A Brief Discourse on Important Questions in High Performance Architectures and Their Compilers

CS243: High Performance Architectures and Their Compilers

Christopher A. Wood, Julian Lettner, Brian Belleville

December 13th 2013

Table of Contents

1. Why do we care about parallelism? What are the limitations of parallelism (Amdahl's law)?

Parallelism has become increasingly important for the following two reasons. Moore's law still holds, that is, the number of transistors on an integrated circuit doubles roughly every 2 years, but due to physical and thermodynamic constraints this does not result in increasing CPU speeds anymore, instead, the number of cores is growing. This means that we can not expect our sequential programs to magically run faster as the underlying hardware gets more sophisticated over time. On the high level we have to design our algorithms and programs with parallelism in mind; on the low level we need smarter tools, i.e. compilers, to take advantage of the increasing numbers of cores. However, the speedup that can be achieved by introducing parallelism is limited. Suppose a program contains a sequential fraction, i.e., a part of the program can not be parallelized. Amdahl's law states that the speedup of a program using multiple processors in parallel is limited by that sequential fraction. For example, if the sequential part of a program is 5% then the maximum speedup is 20x, no matter how many cores are used. A large amount of research has been dedicated to find techniques that reduce the sequential fraction of programs. The following questions are all somewhat related to this topic (pipelining, spatial and temporal parallelism, ILP, vectorization, trace scheduling). The second reason why parallelism is important is that it often allows us to tackle bigger problems. Even if the potential speedup is limited due to the structure of the program or algorithm we might be able to increase the problem size by using more cores. This is especially true in the context of high performance and scientific computing, for example, using a more precise model for weather prediction.

2. What is Instruction-Level Parallelism (ILP)? Why is it hard to achieve an acceptable degree of parallelism (ILP)?

Instruction-level parallelism (ILP) is a measure of the number of instructions in a program that can theoretically be executed concurrently. Accordingly, the amount of actual ILP that can be achieved depends heavily upon the details of the application and corresponding compiler's ability to identify and generate code that exploits ILP, as well as the available hardware features that are able to fetch and execute instructions that can be executed in parallel. Therefore, achieving high measures of ILP is a challenge that falls upon compiler developers and processor designers. Technically, a set of instructions can only be executed in parallel if there does not exist any data or procedural dependencies that exist between the instructions. Once independent instructions are identified and the corresponding code is generated, the processor must support the ability to fetch and identify these independent instructions so that they may be executed in parallel. Achieving a high degree of parallelism is a hard problem as it is not sufficient to just look at basic blocks when searching for ILP. Basic blocks usually only consist of a few instructions, i.e., they are too short to yield an acceptable amount of parallelism. Therefore compilers must look past branches (and loops) when searching for ILP. Some of the techniques that do just that are detailed by the following questions.

3. What is the difference between spatial and temporal parallelism?

Spatial parallelism is parallelism that results from duplicating the available computation units on a particular platform (e.g., CPU cores, functional units, etc.). Traditionally such duplication is done at the hardware level, but spatial parallelism can also be applied at the software level by duplicating the number of available threads that are running concurrently on multiple cores. Conversely, temporal parallelism is parallelism that results from pipelining the execution of computations, or instructions, on some sequence of computational units. In order to exploit this pipeline, each computation must be partitioned into (roughly equal length) tasks. We refer to this as temporal parallelism because each of these smaller sub-tasks can be computed concurrently (i.e., at the same time) on different computation units in the pipeline, while the available hardware resources are not necessarily duplicated. Given these descriptions, the differences between spatial and temporal parallelism are that in the former we achieve parallelism by duplicating the available computation units and performing multiple, independent tasks concurrently, whereas with temporal parallelism we use a single set (chain) of hardware resources to execute smaller sub-tasks of a computation, overlapped in time.

4. What is Vectorization?

Vectorization is a compiler transformation that translates traditionally sequential (scalar) instructions to single parallel (vector) instructions. A simple example of compiler vectorization is the execution of a finite number of iterations of an independent loop body (i.e., one in which loop iterations do not depend on previous iterations or shared variables). Therefore, given the ability to transform N sequential and independent instructions into a single vector instruction means that a theoretical speedup of N can be achieved. Similar to instruction-level parallelism, supporting the execution of vector instructions depends on the compiler's ability to identify data dependencies between such instructions and also on the underlying hardware to support the decoding and execution of such vector instructions. Today, even commodity hardware contains CPUs which support operations on small vectors and there is a trend for using GPUs for massive vector computations.

5. Define and describe a data flow dependency graph and control flow graph. How are they used by compilers?

In the field of compiler development, a data flow dependency graph (DFG) is a graphical representation of the flow of data as it is used in a program, and conversely, a control flow graph (CFG) is a graphical representation of the execution trace of a program at runtime. Compilers leverage DFGs to identify subsets of instructions without data dependencies that can be executed in parallel for improved instruction-level parallelism. In addition, CFGs are fundamental in program analysis for determining structures such as conditional branches, iteration constructs, and function call sites, among other important pieces of information. Compilers will leverage CFGs to determine how a program behaves at runtime to help guide optimizations and identify instruction and statement dependencies. Furthermore, information in a CFG can be used to identify spots for code modification, such as loop unrolling or function inlining.

6. How can solutions to Diophantine equations be used to determine array data dependencies, and how can this information be used to increase parallelism?

This problem arises if we have a loop that is operating on an array, we need a way to prove which array operations have no data dependencies. You can examine the expressions used to compute the loop index and see if they are similar, but that provides no guarantees that there is not dependence. The general approach is when testing if two array accesses **A** and **B** are independent, you must determine the possible values for the array index ***I*** at each point, $\boldsymbol{I_a}$ and $\boldsymbol{I_b}$ in terms of the variables found in the program. The problem becomes finding a solution to the equation $\boldsymbol{I_a} = \boldsymbol{I_b}$. The general form of this becomes finding a solution to the Diophantine equation shown in equation 1. If you find that no solution exists, you have proven that there are no data dependencies between A and B. This can be used to increase parallelism because if it can be shown that there are no data dependencies between A and B, then A and B can be scheduled to be executed in parallel without altering the semantics of the program.

$$I_a = a_1 j_1 + \ldots + a_n j_n$$
$$I_b = b_1 j'_1 + \ldots + a_n j'_n$$
$$a_1 j_1 + \ldots + a_n j_n = b_1 j'_1 + \ldots + a_n j'_n$$

Equation 1. The Diophantine equation relating the array index at **A** and **B**.

7. What is the difference in the DOALL and DOACROSS models of parallelism? Are there instances where they don't apply?

The DOALL model of parallelism is perhaps the most simple model of parallelism that seeks to identify and compute statements of a cross-iteration independent loop body in parallel. Specifically, given a loop with straight-line independent bodies, i.e., bodies that do not have any shared state or rely on shared, non-local variables, the loop iteration space is partitioned among a larger number of available computational units and simultaneously executed independently on each. In this context, we refer to a straight-line loop body as one that does not contain any branches or early exits, which are not able to be transformed to a DOALL execution.

```
int[] A = new int[N];
for (int i = 0; i < N; i++) {
    A[i] = i * i;
}
```

Figure 1. A loop with a simple straight-line body.

Contrary to DOALL parallelism, the DOACROSS model of parallelism seeks to identify and exploit parallelism that exists in loops with cross-iteration dependencies. In doing so, DOACROSS transforms often involve the introduction of synchronization constructs such as per-iteration semaphores or barriers to ensure that parallel executions of dependent loop bodies does not violate program correctness. In a sense, DOACROSS parallelism applies when loops are "partially parallel," and the transform seeks to maximize the amount of parallelism that can be obtained across all such loops. This may result in different statements in separate iterations of a loop being executed concurrently while maintaining loop dependencies. Graphically, this can be viewed as parallelism along the "wavefront" of the iteration space. An example of this wavefront is shown in Figure 2 below, where the wavefront corresponds to the parallelism that can be achieved from the following loop. (Note that this loop is trivial to illustrate the wavefront of parallelism with the DOACROSS transform. It can be rearranged and rewritten for better parallelism.)

```
int[] A = new int[N];
int[] B = new int[N];
for (int i = 0; i < N; i++) {
    A[i] = i * i;
    if (i != 0) B[i - 1] = A[i];
}
```

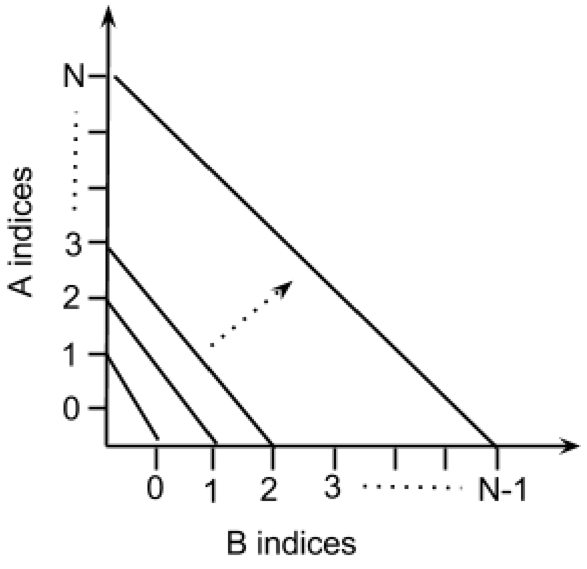

Figure 2. Parallel wavefront of the DOACROSS transform when applied to the (trivial) loop with cross-body dependencies.

The DOACROSS model does not apply when the cost of synchronizing loop-dependent operations outweighs the benefit of performing them in parallel. This is often true when the majority of each loop bodies is in some way dependent on one or more of the former iterations of the same loop.

8. What is Trace Scheduling? What is Superblock Formation?

Trace scheduling is a compiler optimization that seeks to improve instruction-level parallelism by grouping frequently executed paths in a program CFG, which can be determined by static branch prediction techniques or runtime profile information, into a single block, enabling the search for instruction-level parallelism on a larger set of instructions. This linear group of basic blocks, which is referred to as a trace, only consists of branches and regular statements (i.e., there are no iteration constructs) and are assumed to be executed with high probability at runtime. Once a trace is selected, the data flow dependency graph is updated to include this new, larger basic block and is then analyzed for additional instruction-level parallelism. Also, as a result of this grouping, safety code must be added to all branch exits from the trace that aren't included in the larger new basic block group so as to ensure correctness when such branches are taken. For example, if there existed an instruction in the original trace of basic blocks that preceded a conditional branch and, upon trace scheduling, is moved after this conditional branch, this statement would need to be copied on the exit path of the conditional branch after trace scheduling to maintain correctness. The treatment of incoming branches follows the same idea, i.e., adding safety code to guarantee correctness, but is more involved. It is harder because there might not exist a point in the trace that corresponds to the original branch target inside the merged basic block. For example, if instruction I was the target of an incoming branch and succeeded instruction I', but after trace scheduling, those instructions are executed in parallel, then branching to I would also execute I' which is not correct. The solution is to branch to a point in the trace where no additional instructions will be executed and compensate for the missing instructions by generating additional code. Note that there is a related technique called Superblock Formation which follows the same idea as Trace Scheduling. However, it is less sophisticated as it simply splits (copies the remaining part of the merged block) for every incoming branch.

9. What are the difference between a VLIW and superscalar CPU in terms of how parallelism is achieved?

A Very Long Instruction Word (VLIW) CPU is one that is specifically designed to be able to fetch, decode, and execute multiple instructions in parallel, where such instructions are generated statically by a compiler. In other words, VLIW CPUs can execute more than one instruction concurrently if the compiler generates such independent instructions. In contrast, a superscalar CPU is one that has redundant computational units on the same chip that are capable of executing multiple instructions at once, where such instructions are dynamically determined by the CPU control logic. While superscalar CPUs usually employ pipelining for temporal parallelism within each redundant computational unit, there is a distinct difference between these two forms of parallelism, as superscalar CPUs are characterized as a form spatial parallelism.

10. How can runtime behavior be leveraged by a compiler to make better optimization decisions?

Integrating runtime behavior into the traditional compiler workflow, often referred to as feedback-driven optimization (FDO), enables the compiler to perform more aggressive compiler optimizations for that are targeted towards normal, or production-time, use cases. FDO techniques such as profile-guided loop unrolling and function inlining are two prominent cases where runtime behavior can be leveraged to make better optimization decisions. In the former case, static loop unrolling passes are often overly conservative and only unroll loops whose execution count can be determined at compile-time. However, program runtime information such as the average number of loop iterations can be integrated into a new optimization pass that unrolls the loop for this average count. By expectation then, it follows that this level of unrolling will typically benefit the overall program performance by reducing the overhead of loop conditional tests. In the case of function inlining, leveraging the average number of times a function is invoked at a particular call site can be used to help determine whether or not inlining will positively benefit the program via reduced function call, setup, and teardown overhead when compared to the possible degradation that may be caused by decreased data and memory locality. There exists many diverse types of profile-guided optimizations in the literature, and we discuss a variety of such methods in our survey accompanying this document.