COMPUTER SCIENCE • 18(2) 2017 http://dx.doi.org/10.7494/csci.2017.18.2.145

DOMINIK ADAMSKI
GRZEGORZ JABŁOŃSKI

HARDWARE-AWARE TILING OPTIMIZATION FOR MULTI-CORE SYSTEMS

Abstract *This paper presents a proposal for a new tool that improves tiling efficiency for a given hardware architecture. This article also describes the correlation between the changing hardware architecture and methods of software optimization. The first chapter includes a short description of the change in hardware architecture that has occurred over the past ten years. The second chapter provides an overview of the tools that will be used in further research. The subsequent sections contain a description of the proposed hardware-aware tool for optimal tiling.*

Keywords LLVM, tiling, data locality, polyhedral model

Citation Computer Science 18(2) 2017: 145–162

1. Introduction

For many decades, the speedup of program execution has been achieved through the speedup of processor clocks. The rapid growth of processor clock frequencies caused a relatively small change in the architecture of processors. Rapid growth of processor frequency stopped in 2005 due to heat dissipation issues. Since that time, hardware manufacturers have shifted towards multi-core architectures. They introduced advanced multi-level cache memory systems and multiplied the number of processor cores in a single CPU unit. These changes are reflected in a more-sophisticated processor architecture and increasing number of transistors used inside a single CPU.

Unfortunately, the shift in hardware architecture does not provide an automatic speedup of software that has been optimized for sequential processing. Consequently, any new optimization method should take into account the new target hardware architectures. Modern compilers should support parallel task execution, and they should provide new optimization methods that would automatically detect parallel regions and optimize them in order to fully utilize the hardware resources. There is a strong need to develop such techniques of code optimization that could be easily deployed on various hardware architectures on one hand and take into account the many specific hardware features that are different for every target platform on the other. Optimization methods that meet these goals can be easily deployed in various areas of the computer industry. They can be applied to mobile devices, where they can reduce power consumption and prolong battery life. More-effective software for data centers can reduce the cost of energy while also decreasing data-access time.

1.1. Memory optimization

The rapid increase of processor computation power has not been followed by a proportional memory speedup. As a consequence, the overall speed of program execution is limited by the memory latency [2]. Multilevel memory organization allows us to reduce the gap between memory and processor performance. Modern processors are equipped with a small amount of quick cache memory placed near the processor core and larger amount of slower cache shared by the many cores.

Unfortunately, no general model for cache memory organization has come along with the increased variety of processor architectures. GPU processors are characterized by multiple cores with a small amount of shared cache memory and a distributed memory model, while CPU processors use a uniform memory model with a large amount of multilevel cache memory. In general, memory usage optimization should aim to exploit the internal cache memory instead of calling data from the slow external RAM memory. The number of cache misses should be minimized as well as the number of memory transfesr between the respective memory units.

1.2. Data locality

Designers of processing units have introduced a multilevel system of memory organization to improve memory efficiency [10]. They decided to equip processing units with a small amount of fast cache memory. In modern CPUs, there are multiple levels of cache memory characterized by different sizes and speeds. The lower tiers of cache are the fastest, but their sizes are the smallest. Usually, they cooperate with only one core. The higher tiers of cache are often shared between multiple cores. Their sizes are bigger, but they are slower than the cache from the lower tier. If a given variable is used many times by a processor, it is placed in the lower cache. In such a case, the waiting time for data is reduced, allowing the processor to perform faster calculations. If the processor requests data that is not inside the cache memory, then a cache-miss event occurs. In such a case, the processor should wait until the data is transported from the RAM memory. This situation substantially reduces the performance of the processing units.

1.3. Tiling

One of the available techniques for improving memory performance is tiling [19]. The main aim of this optimization is to maximally reuse the fastest cache memory. This goal can be achieved by the division of large loop iteration space into smaller rectangular parts (tiles). Listing 1 illustrates this tiling optimization. The size of the tiles should be chosen in such a way that cache misses are minimized. It has been proven that tile size should be chosen in such a way that the number of cache misses is minimized for all levels of cache memory.

```
//input source code
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    A[i][j] = B[i][j] + C[i][j];

//optimized source code
for (int i =0; i < N; i+=T1)
  for (int j = 0; j< N; j+=T2)
    for (int ii = i; ii < min(i+T1,N); ++ii)
      for (int jj = j; jj < min (j+T2,N); ++jj)
                A[ii][jj] = B[ii][jj] + C[ii][jj];
```

There are many factors that should be taken into account while choosing the optimal tile size. This is largely dependent on the target hardware platform. A given tile size can provide a speedup of calculations for one target while the same tiling configuration can cause a significant slowdown for another target platform. On the other hand, the optimal tile size depends on the iteration space and memory access patterns that are defined by the developer. In the authors' opinion, it is also not possible to determine an accurate analytical model for optimal tiling prediction because of the complexity of hardware systems and the difficulties with the static analysis of input source code that should be optimized.

2. State of the art

Currently, code optimization for multicore architectures is at the center of interest for many research teams and large companies. They try to develop tools that would fully utilize the computational power of their multicore systems. Their research effort is focused on tools for input code analysis. They have also proposed new techniques for code optimization. These techniques include the automatic parallelization of input code and a reduction in cache misses.

2.1. Polyhedral model

Nowadays, major compilers like GCC, LLVM, ICC, and MSVC are equipped with tools for detection loops that can be parallelized. ICC and MSVC compilers are commercial products, and their sources are not publicly available. For this reason, it is not possible to accurately assess the advantages and drawbacks of algorithms implemented in these products. GCC and LLVM compilers are open-source, and there are some projects (like Graphite for GCC and Polly for LLVM) that use a mathematical concept – polyhedrons for detecting parallel regions of input code.

The main idea of the polyhedral model is to describe loops and loop bodies in terms of mathematical equations [1, 20]. Loop boundary conditions are modeled as linear functions that limit iteration space. The dimension of iteration space is equal to the number of nested loops. All data accesses inside of a loop body are described in terms of iteration space coordinates. This mathematical model is used by optimizers who are trying to find the best schedule for a given loop.

Tools for automatic code parallelization provide analytical methods for detecting whether a given set of loops can be executed in parallel. This information is important for finding an optimal loop tiling schedule through a broadening of the search space. It is possible to reorder loops in a parallel region to increase data locality. Such a transformation can simplify tiling analysis; as a consequence, the most appropriate tiling size can be found faster. The Polly compiler is one of the tools for automatic parallelization that can reorder a sequence of parallel loops for improved data locality [8]. It also supports fixed tile size optimization, but such an optimization is not always profitable. Unfortunately, Polly optimizations do not always lead to more-effective software. For some cases, tiling optimization decreases the speed of execution of the programs [6].

2.2. Analytical approach

The analytical approach for finding optimal tile size is based on analysis of the input source code and target hardware. Section B of Figure 1 illustrates this method of optimization. During the compilation process, the compiler should decide how to tile the loops so that the number of cache misses is minimized. The problem of analytically finding the best partition of data in the general case for a multilevel system of cache memory is classified as NP hard [19]. It is not possible to determine in finite time how to place program data into the computer memory so that the time necessary for data transport is minimal. The main difficulty lies in number of combinations that should be analyzed. Therefore, analytical models only cover some special cases for which it is possible to determine the optimal data schedule.

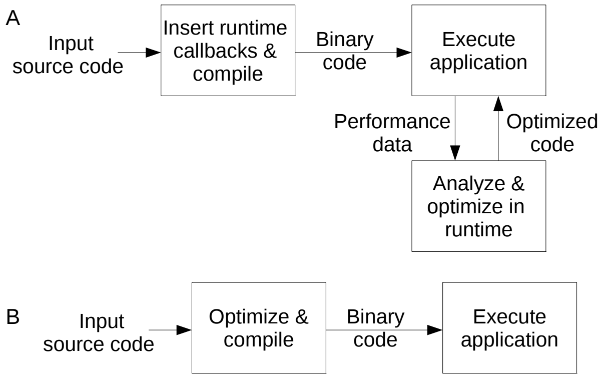

Figure 1. Scheme of statistical (A) and analytical approach of optimization (B).

Analytical models can be divided into two subcategories. The first subcategory contains all models that predict optimal tile size for strictly defined input source code patterns. They try to match the input code with those given loop patterns for which it is possible to find an optimal tile size [5]. The second category is based on some heuristic simplification. The hardware is modeled in a simplified way. Such simplification allows us to find the suboptimal tile size. The range of simplification is strictly combined with the quality of optimization. More-general models find suboptimal tile results faster. For these models, there is a large risk of obtaining poor optimization results. On the other hand, more-sophisticated models require increased computation, and the time for finding suboptimal results may be unacceptably long.

2.3. Statistical approach

Statistic methods for finding suboptimal tile sizes have also been proposed [15, 17]. They try to predict the optimal tile size on the basis of previous results of execution of an optimized loop. Section A of Figure 1 illustrates this approach. This proposition requires a special runtime that gathers information about previous tile sizes and their corresponding execution times. Every time an optimized loop is executed, the runtime tries to provide the most-effective tile size. This approach does not include any theoretical models, and it may require many invocations of tiled loops to find the optimal tile size.

2.4. Other approaches

Some researchers propose shapes of tiling other than rectangular. Grosser et al. proposed a hexagonal shape of tiles for GPU code [7]. Another approach was introduced by Kong et al; they proposed that the minimization of cache misses can be achieved not by data tiling but by dynamic dataflow parallelization [12].

3. Base tools for hardware-aware tiling

As mentioned in the previous section, there are already some tools that try to optimize the data locality in loops. They exist as separate tools, and each of these tools has its strong and weak points. In the authors' opinion, it is worth combining all of these methods into one tool. This new tool should be based on a Polly compiler, its runtime should measure cache misses by the PAPI library, and it should be tested on the Polybench benchmark.

3.1. LLVM framework

The LLVM project was started as an academic tool for multi-stage optimization [14]. Nowadays, it is one of the leading open-source compiler projects. It is entirely written in C++ and characterized by a modular design. It also provides a well-documented API. These features have made LLVM an oft-chosen framework for many compiler projects. Figure 2 presents the internal relationship between LLVM modules.

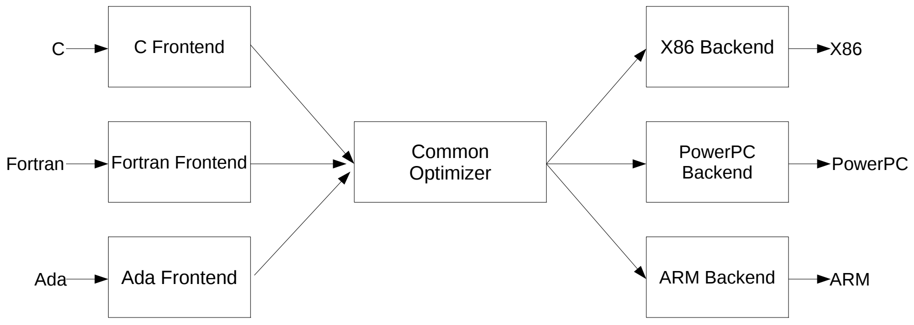

Figure 2. LLVM architecture [13].

Front-end modules are responsible for converting input source code into a simpler form for analysis intermediate (IR) code. IR code is independent from high-level input language. The simplified syntax of IR code helps make data and control flow analysis easier as compared to source code analysis. It is used for hardware independent optimization. All types of IR optimization are executed sequentially. The order of execution is determined by Pass Manager, which analyzes the dependencies between passes. Optimized IR code is transferred to backend modules that generate target-specific binary code.

3.2. Polly compiler

The Polly compiler is a project based on the LLVM framework. This compiler describes loops in terms of mathematical equations; if it detects that some part of the code can be parallelized, then it uses the simplex method for finding the best schedule and then generates a parallelized code [8].

The Polly compiler automatically detects regions of IR code that can be parallelized. The code that is ready for parallelization must satisfy the following conditions:

- The number of loop iterations can be calculated during compilation.
- The result of calculations is independent from the order of loop execution.
- There should be no side-effects inside the input code.

These conditions allow the compiler to freely rearrange the order of statement execution. Such a rearrangement is necessary for tiling optimization. If a loop is parallelizable, then loop tiling optimization can be safely performed.

Each detected parallel region is described by the Static Control Part (SCoP) object in Polly. These objects define the iteration space of parallel loops, memory access patterns inside the loops, and data dependency between elements of the loops. This information is used as the input for polyhedral optimizers that calculate an optimal schedule for a given SCoP. Figure 3 presents the described architecture of Polly.

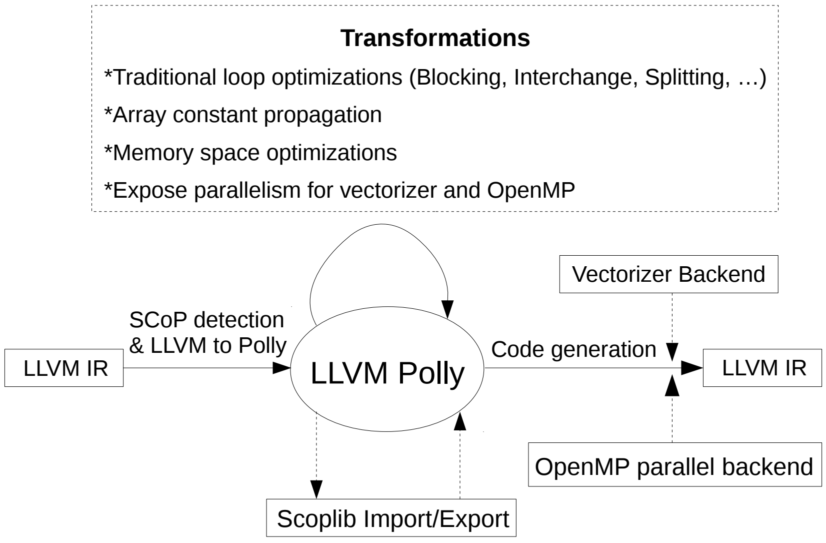
Transformations
*Traditional loop optimizations (Blocking, Interchange, Splitting, ...)
*Array constant propagation
*Memory space optimizations
*Expose parallelism for vectorizer and OpenMP
Vectorizer Backend
SCoP detection & LLVM to Polly
LLVM IR
LLVM Polly
Code generation
LLVM IR
OpenMP parallel backend
Scoplib Import/Export

Figure 3. Architecture of Polly compiler [9].

3.3. PAPI library

The PAPI library provides tools for the accurate measurement of optimized loops [4]. It provides an interface for gathering information about the actual number of cache misses and time of loop execution. This data plays an important role in the assessment of the quality of optimized loops. If the tile size is badly chosen, then the number of cache misses will be high.

3.4. Polybench

Polybench is a set of benchmarks with parallel kernels [16]. These kernels correspond to popular matrix operations like matrix multiplication, the Fourier transform, matrix correlation, or decomposition. Polybench source code will be used as the reference benchmark for the proposed approach for finding optimal tile sizes.

4. Proposed solution

It should be noticed that both the statistical and analytical approaches have some drawbacks. Theoretical considerations about optimal tile size cannot give an exact answer on which tile size is the best, and the statistical approach requires multiple execution of an optimized loop, and this method does not always provide a speedup in loop execution. On the other hand, the polyhedral analysis used for finding an optimal loop schedule for the Polly compiler can be time- and memory-consuming, and it does not always provide the best result.

In the authors' opinion, it is worth combining the tools from static loop optimization with those from dynamic tile selection. The Polly compiler will be used for the static analysis of input source code. It will detect the ready-for-parallelization regions of the IR code, and it can propose a new schedule of loop statements. All optimizations made by Polly are described by the SCoP object, which contains important data about memory access patterns, iteration space, data dependency, and the proposed loop schedule. This information will be saved in output binary code and will be read by runtime functions that are responsible for choosing the proper tile size. The choice of optimal tile size should be based on the heuristic data gathered from previous executions and analytical data from the static code analysis. Figure 4 illustrates the proposed solution.

The main aim of such an approach is to provide more data to the tile selection mechanism. In the authors' opinion, it is the only way to combine static and dynamic analysis results. It is expected that such a combination will give a more-accurate model that will properly estimate the optimal size of the loop tiling. The proposed tile-size-prediction-method algorithm will not limit any other polyhedral optimization. Tiled code can be still parallelized or vectorized. The described optimization method works on IR code so it can be combined with the machine-specific optimization made by the target specific compiler backend.

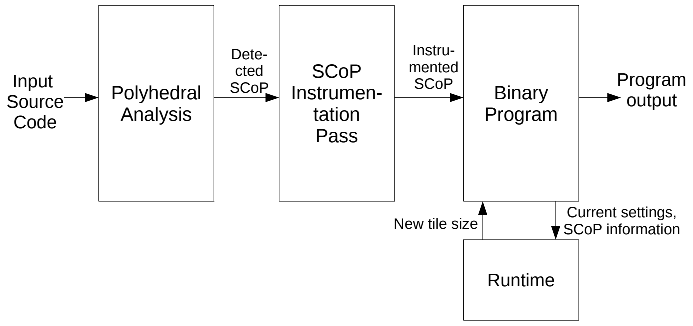

Figure 4. Architecture of proposed solution.

The runtime algorithm used to calculate the optimal tile size has not been specified. In the authors' opinion, it should be done in the last phase of research. First, the mechanism for saving static analysis results in the output binary code should be implemented. Without this mechanism, it is not possible to check which artificial intelligence algorithm predicts the optimal tile size in the best way.

The proposed approach allows us to shorten selection time by adding the more-detailed code description obtained by the polyhedral analysis into the dynamic runtime selection algorithm. It is expected that the compile time will be remain the same as for the standard polyhedral optimization. Runtime overhead can increase during program execution (as compared to simple heuristic models) because more analysis should be done for choosing the best tile size. On the other hand, a more-sophisticated method of finding the best tile size could reduce the number of code executions needed to find the optimal tile size.

This project also requires some changes in the LLVM code. A new pass should be added that will automatically insert runtime callbacks into the optimized loops. These callbacks should automatically adjust the tile size based on the previous optimization results and information about the hardware.

4.1. Target platforms

Today, hardware manufacturers offer multiple solutions for calculation acceleration. Their architecture is different, and it is worth providing a general approach for finding the approximate optimal tile size. Currently, there are three main trends in the design of computing efficient systems. Each of them is different, and runtime should ask for specific values of parameters for each platform separately.

Intel has proposed a new concept of processor architecture; it is called the Many Integrated Core (MIC) architecture. This is characterized by multiple general-purpose processors that share cache memory. In comparison to Haswell, some cores play the role of coprocessor. Their role is flexible and can work in many configurations [11]. The type of operation mode depends on the type of processed algorithm. If the algorithm is easily parallelizable, then the host processors can offload a portion of the calculations to the coprocessors. If the code cannot be executed concurrently, then only one host processor should work. Tiling runtime should take into account how many coprocessors are available, how the workload should be divided by the cores, and which mode of of operation is most-suitable.

GPU systems are characterized by distributed memory systems. The host processor can offload calculations to the GPU. The offloading procedure requires a data transfer between the CPU and GPU memory. This transfer strongly affects the speed of the calculations. Moreover, the processing units on the GPU are optimized for stream processing. As a consequence, the execution of the branch instruction takes more time than for the CPU. The tiling runtime should take into account the number of threads available on the target GPU. The best tile size should effectively minimize the number of branches, and it should allow as many threads as possible to execute the calculations in parallel.

The third target platform is the combination of a traditional CPU processor with a Field Programmable Gate Array (FPGA) device such as Xilinx ZynQ. This approach allows us to offload calculations to a device that can be easily tailored to the end user's computational needs. Recent research shows that tiling can improve usage of the available hardware [3]. The runtime selecting the best tile size for the FPGA device should take into account not only the results of code analysis but also the available resources (number of available gates, memory space, and memory bandwidth). Due to the long time necessary to program an FPGA device, it would probably be impossible to tune the tile size during runtime. For this platform, runtime can only gather execution data (like the size of the used resources or time of kernel execution), and it should propose the best tile size when the kernel code is once again recompiled to bitstream code and then loaded into the FPGA device.

5. Proof of concept

This section presents the proof of concept results. The methodology was as follows: a function (presented in the listing below) was manually tiled. For each tested tile size, the time of function execution and number of data cache misses was recorded by the PAPI functions. They were inserted into the beginning and at the end of the `test_function` code.

```
void test_function (int *x1, int *x2,
        int *A[], int *y1, int *y2, int _PB_N) {
  for (i = 0; i < _PB_N; i++)
    for (j = 0; j < _PB_N; j++)
      x1[i] = x1[i] + A[j][i] * y_1[j];

  for (i = 0; i < _PB_N; i++)
    for (j = 0; j < _PB_N; j++)
      x2[i] = x2[i] + A[j][i] * y_2[j];
}
```

The Polly compiler detects that both loops can be described as one SCoP. As a consequence, it is possible to freely interchange the loop order. There are two variants of tiling optimization examined; the first concerns the tiling of each loop separately:

```
void test_function (int *x1, int *x2,
        int *A[], int *y_1, int *y_2,
        int _PB_N, int _TILE_I, int _TILE_J) {

  // _TILE_I and _TILE_J define tile size
 for (i = 0; i < _PB_N; i+=_TILE_I)
   for (j = 0; j < _PB_N; j+=_TILE_J)
     for (ii=i; ii < MIN(i+_TILE_I, _PB_N);ii++)
       for (jj = j; jj < MIN(j + _TILE_J, _PB_N); jj++)
         x1[ii] = x1[ii] + A[ii][jj] * y_1[jj];

 for (i = 0; i < _PB_N; i+=_TILE_I)
   for (j = 0; j < _PB_N; j+=_TILE_J)
     for (ii=i; ii < MIN(i+_TILE_I, _PB_N);ii++)
       for (jj = j; jj < MIN(j + _TILE_J, _PB_N); jj++)
         x2[ii] = x2[ii] + A[ii][jj] * y_2[jj];
}
```

The Polly compiler can propose that these statements:

```
x1[ii] = x1[ii] + A[ii][jj] * y_1[jj];
```

and

```
x2[ii] = x2[ii] + A[ii][jj] * y_2[jj];
```

can be combined into one loop. For this reason, the following tiling schedule was also analyzed:

```
void test_function (int *x1, int *x2,
        int *A[], int *y_1, int *y_2,
        int _PB_N, int _TILE_I, int _TILE_J) {

  // _TILE_I and _TILE_J define tile size
 for (i = 0; i < _PB_N; i+=_TILE_I)
   for (j = 0; j < _PB_N; j+=_TILE_J)
     for (ii=i; ii < MIN(i+_TILE_I, _PB_N);ii++)
       for (jj = j; jj < MIN(j + _TILE_J, _PB_N); jj++) {
         x1[ii] = x1[ii] + A[ii][jj] * y_1[jj];
         x2[ii] = x2[ii] + A[ii][jj] * y_2[jj];
       }
}
```

5.1. Hardware platforms

The code was compiled by gcc 4.8 and executed without any parallel optimization on two platforms. The first one was an Intel i7-2600K. This CPU has 8 MB of 3-level data cache. This PC is equipped with 8 GB of DDR3 RAM memory. The last level cache memory is shared between four cores. The first and second levels of cache memory are dedicated to one core. The second platform was an Intel Core 2 Duo t5500 with 2 GB of DDR2 RAM memory. This is a processor that was designed in 2007 for notebooks. It is equipped with 2 level 2 MB data cache memory. Both platforms run on Ubuntu 14.04.

5.2. Empirical results

This section includes the empirical results of tiling efficiency. The first group of plots shows the dependencies between data cache misses and tile size. The second group of plots shows the dependencies between time of kernel execution and tile size.

The performed tests show that tiling optimization is hardware dependent. Figures 6, 8, 10, and 12 show that there is wide range of tile sizes for which the time of execution is close to minimal. This can be explained by the fact that the number of cache misses is comparable for most tile sizes (Figures 5, 7, 9, 11).

Time of execution for Intel Core 2 Duo processor deteriorates when `tile_j` is equal to 2. For this case, the processor executes many branch instructions that cause a significant slowdown. If `tile_i` is equal to 2, then the number of cache misses is considerably high. This situation can be explained by the fact that chunks of arrays x1 and x2 cannot be correctly optimized.

The i7 processor better utilizes hardware resources. Figures 14, 16, 18, and 20 indicate that this processor can execute a kernel function in a smaller number of clock cycles. The higher performance and efficient cache memory cause that tiling optimization can significantly change the time of kernel execution. Comparison of the same kernels for the same problem size (for example, Figures 8 and 16) reveals that the number of cache misses is lower for the i7 processor. As a consequence, even a small increase in memory cache misses can cause a performance drop (see Figures 13, 15, 17, 19).

A comparison of two cases with the same memory access pattern that are executed on the same hardware (for example, fissioned kernel executed on the i7 processor: Figures 18 and 20) reveals that optimal tile size is more dependent on the memory access pattern than on the problem size. For both cases, the minimal time of execution is if `tile_j` is within range <4,32 >and tile_i is within range <4,32 >. Figures 21 and 22 indicate that, for these tile sizes, the number of L1 and L2 is minimal. This fact can be explained by an analysis of memory access patterns. Data access for the `x1` and `x2` arrays is optimized by `tile_i`. Meanwhile, the `y1` and `y2` arrays are optimized by `tile_j`. Array `A` is two-dimensional, and it is optimized by `tile_i` and `tile_j`. The presented figures indicate that the optimal tile size lies within the region where accesses for all arrays are optimized.

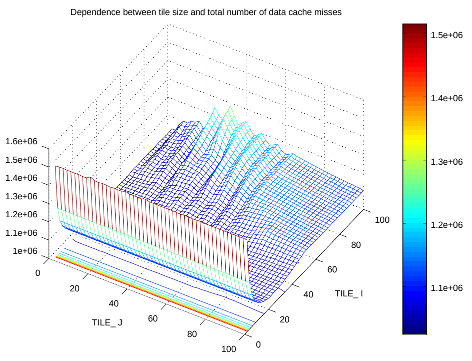

Figure 5. Data cache misses for L1 and L2 for Intel Core 2 Duo t5500 with _PB_N = 2000 and fused loops.

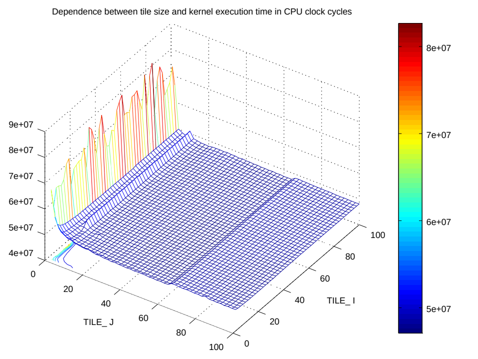

Figure 6. Time of kernel execution for Intel Core 2 Duo t5500 with _PB_N = 2000 and fused loops.

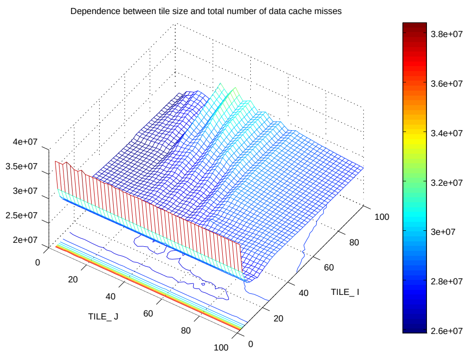

Figure 7. Data cache misses for L1 and L2 for Intel Core 2 Duo t5500 with _PB_N = 10,000 and fused loops.

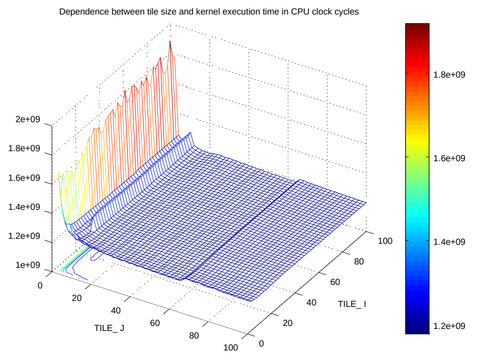

Figure 8. Time of kernel execution for Intel Core 2 Duo t5500 with _PB_N = 10000 and fused loops.

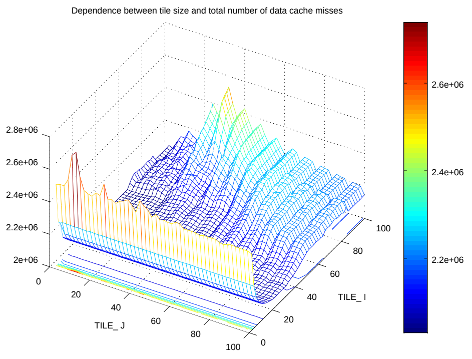

Figure 9. Data cache misses for L1 and L2 for Intel Core 2 Duo t5500 with _PB_N = 2000 and fissioned loops.

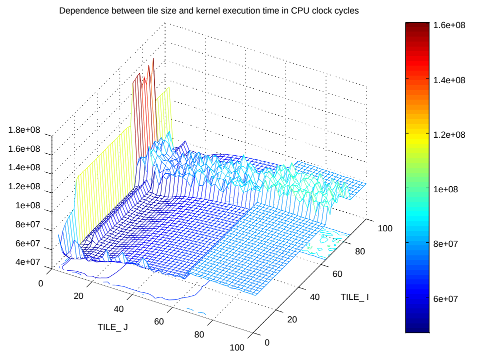

Figure 10. Time of kernel execution for Intel Core 2 Duo t5500 with _PB_N = 2000 and fissioned loops.

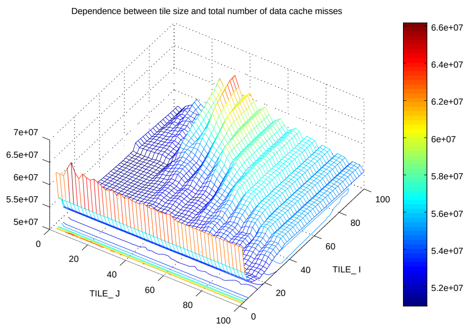

Figure 11. Data cache misses for L1 and L2 for Intel Core 2 Duo t5500 with _PB_N = 10,000 and fissioned loops.

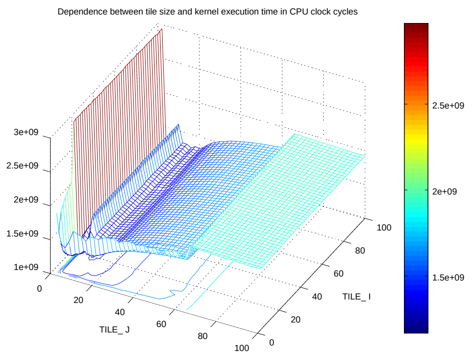

Figure 12. Time of kernel execution for Intel Core 2 Duo t5500 with _PB_N = 10,000 and fissioned loops.

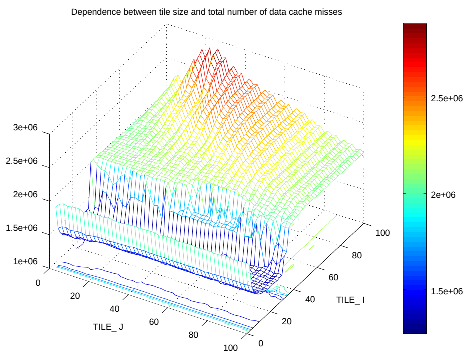

Figure 13. Data cache misses for L1 and L2 for Intel i7-2600K with _PB_N = 2000 and fused loops.

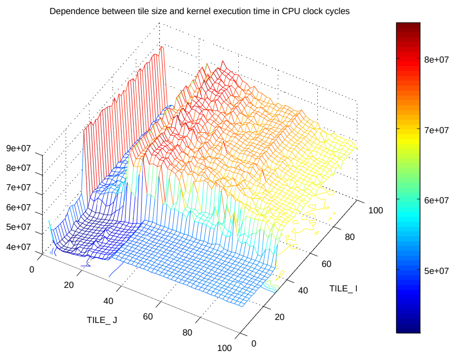

Figure 14. Time of kernel execution for Intel i7-2600K with _PB_N = 2000 and fused loops.

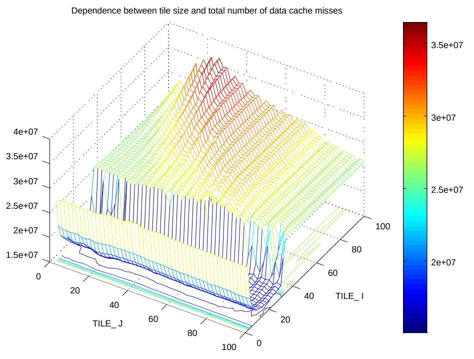

Figure 15. Data cache misses for L1 and L2 for Intel i7-2600K with _PB_N = 10,000 and fused loops.

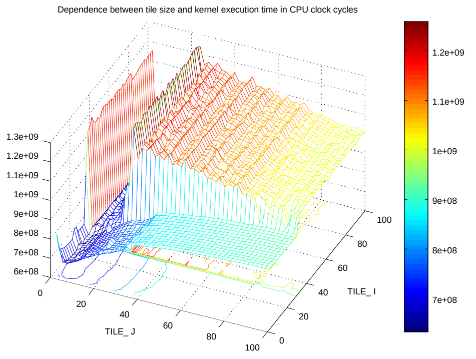

Figure 16. Time of kernel execution for Intel i7-2600K with _PB_N = 10,000 and fused loops.

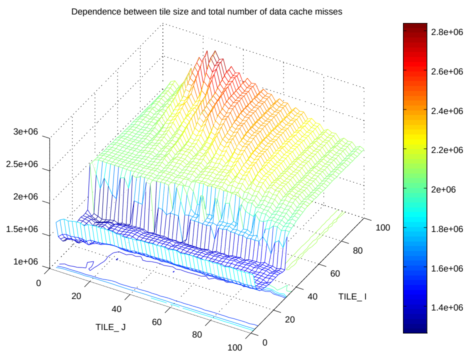

Figure 17. Data cache misses for L1 and L2 for Intel i7-2600K with _PB_N = 2000 and fissioned loops.

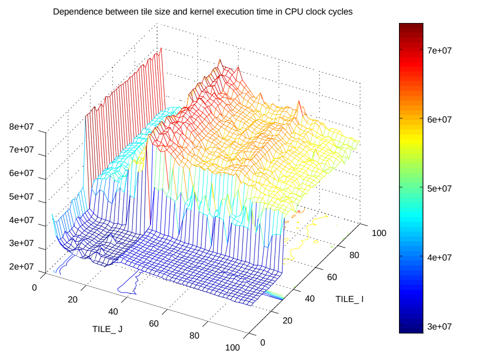

Figure 18. Time of kernel execution for Intel i7-2600K with _PB_N = 2000 and fissioned loops.

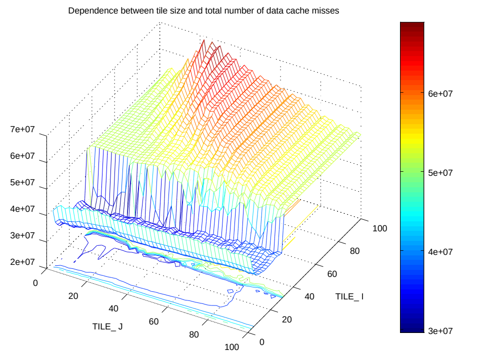

Figure 19. Data cache misses for L1 and L2 for Intel i7-2600K with _PB_N = 10,000 and fissioned loops.

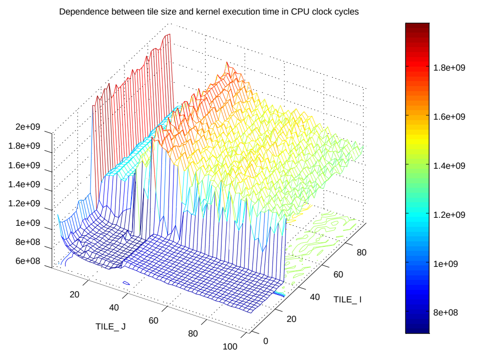

Figure 20. Time of kernel execution for Intel i7-2600K with _PB_N = 10,000 and fissioned loops.

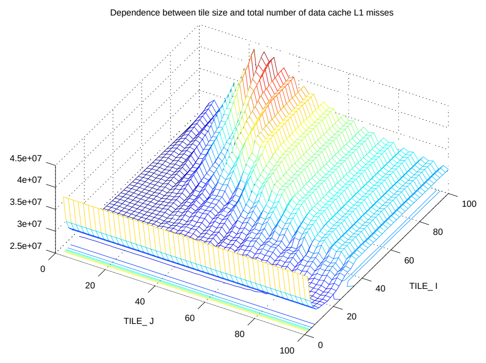

Figure 21. Number of data L1 cache misses for Intel i7-2600K with _PB_N = 10,000 and fissioned loops.

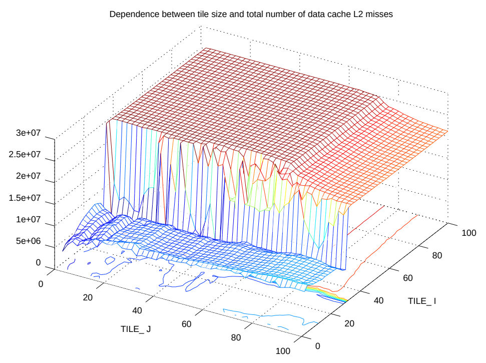

Figure 22. Number of data L2 cache misses for Intel i7-2600K with _PB_N = 10,000 and fissioned loops.

For both processors, the biggest bottleneck is memory speed. Time of data access is considerably longer than the time of loop branches. Fissioned loops are executed in a time comparable to the fused loops. If the tile size is wrongly chosen, then the time of execution of the fused loops can be longer than for the fissioned loops.

6. Conclusions

Tiling optimization can significantly reduce the number of data cache misses. The experimental results show that the efficiency of tiling is strongly dependent on memory access patterns for any given SCoP and hardware platform. Measurement data indicates that code analysis cannot be skipped in optimized tile size analysis.

Experimental results have shown that tiling optimization cannot be focused only on a single loop. Efficient tiling optimization should take into account the dependencies between neighboring and nested loops. SCoP analysis can indicate such regions of code. This analysis can state if it is safe to tile a given loop. It can also provide some information about the memory access patterns. This data can be used during runtime for quick and accurate suboptimal tile size prediction.

Measurements indicate that the efficiency of tiling is hardware-dependent. Execution of the same code on different hardware platforms causes the output results to be different. In general, tiling optimization can provide a higher gain for powerful processors. Figure 15 indicates that, for an Intel i7, tiling optimization reduces the number of cache misses by double. For the older platform, the dependency between tile size and number of cache misses is weaker. This conclusion should be used in runtime design. If it is possible to detect such a situation where the number of cache misses does not change radically for multiple tile sizes, then runtime should not spend much time on finding the most-optimal tile size. In such a case, the coarse result will be acceptable.

An efficient optimization procedure should take into account more parameters than the number of data cache misses or execution time. There is a strong need to define a holistic approach for efficient tiling optimization. This task is difficult because of the vast variety of hardware platforms. The dependencies between hardware specifications and executed software should be stated. It is vital to state which dependencies can be skipped during the optimization process. This simplification will lead to a reduction in time, which is necessary for the fine-tuning of the tiled loop.

References

[1] Bastoul C.: 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, pp. 7–16, IEEE Computer Society, Washington, DC, USA, 2004. `http://dx.doi.org/10.1109/PACT.2004.11`.

[2] Carvalho C.: The Gap between Processor and Memory Speed. In: *Proceedings of the Internal Conference on Computer Architecture*, pp. 27–34, 2002.

[3] Deest G., Estibals N., Yuki T., Derrien S., Rajopadhye S.: *Towards Scalable and Efficient FPGA Stencil Accelerators*, article presented during 6th International Workshop on Polyhedral Compilation Techniques 2016. `http://impact.gforge.inria.fr/impact2016/papers/impact2016-deest.pdf`.

[4] Dongarra J., Jagode H., Mucci P., Vaccaro P., YarKhan A.: PAPI Library. Project description available on webpage `http://icl.cs.utk.edu/papi/index.html`.

[5] Frigo M., Leiserson C.E., Prokop H., Ramachandran S.: Cache-Oblivious Algorithms. In: *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pp. 285–. IEEE Computer Society, Washington, DC, USA, 1999. `http://dl.acm.org/citation.cfm?id=795665.796479`.

[6] Grosser T.: Speedup of Polly tiling optimization in comparison to gcc -O3. Figure available on webpage `http://polly.llvm.org/performance.html`.

[7] Grosser T., Cohen A., Holewinski J., Sadayappan P., Verdoolaege S.: Hybrid Hexagonal/Classical Tiling for GPUs. In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pp. 66–75. ACM, New York, NY, USA, 2014. `http://doi.acm.org/10.1145/2544137.2544160`.

[8] Grosser T., Größlinger A., Lengauer C.: Polly – Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. In: *Parallel Processing Letters*, vol. 22(4), 2012. `http://dx.doi.org/10.1142/S0129626412500107`.

[9] Grosser T., Zheng H., Aloor R., Simbürger A., Größlinger A., Pouchet L.N.: Polly – Polyhedral optimization in LLVM. In: *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*. Chamonix, France, 2011.

[10] Hennessy J.L., Patterson D.A.: *Computer Architecture, Fifth Edition: A Quantative Approach*, Morgan Kaufmann Publishers, San Francisco, CA, USA, 5th ed., 2011.

[11] *Intel Xeon Phi^{TM} Coprocessor System Software Developers Guide*, document available on webpage `http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-coprocessor-system-software-developers-guide.pdf`.

[12] Kong M., Pop A., Pouchet L.N., Govindarajan R., Cohen A., Sadayappan P.: Compiler/Runtime Framework for Dynamic Dataflow Parallelization of Tiled Programs. In: *ACM Trans. Archit. Code Optim.*, vol. 11(4), pp. 1–30, 2015. `http://doi.acm.org/10.1145/2687652`.

[13] Lattner C.: LLVM. Figure available on webpage `http://www.aosabook.org/en/llvm.html`.

[14] Lattner C., Adve V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pp. 75–86, IEEE Computer Society, Washington, DC, USA, 2004. `http://dl.acm.org/citation.cfm?id=977395.977673`.

[15] Malik A.M.: Optimal Tile Size Selection Problem Using Machine Learning. In: *Proceedings of the 2012 11th International Conference on Machine Learning and Applications – Volume 02*, ICMLA '12, pp. 275–280, IEEE Computer Society, Washington, DC, USA, 2012. http://dx.doi.org/10.1109/ICMLA.2012.214.

[16] Puchet L.N.: PolyBench/C the Polyhedral Benchmark suite. Benchmark available on webpage http://web.cse.ohio-state.edu/~pouchet/software/polybench/.

[17] Rahman M., Pouchet L.N., Sadayappan P.: Neural Network Assisted Tile Size Selection. In: *International Workshop on Automatic Performance Tuning (IWAPT'2010)*. Springer-Verlag, Berkeley, CA, 2010.

[18] Rupp K., Horovitz M., Labonte F., Shacham O., Olukotun K., Hammond L., Batten C.: 40 Years of Microprocessor Trend Data. Figure available on webpage http://www.karlrupp.net/wp-content/uploads/2015.06/40-years-processor-trend.png.

[19] Shirako J., Sharma K., Fauzia N., Pouchet L.N., Ramanujam J., Sadayappan P., Sarkar V.: Analytical Bounds for Optimal Tile Size Selection. In: *Proceedings of the 21st International Conference on Compiler Construction*, CC'12, pp. 101–121. Springer-Verlag, Berlin, Heidelberg, 2012. http://dx.doi.org/10.1007/978-3-642-28652-0_6.

[20] Verdoolaege S.: isl: An Integer Set Library for the Polyhedral Model. In: Fukuda K., Hoeven J., Joswig M., Takayama N. (eds.), *ICMS'10 Proceedings of the Third International Congress on Mathematical Software*, pp. 299–302, Springer--Verlag, 2010.

Affiliations

Dominik Adamski
Lodz University of Technology, Department of Microelectronics and Computer Science

Grzegorz Jabłoński
Lodz University of Technology, Department of Microelectronics and Computer Science

Received: 03.02.2016
Revised: 26.11.2016
Accepted: 26.11.2016