Iterative Compilation and Performance Prediction

Grigori Fursin

G.Fursin@ed.ac.uk

Abstract: This poster describes a new platform and language independent automatic iterative optimisation approach for improving a program's memory performance on rapidly evolving hardware, and presents a new performance prediction technique for minimizing iterative compilation time, considerably outperforming native commercial compilers within a reasonable number of iterations.

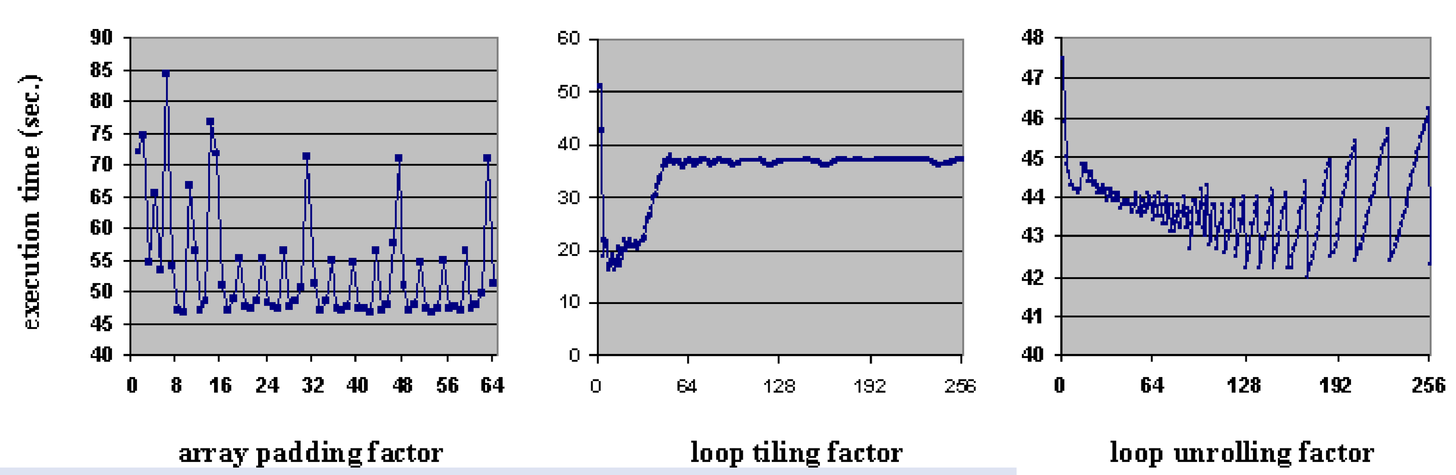
Problem

As the current rate of improvement in processor performance far exceeds the rate of improvement in memory performance, memory latency can be the dominant overhead in many performance critical applications. Often, automatic compiler-based optimisations to improve memory performance are very limited. Furthermore, a diverse range of a rapidly evolving hardware makes the optimisation process even more complicated.

Basic iterative compilation

This research presents a platform independent optimisation approach based on iterative feedback-directed program restructuring using just three transformations available in most of the recent commercial compilers: array padding, loop tiling and loop unrolling.

The following figure shows the change in execution time of the MatMul kernel (matrix multiplication) on Alpha platform after applying the above transformations. The oscillatory nature of these graphs with many local minimums explains the problems which compilers face when trying to find the best program transformation statically.



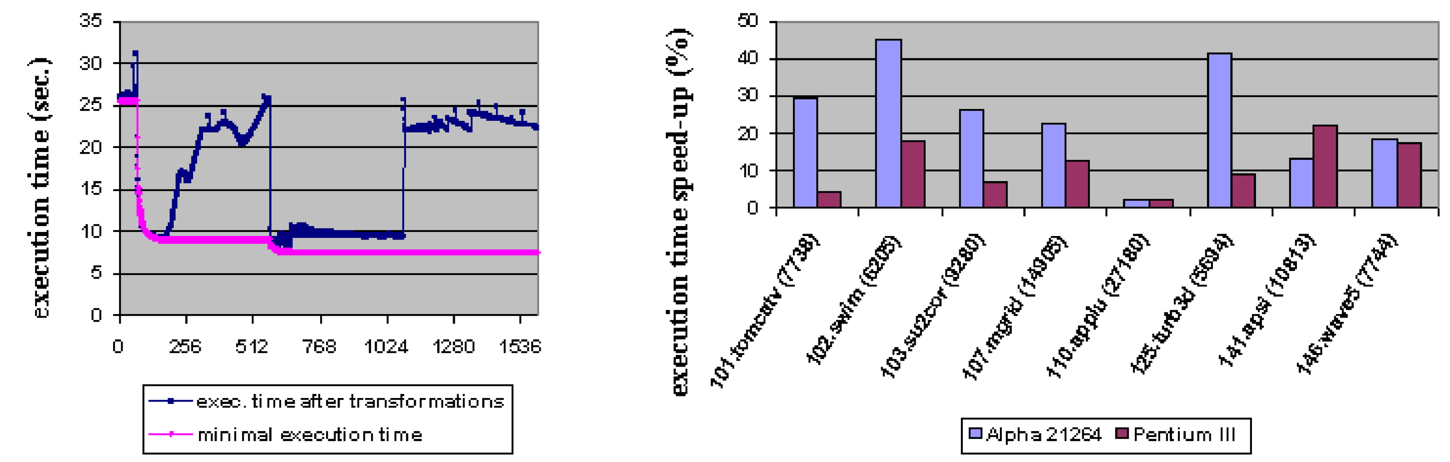
MatMul execution time after applying padding, tiling and unrolling

The new proposed algorithm for the automatic iterative compilation which overcomes the above problems is presented below:

1. *profile original program*
2. *choose set of arrays and loops*
3. *apply data transformations:*
   - *apply array padding (1..Na) for all global arrays and run new code each time*
   - *choose the best transformation (minimal execution time)*
4. *apply loop transformations:*
   - *if loop is perfectly nested:*
     - *apply loop tiling (1..Nt) for the loop nest and run new code each time*
     - *choose the best transformation*
     - *apply loop unrolling (1..Nu) over loop tiling (if found) for innermost loop and run new code each time*
     - *choose the best transformation*
   - *if loop is single nested:*
     - *apply loop unrolling (1..Nu) and run new code each time*
     - *choose the best transformation*
5. *finish: code should now contain a sequence of data and loop transformations that minimise its execution time*

This algorithm searches through all allowed transformation factors, runs the transformed code in each case and compares the execution time with the best achieved to choose the best transformation. More details about algorithm and implementation are available from our paper {1}.

As the number of iterations increases, the execution time of the optimised program approaches its optimum as shown in the figure below for the MatMul kernel. The developed iterative search was evaluated using 8 floating point SPEC benchmarks on Alpha 21264 and Pentium III platforms. The following figure also presents the execution time speed-ups after applying our technique in comparison with the best execution time obtained using native commercial compilers with both static and profile-directed options.



MatMul execution time reduction with the number of iterations and SPEC benchmark's speed-ups after iterative compilation

The figure above shows that, in most of the cases, using the new optimisation approach gives a considerable performance improvement on both platforms in comparison with the native compilers. However, the number of iterations needed to achieve such speed-ups (shown in brackets) is too high. The following section will introduce a new technique to reduce this number.

Performance prediction

Since iterative compilation is very time consuming, the following question arise: is it possible to estimate the potential benefit from program transformations before applying costly iterative compilation?

The main existing methods are based on using platform simulators or hardware counters. However, simulators are very slow and hardware counters give wrong prediction on current processors with out-of-order execution (more details are available in our paper {2}).

This research introduces a reasonably fast and accurate technique to determine the lower bound on execution time of a program, which is defined as a program with no cache misses, and obtained by by modifying all assembler instructions with memory access in such a way that they refer to constant addresses.

```
DO I=1, N
   ... = ... A[I] ...
   ...
END DO
```

```
...
lda $19,   8($19)
ldt $f13,   ($19)
...
```

```
...
lda $19,        8($19)
ldt $f13, Const($28)
...
```

Example of the performance prediction transformation (on Alpha assembler)

In this case all the instructions are the same; the same number of computations are performed, but now the address referenced by each instruction is constant over the whole loop execution. Consequently, the memory footprint fits the cache and all cache misses are removed. More details about the algorithm, implementation and miscellaneous issues are available from our paper {2}.

Comparing the execution time of the original loops and loops without cache misses can provide information about whether these loops should be included into the iterative search or not: if there is a considerable difference then this loop can possibly benefit from transformations, otherwise it is excluded from the iterative search.

Random iterative compilation

Another way of reducing the number of iterations is by using random search, i.e. the basic algorithm will be the same as above, but instead of trying all possible transformation factors (Na, Nt and Nu), only N (5 in our experiments) random factors will be tried. This dramatically reduces the search space from thousands of iterations to just tens with a similar speed-up (1...3% worse than speed-ups achived by long iterative search).

The following table demonstrates the optimisations results of MatMul kernel on the Pentium III platform using performance prediction and random iterative compilation:

	time:
original time	83.9 s.
predicted time	6.2 s.
time after basic iterative compilation	7.6 s. (2110 iterations)
time after random iterative compilation	8.1 s. (20 iterations)
time for highly tuned Intel library	6.8 s. (months of research)

This table shows that the execution time of the MatMul kernel approaches the predicted time after just 20 iterations when using the automatic random iterative compilation technique and is close to the time obtained when using the basic search with 2110 iterations. Moreover it approaches the time for the same kernel from the Intel library which is usually tuned manually on assembler level for each new platform, while our approach can be applied automatically regardless of the platform.

Conclusions

The combination of the new performance prediction technique and the random iterative compilation considerably ouperforms native commercial compilers (up to 45% after only 20..90 iterations) Besides, the developed algorithms are platform and language independent which allows programs to be automatically optimised on a diverse range of a rapidly evolving hardware.

References

1. G.G.Fursin, M.F.P.O'Boyle, P.M.W.Knijnenburg; "Evaluating Iterative Compilation"; Proc. Languages and Compilers for Parallel Computers (LCPC'02); Springer Lecture Notes in Computer Science, 2002
2. Grigori Fursin, Mike O'Boyle, Olivier Temam, Gregory Watts; "Fast and Accurate Method for Determining a Lower Bound on Execution Time"; Journal 'Concurrency Practice and Experience 2002'

THE UNIVERSITY OF EDINBURGH

Division of informatics