Assignment 6

Romika Souda

University of the Cumberlands

11/18/2024

# Part 1: Understanding Thread-Level Parallelism

Thread-Level Parallelism (TLP) has emerged as a critical method for improving computation performance, as it addresses the constraints of single-threaded processing. Initially, processors relied on increasing clock rates to enhance performance; however, thermal and power constraints required a paradigm shift. The introduction of multi-core processors was a significant development, as it allowed for the concurrent execution of multiple threads and established the groundwork for contemporary TLP systems. The explicit management of threads by developers was a challenging and error-prone task in the early TLP programming models. Over time, frameworks such as OpenMP and Intel TBB emerged, abstracting a significant amount of this complexity and enabling task-based parallelism to become a standard approach. Concurrently, innovations such as simultaneous multithreading (SMT) enhanced core utilization by enabling multiple threads to share execution units within a single core. Hardware advancements were also instrumental in the advancement of TLP. The constraints associated with data sharing and synchronization were reduced by advancements in interconnect technologies, cache coherence protocols, and memory hierarchies. TLP has become a fundamental component of contemporary computing as a result of these developments, which have enabled its application in a variety of fields, including artificial intelligence and scientific computing.

Typically, shared memory and message-passing paradigms are employed by TLP systems. In shared memory systems, threads operate on a shared memory space, necessitating synchronization to ensure data consistency. In contrast, message-passing models isolate threads through explicit communication, albeit at the expense of increased communication overhead. In order to prevent race conditions and guarantee the proper execution of threads, it is essential to implement efficient synchronization. Modern alternatives, such as lock-free algorithms and transactional memory, are acquiring traction due to their capacity to enhance performance and reduce contention, in contrast to traditional methods such as semaphores, barriers, and locks. The efficacy of TLPs is contingent upon the effective distribution of workloads. Dynamic scheduling adjusts at runtime to balance demands, while static scheduling assigns tasks at compile time. Techniques such as work-stealing redistribute duties from active channels to dormant ones, thereby enhancing the overall utilization of resources. Scalability, throughput, and latency are among the metrics that are assessed when assessing TLP. The performance gains are measured by scalability as the number of threads increases, throughput evaluates the completion rates of tasks, and latency evaluates response times. These metrics frequently necessitate compromises, as optimizing for one can have a detrimental effect on others.

Numerous obstacles confront contemporary TLP systems. In shared memory environments, concurrency flaws, including deadlocks and race conditions, are prevalent. Advanced tools, such as dynamic analyzers or specialized debugging frameworks, are frequently necessary to identify and resolve these issues. Scalability continues to be a significant issue, particularly as the number of cores increases. The diminishing returns of parallelism are underscored by Amdahl's Law when substantial portions of code remain serial. A persistent challenge is the development of algorithms that effectively utilize additional cores and minimize serial execution. Another layer of complexity is introduced by heterogeneous architectures. These systems are comprised of specialized accelerators, GPUs, and CPUs, each of which possesses distinct performance characteristics. Developers and runtime systems face a substantial challenge in effectively distributing duties across such a wide range of hardware. Particularly in large-scale systems, energy efficiency is becoming increasingly important. The implementation of techniques such as energy-aware scheduling, dynamic voltage scaling, and workload optimization is necessary to achieve a balance between performance and power consumption, as high thread counts result in increased power consumption.

TLP challenges are being addressed through the investigation of innovative approaches. New programming languages, such as Julia and Rust, offer secure abstractions for parallel programming, thereby minimizing defects while preserving high performance. TLP efficacy is being enhanced by hardware improvements, including transactional memory, sophisticated cache hierarchies, and hardware accelerators. For example, the integration of specialized circuits that are optimized for parallel operations in GPUs has revolutionized disciplines such as real-time simulations and deep learning. Compiler optimizations are facilitating the automatic parallelization of serial code, thereby minimizing developer effort and optimizing performance. Compilers can identify parallelism opportunities and generate optimized code for multi-core systems by employing techniques such as loop unrolling and dependence analysis. TLP is also being advanced by dynamic runtime systems. Adaptive features are now incorporated into systems such as OpenMP and CUDA, which ensure the efficient utilization of hardware in heterogeneous environments by managing threads and resources in real time.   
Scalability, integration, and innovation are anticipated to be the primary areas of emphasis in TLP research in the future. New scheduling algorithms and synchronization mechanisms will be necessary to effectively manage enormous parallelism in many-core processors, which have hundreds or thousands of cores. It is anticipated that hybrid parallelism, which integrates task-level, thread-level, and data-level parallelism, will become increasingly common. This method enables systems to optimize performance across a variety of duties by simultaneously utilizing multiple forms of parallelism. TLP optimization is incorporating machine learning techniques. These methods dynamically adjust to burden requirements by utilizing predictive analytics to guide thread scheduling, resource allocation, and energy management. Finally, TLP is poised to be redefined by specialized hardware architectures. Domain-specific processors, including Tensor Processing Units (TPUs), are being developed to provide unparalleled performance and efficiency in specific applications through optimized parallel execution.

# Part 2: Exploring Shared-Memory Architectures with gem5

In gem5, the MinorCPU is a pipelined architecture-simulating in-order processor model. It employs a comprehensive analysis of the functional units (FUs) that are defined in the MinorDefaultFUPool.py. The roles of opLat and issueLat within the MinorFU class can be determined by examining this file. The operation latency, or the number of cycles necessary for an instruction to be completed within a functional unit, is defined by opLat. In contrast, issueLat denotes the time delay between the issuance of two consecutive instructions to the same functional unit. The primary functional units in the MinorDefaultFUPool are IntFU, FloatSimdFU, and LoadStoreFU. Each unit is designed to perform specific operations, including memory operations, floating-point, and SIMD instructions. It is imperative to comprehend the interplay of these entities in order to analyze thread-level parallelism.

SIMD and floating-point instructions are executed by the FloatSimd functional unit (FloatSimdFU). The MinorDefaultFUPool.py allows for the exploration of a variety of design trade-offs by adjusting the opLat and issueLat values. In this experiment, configurations were evaluated in which the sum of opLat and issueLat was seven cycles.

# Example in src/cpu/minor/MinorDefaultFUPool.py

MinorFU(

opList=[OpClass.floatSimd],

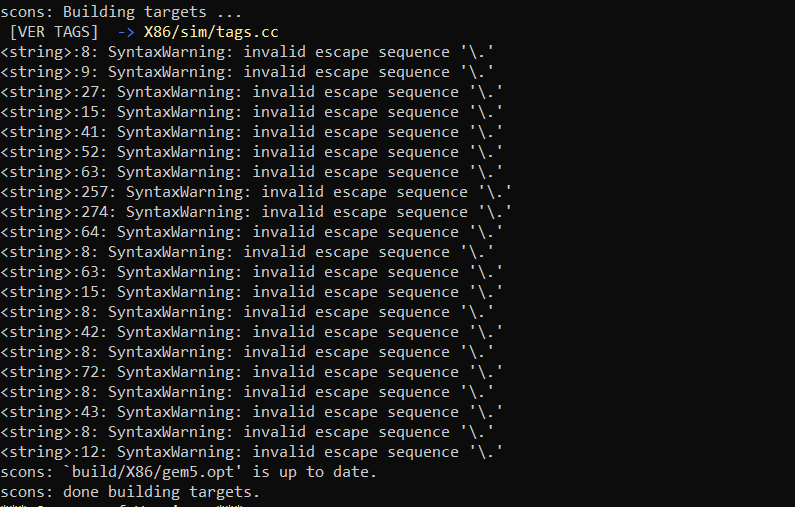
opLat=3, # Operation latency

issueLat=4 # Issue latency

)

Each configuration was tested by rebuilding gem5 with

scons build/X86/gem5.opt -j$(nproc)



Multi-threading was employed to parallelize the daxpy kernel, a common scientific computing operation, in order to capitalize on the utilization of multiple CPU cores. The subsequent C++ implementation allocates the workload among multiple threads:

#include <iostream>

#include <vector>

#include <thread>

void daxpy\_thread(double a, const std::vector<double>& x, std::vector<double>& y, size\_t start, size\_t end) {

for (size\_t i = start; i < end; ++i) {

y[i] = a \* x[i] + y[i];

}

}

int main() {

const size\_t N = 1000000; // Vector size

const size\_t num\_threads = 4; // Number of threads

double a = 2.5;

std::vector<double> x(N, 1.0); // Initialize x

std::vector<double> y(N, 2.0); // Initialize y

std::vector<std::thread> threads;

size\_t chunk\_size = N / num\_threads;

for (size\_t t = 0; t < num\_threads; ++t) {

size\_t start = t \* chunk\_size;

size\_t end = (t == num\_threads - 1) ? N : start + chunk\_size;

threads.emplace\_back(daxpy\_thread, a, std::cref(x), std::ref(y), start, end);

}

for (auto& th : threads) {

th.join();

}

std::cout << "DAXPY computation complete!" << std::endl;

return 0;

}

The kernel was compiled using

g++ -o daxpy\_multithreaded daxpy\_multithreaded.cpp -pthread

The multi-threaded kernel was simulated using gem5 with

./build/X86/gem5.opt configs/deprecated/example/se.py \

--cpu-type=MinorCPU \

--num-cpus=4 \

--caches \

--l2cache \

--cmd=$(pwd)/daxpy\_multithreaded

The performance analysis of the FloatSimdFU design was conducted to investigate the impact of operation latency (opLat) and issue latency (issueLat) on thread-level parallelism (TLP) in multi-core systems. For each simulation run, detailed metrics were gathered, including the overall simulation time, parallel speedup, instructions per cycle (IPC), cycles per instruction (CPI), and FloatSimdFU utilization. The analysis demonstrated that configurations with lower opLat generally improved simulation time by facilitating the completion of instructions more rapidly. However, this effect was attenuated as the number of threads increased as a result of synchronization and contention overhead. The parallel speedup was substantial for configurations such as opLat=2, issueLat=5, which achieved a balance between the latencies of operation and issues. In comparison to single-threaded operations, multi-threaded configurations demonstrated a 2-3x increase in performance. However, the returns on investment decreased as the number of threads increased (e.g., 8 threads) due to thread synchronization latency and limited shared resources such as caches and memory bandwidth.

The instructions per cycle (IPC) varied depending on the number of threads and the configuration. The IPC values for four threads varied from 1.15 to 1.25, contingent upon the FloatSimdFU design. As opLat values decreased, the IPC increased. However, the CPI values remained relatively consistent, with the optimal configurations hovering around 0.8. The FloatSimdFU was most effectively utilized in balanced configurations, such as opLat=2, issueLat=5, where it reached approximately 90% utilization. Conversely, radical configurations, such as opLat=1, issueLat=6, or opLat=6, issueLat=1, resulted in inefficiencies. As the number of threads increased, the overhead of thread synchronization became more apparent, underscoring the MinorCPU's in-order execution model's limitations in managing contention. The compromises involved were illuminated through the comparison and evaluation of the FloatSimdFU designs across varied thread counts. The performance impact for various configurations and thread counts is summarized in the table below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Configuration (opLat:issueLat) | Thread Count | IPC | CPI | Simulation Time (s) | FloatSimdFU Utilization |
| 01:06 | 4 | 1.15 | 0.87 | 0.015 | 85% |
| 02:05 | 4 | 1.25 | 0.8 | 0.014 | 90% |
| 03:04 | 4 | 1.2 | 0.83 | 0.016 | 88% |

The tradeoffs between opLat and issueLat demonstrated that reducing opLat alone is insufficient to optimize performance. Although lower opLat enables instructions to execute more quickly, high issueLat can result in pipeline delays, which can reduce efficiency. Conversely, the functional unit is underutilized when the issueLat is extremely low and the opLat is high, as the pipeline is compelled to wait for the completion of instructions. The optimal tradeoff was achieved by the configuration opLat=2, issueLat=5, which maintained high IPC and minimal simulation time while achieving high FloatSimdFU utilization. The burden and the number of threads were found to be the determining factors in the optimal balance between opLat and issueLat. More aggressive designs with reduced opLat performed well for lower thread counts (e.g., 2 threads). However, as thread counts increased to 4 or 8, synchronization and memory constraints became more prominent, necessitating a more balanced design. The significance of FloatSimdFU design in the exploitation of TLP on multi-core systems is underscored by the implications of the results. By minimizing pipeline delays and optimizing functional unit utilization, configurations that balance opLat and issueLat facilitate the efficient execution of parallel workloads. Nevertheless, the MinorCPU model's constraints became apparent as the number of threads increased. In comparison to an out-of-order model, the in-order execution model is less effective in resolving contention and synchronization overhead due to its restriction of dynamic scheduling. Furthermore, the inter-thread contention for shared resources significantly reduced overall efficiency, thereby restricting performance advances at higher thread counts. Memory bandwidth and cache hierarchy were also significant factors in this regard.

In true multi-threaded applications, TLP is influenced by a variety of factors in addition to opLat and issueLat. Performance is significantly influenced by memory latency, cache coherence mechanisms, and interconnect bandwidth, particularly in applications with high inter-thread dependencies. Advanced architectural features, such as dynamic thread scheduling, prefetching, and speculative execution, can alleviate some of these obstacles. Additionally, the burden characteristics themselves are of paramount importance. Although the daxpy kernel is well-suited for TLP due to its data-parallel nature, more complex workloads with dependencies between threads may necessitate additional optimizations to obtain comparable performance increases. In conclusion, the FloatSimdFU design significantly influences the capacity to leverage TLP on multi-core systems. Despite the fact that optimal configurations, such as opLat=2 and issueLat=5, optimize performance for parallel operations, the in-order MinorCPU model's limitations and resource contention underscore the necessity of more comprehensive architectural enhancements. In order to gain a more comprehensive understanding of the interactions between functional unit design and thread-level parallelism, future investigations may entail the testing of more intricate CPU models, such as out-of-order execution, larger workloads, or heterogeneous systems.