer is a tool that converts serial code in C, C++ and Fortran to parallel code. This is an important problem because most hardware today is parallel and manually rewriting the vast repository of serial code is tedious and error prone. We have developed an automatic parallelizing compiler for source code, AESOP targeting shared memory machines. AESOP leverages the LLVM infrastructure and presently works with LLVM-3.0. It targets parallelism for dense array-based codes with affine-based analysis using traditional methods.**

The focus of AESOP is not to break new ground in parallelization theory, but to deliver a robust and fast compiler that can handle large programs while getting good speedups for large, real affine-function codes. It is unique among research compilers in the scope of programs parallelized, with programs totalling over 2 million lines of source code (LOC) being correctly handled including primarily the entire SPEC CPU 2006 suite. Smaller programs totalling a few thousand LOC from OMP 2001, NAS and Polybench benchmark suites are also parallelized. Of the tested programs, AESOP identified a subset of SPEC CPU 2006 benchmarks and most of the other programs with over 100,000 LOC in total that it was able to automatically get good speedups on. The fraction of benchmarks parallelized is small for SPEC CPU 2006 because SPEC benchmarks are predominantly serial programs with no exploitable affine parallelism. We have not seen speedup numbers on parallelizing the entire SPEC CPU 2006 benchmark suite (or any significant portion thereof) using affine methods in any other published work.

In comparing with the leading open-source polyhedral-based compiler (Pluto), AESOP delivered significantly better speedups on Polybench benchmarks owing to its greater tolerance of different types of affine access patterns, and better affine-based cache optimizations. Pluto was not able to compile any of the larger benchmark suites we list above.

I. Introduction

An automatic parallelizer is a tool that takes as input serial code and produces as output parallel code. Traditionally such tools have been designed for source code with affine loops in them. Affine loops are loops that contain array indices that are a linear combination of the induction variables of the loop nest *i.e.* loops containing `A[i][j]`, `A[i + 3j + 5][4i + 2j + 10]`, `A[i][i]` are all affine whereas loops containing `A[i/2]` or `A[i^2]` are not.

There have been two schools of thought in literature to build affine automatic parallelizers: (i) parallelizers built on the traditional models using distance/direction methods, and (ii) parallelizers built on the polyhedral analysis.

A. Traditional methods

Traditional methods [15] [14] [7] [3] [4] [11] [13] are those which are based on modeling loops as the units of consideration, where matrices are used to model most concepts including affine indices, iteration vectors, dependence vectors and loop transformations. Methods have been proposed for deciding what order of transformations should be applied and in what order.

The traditional methods are divided into three steps: (i) representation of the dependencies using distance or direction vectors characrizing all the dependences present in a loop; (ii) decision algorithms to transform loop nests, to further maximize parallelization in a loop using algorithms to reason about different loop transformations and (iii) then methods to generate target code using standard compiler techniques.

B. Polyhedral methods

Polyhedral methods [12] [10] [9] [6] [5] [8] are the second class used for affine analysis and automatic parallelization. They represent each statement in an affine loop separately as a point in an iteration domain. After this is done decisions to transform the loop are taken using affine scheduling functions each of which maps each run-time statement instance to a logical execution date and parallel code is generated using syntax tree construction schemes that consist of a recursive application of domain projections and separations.

Polyhedral methods have the following three advantages over traditional methods:

- First, polyhedral models handle imperfectly nested loops seamlessly in their model.
- Second, they are able to model dependence between every dynamic instance in the loop.
- Third, complex affine transformations can be modeled as scheduling functions, which in a few instances, can discover multiple traditional transformations in one step;

Traditional methods have the following advantages over the polyhedral methods:

- First, their worst-case complexity is in the order of polynomial complexity against the exponential complexity of polyhedral methods. This has been shown to be acceptable for smaller programs, but no one has demonstrated a working polyhedral compiler that can handle programs in the hundreds of thousands or millions of LOC. Their scalability has not been tested since the brittleness of existing polyhedral implementations has prevented their testing on large programs.

- Second, the implementation complexity of traditional methods is significantly lower than that of polyhedral methods.
- Third, polyhedral methods are known to have strict requirements on what kind of the codes they can handle, and cannot parallelize codes even slightly outside their requirements. This is problematic for large, real world programs for which rewriting by hand is tedious and defeats the purpose of automatic parallelization. Traditional methods are more forgiving.

In balance we decided to implement our technologies in the traditional model.

II. Affine Automatic Parallelizer

In this section we present a detailed description of the architecture of the AESOP source parallelizing compiler.

The block diagram of the affine automatic parallelizer developed by us is shown in figure 1. We describe these blocks and flow briefly first followed by their detailed description in the following subsections.

First, the serial llvm IR is fed into the *undoing compiler optimizations* module. These include the "Loop Simplify" and "Induction Variable Simplify" passes from LLVM. The LLVM IR obtained after this module is still serial (lets call it "new serial LLVM IR"), however the loops are simplified *i.e.* contain only one exit block when possible and cannonical induction variables are introduced into loops whenever possible. Such simplification of loops is essential to run our affine analysis on them and then generate parallel code.

Second, this new serial LLVM IR is then passed into the *loop dependence analysis* block, which consists of the *alias analysis* module and the *distance vector generator*. Every pair of memory accesses in a loop are passed into the *alias analysis* module and the *distance vector generator*. The alias analysis passes that are called in our parallelizer are the standard ones present in LLVM. We did not write any new alias analysis passes. If using the standard alias analysis passes from LLVM we discover that the two references do not alias, we can say that there is no dependence between them and that the distance vector associated with this pair of references is $(0, 0, \cdots, 0)$ consisting of as many zeroes as the loop nesting depth. If alias analysis is unable to prove that the two references do not alias with one another we pass them onto the distance vector module. This module consists of the delta and banerjee's inequality tests from affine literature. It helps us discover distance/direction vectors for this pair of memory references. After we have analyzed every pair of accesses in the loop we would have generated all the distance/direction vectors associated with this loop. Hence, the output of the *loop dependence analysis* block are all distance/direction vectors associated with each loop.

Third, the distance/direction vectors and the new serial LLVM IR are passed into the *parallelizer* block that talks to the *decision algorithm* block. The decision algorithm decides which loop dimensions to parallelize. The decision algorithm talks to scalar reduction and takes a decision based on the

```
for i from lb_i to ub_i
  tmp = 0;
  for j from lb_j to ub_j
   A[i,j] = A[i,j] + tmp;
   tmp = B[j] + 10;
  end for
end for
```

Fig. 2. Example of a loop that carries a scalar dependence

scalar dependence information and *array dependence information*(distance/direction vectors). Subsection II-A presents the method used to collect scalar dependence information from loops. It is important to analyze the scalar dependences in a loop since if there is a loop carried scalar dependence in a loop it may prevent parallelization of that dimension. However, some loop carried dependences such as the ones that perform reduction operations do not prevent parallelization. These will also be discussed in detail in section II-A. The details on how we make the parallelization decision is presented in section II-B. The output of the parallelizer block called the parallelization decision is a list of loops that we have decided to parallelize.

Finally, after the parallelization decision is taken and we have decided which loops to parallelize we pass this information along with the new serial LLVM IR to the *parallel code generator* block which generates SPMD parallel code for each of the parallel loops. The details of the *parallel code generator* are described in subsection II-C.

A. Scalar dependencies

A scalar dependence is present in a loop if a location is defined in one iteration of the loop and used in another iteration of the loop. We recognize scalar dependencies from source by analyzing def-use chains. All variables are checked to see if they are defined in one iteration and used in a later iteration. This is a check on def-use chains of variables, to see if the variable is live at the exit block of the loop. We check for the presence of scalar dependencies at every loop depth as a certain dependence may be present at one depth and not at another loop depth. For example in the code in figure 2 *tmp* has a scalar dependence on $\texttt{Loop}_\texttt{j}$, however there is no scalar dependence on $\texttt{Loop}_\texttt{i}$. Hence $\texttt{Loop}_\texttt{i}$ can be parallelized in this code. Variable *tmp* may be register allocated in a binary (say to `tmp_r`) and data flow will tell us that it is live across the $\texttt{Loop}_\texttt{j}$ but not live across $\texttt{Loop}_\texttt{i}$.

1) Special case of scalar dependence: Reduction: Certain scalar loop carried dependencies such as reduction do not prevent parallelization as known in affine literature [11]. One example loop that contains a reduction on the variable `sum` is presented in figure 3. For every scalar variable that is live across the loop, we check to see if this is due to a reduction operation. Reduction operations known and implemented from traditional affine technologies are sum, product and min/max operations. Once such scalar values are recognized using the standard rules of reduction, this scalar value is marked as reduction and no more prevents parallelization. For *e.g.* , the variable `sum` is marked as a reduction on $\texttt{Loop}_\texttt{j}$. If this loop level

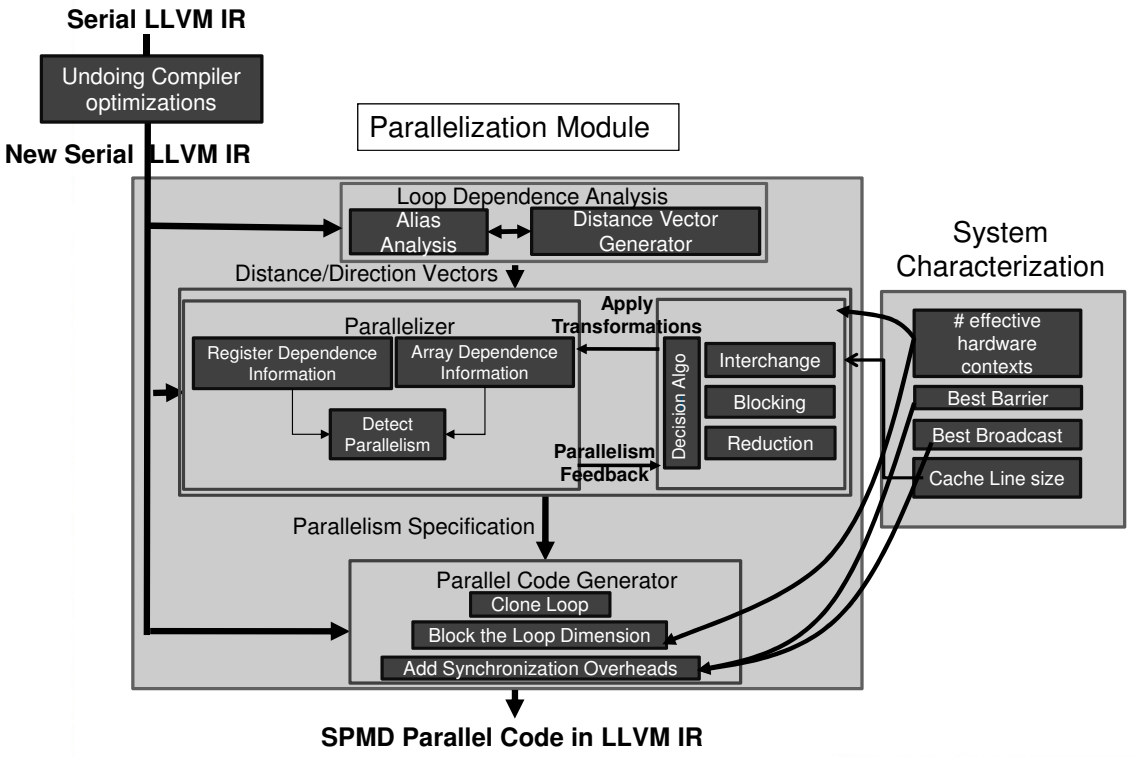

Fig. 1. Detailed diagram of the Affine Parallelizer

is chosen for parallelization then code is generated such that each parallel thread accumulates a part of `sum` and after all the parallel loops have executed they are all added up. Reduction is a standard transformation and has been explained in detail in [11].

```
for i from lb_i to ub_i
  sum = 0;
  for j from lb_j to ub_j
   sum = sum + A[i,j]
  end for
end for
```

Fig. 3. Example of a loop that carries a reduction dependence

B. Deciding Partitions

In this section we present the algorithm we use to decide which loop dimensions to parallelize in any affine loop nest.

Say that a loop nest `(i,j)` has a dependence vector `vecD` = (1,0), indicating that there is a dependence along `i`, whereas there is no dependence along induction variable `j`. So, if we execute all iterations of `i` on one processor then we can parallelize the iterations along `j` among all the processors. Pictorially, this is represented as partition 1 in figure 4, which shows the iteration space as a 2-D matrix of `i` and `j` values. Conversely, if the only dependence vector is $\tilde{D}$ = (0,2), indicating that there is a dependence in steps of two along induction variable `j`, and no dependence along induction variable `i`. In this case we can execute all iterations of `j` on one processor then we can parallelize the iterations along `i` among all the processors. Pictorially, this is represented as partition 2

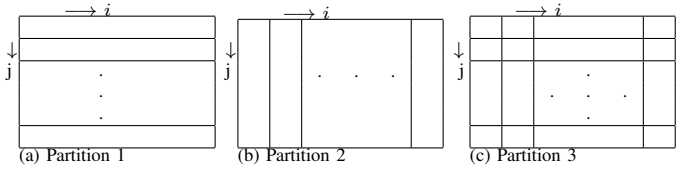

Fig. 4. Different partitions of the iteration space

in figure 4. Partition 3 in that figure can be used when there is no loop-carried dependence on either loop dimension (*i.e.* $\tilde{D}$ = (0,0)).

However, it is not always this simple when dealing with arbitrarily nested loop dimensions. It is essential to have a algorithm to effectively decide which loops to parallelize to maximize speedup from parallelization. We would gain maximum from parallelizing the outer most dimensions that is parallel in loop nests. The algorithm used to determine the outer most dimensions to parallelize in arbitrarily nested affine loops is presented in algorithm 1. Essentially, using the algorithm we choose all loops that can be parallelized for which none of the parent loops is parallel. These loops are added for parallel code generation. A loop level is considered parallel if all distance/direction vectors associated with this loop have a 0 component for it and there is no parallelization preventing loop carried scalar dependence on it.

We now illustrate which loops our algorithm will parallelize for the three arbitrarily nested loops shown in figure 5. In each of the example loops, $\texttt{Loop}_i$ is the outermost loop and for each loop level we have marked (X) to indicate that

Algorithm 1 Algorithm to decide which loop dimensions to parallelize

Input: All loops in the program
Input: Register dependence information, Array dependence information
Output: Loop dimensions to parallelize
for all $Loop_i$ in the program **do**
  if $Loop_i$ is parallel based on register & array dependence information **then**
    if None of the parent loops of $Loop_i$ are parallel **then**
      $Loop_i$ is added to list of loops to be parallelized
    end if
  end if
end for

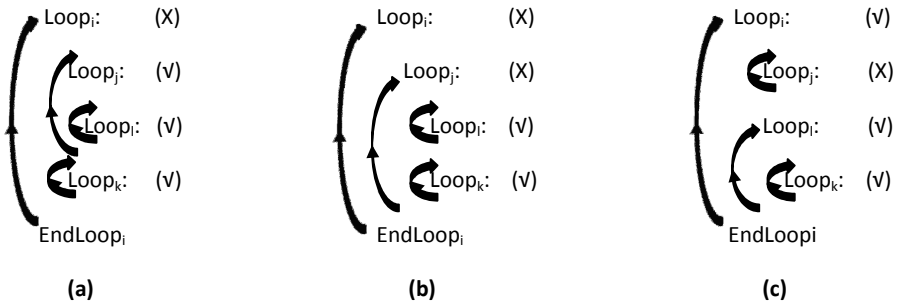

Fig. 5. Arbitrarily nested affine loops to illustrate which loops will be parallelized by our decision algorithm

the loop is not parallel and (√) to indicate that the loop level is parallel. For the loop structure shown in figure 5(a) we parallelize loop dimensions $\texttt{Loop}_\texttt{j}$ and $\texttt{Loop}_\texttt{k}$. We do not recommend $\texttt{Loop}_\texttt{l}$ for parallelization since one of its parent loop $\texttt{Loop}_\texttt{j}$ can be parallelized and has been recommended for parallelization. Next, for the loop structure in figure 5(b) we recommend loops $\texttt{Loop}_\texttt{l}$ and $\texttt{Loop}_\texttt{k}$ for parallelization since none of the parent loops of these two loops $\texttt{Loop}_\texttt{i}$ and $\texttt{Loop}_\texttt{j}$ are not parallel. Finally, for the loop structure in figure 5(c) only $\texttt{Loop}_\texttt{i}$ is recommended for parallelization since it is the outermost loop and is parallel. $\texttt{Loop}_\texttt{l}$ and $\texttt{Loop}_\texttt{k}$ are both parallel, but are not recommended for parallelization since one of their parent loops $\texttt{Loop}_\texttt{i}$ is parallel.

C. Code Generation

After the the distance/direction vectors are calculated, transformations done, parallelization decision taken, and the loop dimensions to be parallelized are decided, code needs to be generated for each parallel loop dimension. Since the body of the loop is executed on all parallel threads, the most convenient and efficient code generation model is the Single Program Multiple Data (SPMD) model. The underlying idea is that the iterations of the loop are divided among threads; hence to keep the code-size increase to a minimum, the same code is executed on all threads using different loop bounds.

From source code, SPMD code can be generated by simply replacing the symbolic values of the lower and upper bounds of loop induction variables by new values. These methods are fairly straight forward as the symbolic information is readily available in source.

$$\texttt{new\_lb} = \texttt{Base} + \texttt{lb}_\texttt{i} * \texttt{size\_j} + \texttt{lb}_\texttt{j} * \texttt{elem\_size} + \frac{\text{PROC\_ID} * (\texttt{ub}_\texttt{j} - \texttt{lb}_\texttt{j}) * \texttt{elem\_size}}{\text{NPROC}} \quad (1)$$

$$\texttt{new\_ub} = \texttt{min}(\texttt{ub\_j}, \texttt{new\_lb}_{\texttt{addr\_reg}} + \frac{(\texttt{ub}_\texttt{j} - \texttt{lb}_\texttt{j}) * \texttt{elem\_size}}{\text{NPROC}}) \quad (2)$$

The `new_lb` and `new_ub` are replaced for each of the parallel thread codes. Data partitioning is not necessary since we target shared memory platforms common in multi-cores.

Generating parallel code requires the use of some parallel thread library. We implement POSIX-compliant *pthreads* calls, given that POSIX is a widely used portable industry standard. POSIX-complaint parallel threads are created once at the start of *main()*, rather than at each loop to avoid paying the steep thread-creation cost multiple times. Only the main thread executes serial code between parallel loops. Parallel threads only execute loop code. When a parallel thread finishes one loop it waits for the main thread to inform it which loop to execute next in a broadcast. The broadcast also contains the values of registers calculated by the main thread that are needed by the parallel loop threads. A barrier is inserted into the binary at the end of every loop.

We can also use the barrier most profitable for the machine we are working on. We have implemented central, tree and butterfly barriers. We also compare these to the platform specific barrier present on any systems (such as pthreads_barrier).

III. AESOP: System Implementation

AESOP is open source and can be downloaded from aesop.ece.umd.edu. It has been developed at the University of Maryland, College Park. Two versions are available for download: (i) a source code release, and (ii) a binary distribution for x86-64 Ubuntu 12.10 targeting i386. We note that while AESOP may work on any system supported by LLVM, we have only heavily tested targeting 32-bit Linux.

We now explain briefly how AESOP works. We know that by feeding serial LLVM IR to the affine parallelizer we obtain SPMD parallel LLVM IR. We have three scripts that help us do this as part of AESOP for source code: (i) *aesopcc* which uses *clang* [1] (a C language front-end for llvm) to generate LLVM IR for source code written in C and then feeds it to the affine parallelizer; (ii) *aesopgcc* that uses *dragonegg* [2] plugin (a plugin that integrates the LLVM optimizers and code generator with GCC) to generate LLVM IR from C/C++ source code programs and feeds it into the affine parallelizer and (iii) *aesopgfort* that uses *dragonegg* [2] plugin to generate LLVM IR from fortran programs and feeds it to the affine parallelizer. All the three scripts use the x86 back-end of LLVM to obtain parallel executable for the source code fed to them.

We have tested AESOP on benchmarks whose source code exceeds 2 million lines of code. The testing infrastructure is available for download along with AESOP and contains benchmarks from polybench, SPEC2006, OMP2001, NPB and hpcc. There is also a very easy method to add new benchmarks to AESOP. The barriers used for generating parallel code are in a library and is also available for download with AESOP.

IV. results

Our benchmarks regularly are compiled on the AMD Opteron(TM) 6212 machine. We also upload our results to a shared google doc regularly. Figures 6 and 7 present the speedups for polybench and the rest of the benchmarks as of March 2013. We will revise these speedups if there is any significant difference from time to time. On an average we obtained 1.88X speedup from all the benchmarks. Our benchmarks show that we scale well and have parallelized benchmarks exceeding million lines of code.

References

[1] clang: a C language family frontend for LLVM. http://clang.llvm.org/.

[2] DragonEgg - Using LLVM as a GCC backend. http://dragonegg.llvm.org/.

[3] U. K. Banerjee. Unimodular transformations of double loops. In *In proceedings of the third Workshop on Languages and Compilers for Parallel Computing*, pages 192–219, August 1990.

[4] U. K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[5] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[6] C. Bastoul and P. Feautrier. Improving data locality by chunking. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 320–334, Berlin, Heidelberg, 2003. Springer-Verlag.

[7] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The next generation in parallelizing compilers. In *Proceedings of the workshop on languages and compilers for parallel computing*, pages 10–1. Springer-Verlag, Berlin/Heidelberg, 1994.

[8] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[9] P. Boulet, A. Darte, and G. andr Silber. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Computing*, 24, 1997.

[10] M. Griebl. Automatic parallelization of loop programs for distributed memory architectures, 2004.

[11] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[12] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 228–237, New York, NY, USA, 1999. ACM.

[13] K. S. McKinley. A compiler optimization algorithm for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1998.

[14] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):452–471, 1991.

[15] M. J. Wolfe. *Optimizing supercompilers for supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1982.

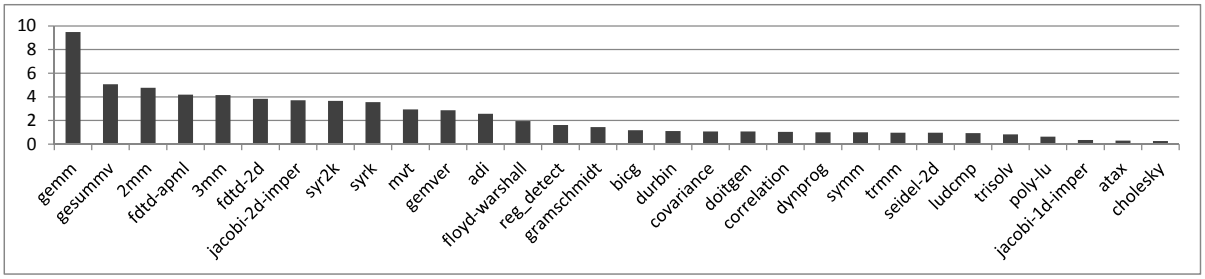

Fig. 6. Speedup on 4 threads for the Polybench benchmark suite

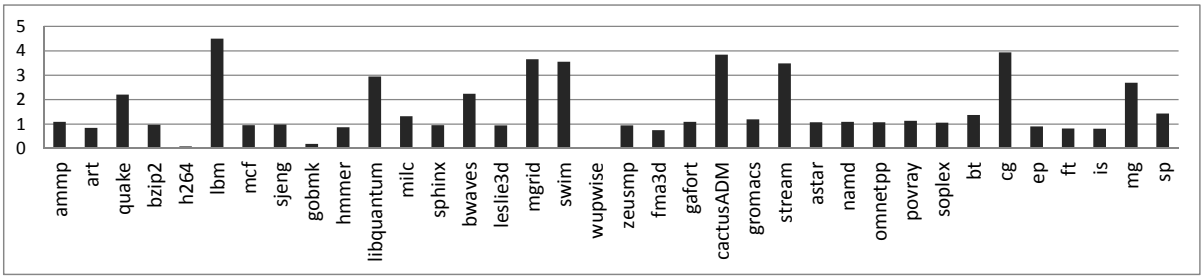

Fig. 7. Speedup on 4 threads for the SPEC 2006, OMP 2001 and NAS benchmark suites