Computer Languages, Systems & Structures 41 (2015) 21–41

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

# A methodology for speeding up loop kernels by exploiting the software information and the memory architecture

Vasilios Kelefouras [a,*], Angeliki Kritikakou [b], Costas Goutis [a]

[a] Department of Electrical and Computer Engineering, University of Patras, Greece
[b] Education and Research Department in Computer Science and Electrical Engineering, University of Rennes 1, France

ARTICLE INFO

*Article history:*
Received 21 July 2014
Received in revised form
15 December 2014
Accepted 18 January 2015
Available online 3 February 2015

*Keywords:*
Data reuse
Register allocation
Optimization
Memory hierarchy
Loop tiling
Data locality
Diophantine equations

ABSTRACT

It is well-known that today's compilers and state of the art libraries have three major drawbacks. First, the compiler sub-problems are optimized separately; this is not efficient because the separate sub-problems optimization gives a different schedule for each sub-problem and these schedules cannot coexist as the refining of one, causes the degradation of another. Second, they take into account only part of the specific algorithm's information. Third, they take into account only a few hardware architecture parameters. These approaches cannot give an optimal solution.

In this paper, a new methodology/pre-compiler is introduced, which speeds up loop kernels, by overcoming the above problems. This methodology solves four of the major scheduling sub-problems, together as one problem and not separately; these are the sub-problems of finding the schedules with the minimum numbers of (i) L1 data cache accesses, (ii) L2 data cache accesses, (iii) main memory data accesses, (iv) addressing instructions. First, the exploration space (possible solutions) is found according to the algorithm's information, e.g. array subscripts. Then, the exploration space is decreased by orders of magnitude, by applying constraint propagation to the software and hardware parameters.

We take the C-code and the memory architecture parameters as input and we automatically produce a new faster C-code; this code cannot be obtained by applying the existing compiler transformations to the original code. The proposed methodology has been evaluated for five well-known algorithms in both general and embedded processors; it is compared with gcc and clang compilers and also with iterative compilation.



## 1. Introduction

Regarding data dominant applications (for example linear algebra, image, signal and video processing algorithms), the major performance critical parameters are (i) the number of main memory accesses, (ii) the number of L3/L2 cache accesses, (iii) the number of L1 data cache accesses and (iv) the number of executed instructions (we assume that the number of the algorithm instructions cannot be reduced and thus we reduce only the number of addressing instructions). The above compilation/scheduling sub-problems are interdependent and thus they cannot be optimized separately; actually, the refining of one sub-problem causes the degradation of another, e.g. a decrease of the number of L2 data cache accesses will

* Corresponding author.
*E-mail address:* kelefouras@ece.upatras.gr (V. Kelefouras).

http://dx.doi.org/10.1016/j.cl.2015.01.003
1477-8424/

consequently increase the number of L1 data cache accesses. Researchers try to solve this problem by using iterative compilation techniques.

Iterative compilation has five major drawbacks, (i) there are memory efficient schedules which cannot be produced by applying the existing compiler transformations, (ii) iterative compilation does not use all the existing transformations, including all the different transformation parameters, e.g. unroll factor values and tile sizes, because in this case compilation will last for years, (iii) only one level of tiling is applied, which is not efficient, (iv) register allocation is applied without taking into account the data reuse; this means that the arrays references are assigned into registers, without taking into account that some are accessed a lot and others do not, (v) the data array layouts are not taken into account; we will show that when tiling to multidimensional arrays is applied, the data array layouts must change. These drawbacks are overcome by the proposed methodology.

The proposed methodology finds the exploration space (all possible solutions), neither by applying compiler transformations nor by utilizing the above sub-problems separately. Instead, the exploration space is produced by exploiting the algorithm's information; we create mathematical equations and inequalities, according to the array subscripts, the loops iterators and the loops bounds. These equations (subscript equations) give the data reuse and the production-consumption of the arrays; the memory access pattern of each array reference is given by its subscript equation. Given that the memory access pattern of each array is given by its subscript equation, we claim that all memory efficient solutions (exploration space) can be produced by processing these equations. The subscript equations are processed and a new iteration space is created. Each subscript equation gives either its iterators or even new iterators, to the new iteration space. Then, the exploration space is orders of magnitude decreased by applying constraint propagation to the software and hardware parameters. Regarding the hardware parameters, we produce register file and data cache inequalities, which contain all the (near)-optimum tile sizes; these inequalities contain (i) the tiles sizes in elements, (ii) the shape of each array's tile. Furthermore, new data array layouts are generated, according to the data cache associativity. All the schedules with different tile sizes and data array layouts, than these the proposed methodology gives, are not considered, decreasing the exploration space.

The major contributions of this paper are: (i) the optimization of the above subproblems as one problem and not separately for a wide range of algorithms and computer architectures, (ii) the software information and several hardware parameters are fully exploited giving high execution speed solutions and a smaller search space, (iii) the proposed methodology, due to the major contribution of number (ii) above, gives a smaller code size and a smaller compilation time, as it does not test a large number of alternative schedules, as the state of the art (SOA) libraries and iterative compilation do.

The experimental results are taken by using a general purpose processor, an embedded processor and Simplescalar simulator [1]. The proposed methodology is evaluated for five well-known data dominant algorithms over two different compilers (speedup from 1.8 up to 18.3) and iterative compilation technique (speedup up to 2.2).

The remainder of this paper is organized as follows. In Section 2, the related work is given. The proposed methodology is given in Section 3 while the experimental results are given in Section 4. Finally, Section 5 is dedicated to conclusions.

## 2. Related work

The independent optimization of the back end compiler phases (e.g. transformations, register allocation) leads to inefficient binary code due to the dependencies among them. These dependencies require that all phases should be optimized together as one problem and not separately. Toward this, much research has been done, either to simultaneously optimize only two phases, e.g. register allocation and instruction scheduling [2,3] or to apply predictive heuristics [4,5]. Nowadays compilers and related works, apply (i) iterative compilation techniques [6–9], (ii) both iterative compilation and machine learning compilation techniques to restrict the configurations' search space and thus to decrease the compilation time [10–15], (iii) iterative optimizations or compiler transformations, by using the Polyhedral model [16–19], (iv) compiler transformations by using heuristics and empirical methods [20]. In iterative compilation, a large number of different versions of the program are generated-executed by applying many compiler transformations, at all different combinations. Iterative compilation requires extremely long compilation times – to decrease the exploration space iterative compilation is applied with machine learning compilation techniques. The five major iterative compilation drawbacks are referred to the introduction. The proposed methodology achieves up to 2.1 times lower execution time and an order of magnitude lower compilation time (Section 4).

The state of the art software libraries, such as ATLAS [21], GotoBLAS2 [22], Eigen [23], Intel_MKL [24], PHiPAC [25], FFTW [26], OpenCV [27] and SPIRAL [28], manage to find a near-optimum binary code for a specific application by using a large exploration space (many different executables are tested and the fastest is picked). Although they achieve high speed, they are application specific and the final schedule is found mostly by using heuristics and empirical techniques. A comparison with the above libraries would be unfair because they use the SIMD (Single Instruction Multiple Data) vector instructions (they support load/store and arithmetical instructions with 128/256-bit data); however, our future work includes the support of SIMD instructions. In [29–32], we have developed algorithm specific methodologies (we used the SIMD instructions), which produce lower execution time, lower compilation time and lower number of data accesses, than ATLAS [29,30], FFTW [30] and OpenCV [32]. A comparison between the proposed methodology and [29,30] is made in Section 4.

Furthermore, many sub-optimum methods exploiting the memory hierarchy have been analyzed in the past, such as [33–38]. These works apply compiler transformations to the original code (this is not performance efficient). The cache