CMPSCI 691W Parallel and Concurrent Programming **Spring 2006**

Lecture 7: Mar 1

Lecturer: Emery Berger *Scribe: Shangzhu Wang*

7.1 Overview

Most of the server architectures introduced in the previous lecture cannot provide both good performance and the ease of programming. MT/MP (Multi-thread/Multi-process) architectures have big performance overhead from context switches and are subject to concurrency issues such as race conditions. Event-driven architectures SPED (single-process event-driven) and AMPED (asymmetric multiple event-driven) have better performance but are more complex to program. This lecture introduces two advanced approaches that attempt to alleviate the problems with performance and complexity for server architectures.

7.2 Capriccio

Thread-based server architectures are more intuitive and easier to program, but they have significant scalability problem. Additional difficulties with thread-based architectures include admission control and dealing with potential race conditions.

Capriccio is a scalable, user-level thread package for high-concurrency servers developed at U.C. Berkeley by Behren et al. [2]. Three key features of Capriccio are *scalability to 100,000 threads*, *efficient stack management*, and *resource-aware scheduling*.

7.2.1 Linked Stacks

Capriccio uses *linked stacks management* to support dynamic stack chunk allocation to avoid waste of stack space in traditional thread systems. This is supported by a combination of compile-time analysis and run-time checks.

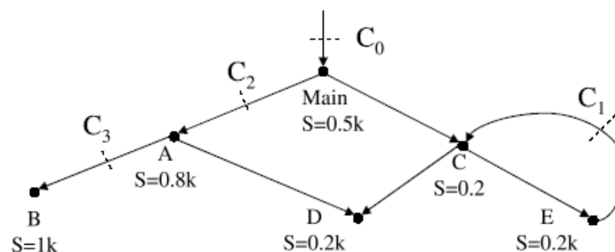

Figure 7.1: An example of a call graph annotated with stack frame sizes.

The compiler-time analysis is based on a *weighted call graph* [1](Figure 7.1) representation for the program. Each function in the program is represented by a node in the graph, weighted by the maximum amount of stack space that a single stack frame for that function will consume. Each edge in the graph represents a direct function call. Thus, paths between nodes correspond to sequences of stack frames that may appear on the stack at run time.

The analysis is based on *checkpoints* that are placed at call sites at compiler-time. A checkpoint is a small piece of code that determines whether there is enough stack space left to reach the next checkpoint without causing stack overflow. If not enough space remains, a new *stack chunk* is allocated, and the stack pointer is adjusted to point to this new chunk. At the exit of the function, the stack chunk is unlinked and returned to a free list. This approach may run into problems when it has to handle library code.

7.2.2 Resource-Aware Scheduling

Capriccio views an application as a sequence of stages, deduce the stages automatically, and have direct knowledge of the resources used by each stage. This automated scheduling can give priority to tasks that are closer to completion to reduce the load on the system. It can identify which stages are bottlenecks and thus when the server is overloaded. It can also be used to provide admission control and to improve response time.

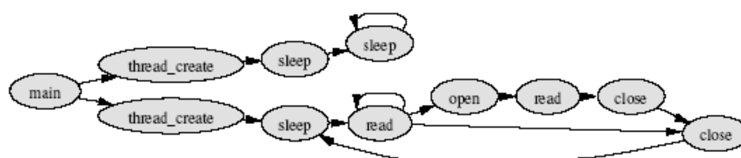

Figure 7.2: An example of a call graph

The key abstraction used for scheduling in Capriccio is the *blocking graph*(Figure 7.2), which contains information about the places in the program where threads block. Each node is a location in the program that blocked, and an edge exists between two consecutive blocking points. The "location" represents the call chain that was used to reach the breaking point. The average running time of each stage, the number of times an stage is taken, and the changes in resource usage are tracked at run time through annotations on edges and nodes in the blocking graph.

Capriccio keeps track of resource utilization levels and decides if a resource is at its limit. It also predicts the impact on each resource of a specific scheduling based on the node annotations. Based on these information, Capriccio increase utilization until it reaches maximum capacity, and then throttle back by scheduling nodes that release that resource. In the implementation, separate run queues are maintained for each node in the blocking graph. Capriccio periodically determines the relative priorities of each node based on the prediction of their subsequent resource needs and the overall resource utilization of the system. Once the priorities are known, a node is then selected by *stride scheduling* [3]. Stride scheduling is a deterministic variant of the lottery scheduling algorithm. The basic idea of the lottery scheduling is to give out "tickets" in proportion to the priorities and then pick a winner.

The resources that are tracked in Capriccio are CPU, memory, and the number of file descriptors.

In summary, such resource-aware scheduling does not quite work because in practice, there is no good way

[1]it actually contains both control flows (such as a while loop) and function calls

of telling whether an application is consuming resources in a way it is supposed to or in a way that may cause unexpected problems such as overloading.

7.2.3 Scalability Results

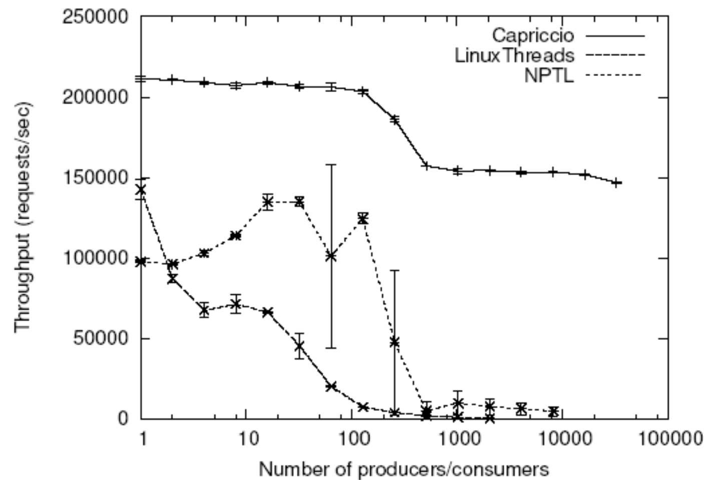

Figure 7.3: Producer-Consumer scheduling and synchronization performance.

Figure 7.3 shows the average throughput and standard deviations when a simple producer-consumer microbenchmark is run on Capriccio, LinuxThreads (the standard Linux kernel thread package), and NPTL version 0.53 (the new Native POSIX Threads for Linux package). Capriccio outperforms NPTL and Linux-Threads in terms of both raw performance and scalability. Throughput of LinuxThreads begins to degrade quickly after only 20 threads are created, and NPTL's throughput degrades after 100.

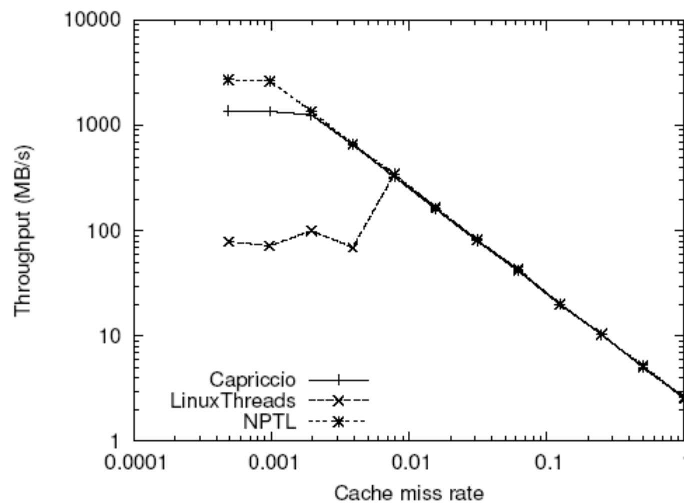

Figure 7.4: Disk I/O performance with buffer cache.

Figure 7.4 shows the disk I/O performance of the three thread packages when using the OS buffer cache. Based on the same microbenchmark, this test measures the throughput achieved when 200 threads read continuously 4K blocks from the file system with a specified buffer cache miss rate. For a higher miss rate, the test is disk-bound; thus, Capriccio's performance is identical to the other two. When the miss rate is low, the program is CPU-bound, so throughput is limited by per-transfer overhead. In this case, Capriccio's maximum throughput is about half of NPTL's, which means Capriccio's overhead is twice as much.

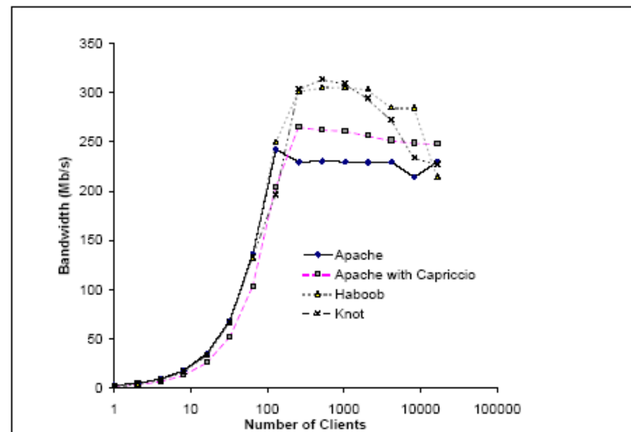

Figure 7.5: Web server bandwidth versus the number of simultaneous clients.

Figure 7.5 shows the performance results when Capriccio is run under a realistic web server load. Apache's performance improved nearly 15% when run under Capriccio. Knot's performance matched that of the event-based Haboob web server.

7.3 Flux

Programming high-performance server applications is challenging. Flux [1] is a language that allows programmers to compose off-the-shelf, sequential C or C++ functions in to concurrent servers. Flux programs are type-checked and guaranteed to be deadlock-free. It is independent of any runtime architectures. The Flux compiler also automatically generates discrete event simultaneous that accurately predict actual server performance under load and with different hardware resources.

A number of servers have been built using Flux including a web server, an image-rendering server, a BitTorrent peer, and a game server.

7.3.1 Flux Language

Concrete nodes represent the actual C and C++ implementation. Flux requires type signatures for each node. The name of a node is followed by the input arguments in parentheses, followed by an arrow and the output arguments. *Source nodes* are concrete nodes that only produce output to initiate a data flow. *Abstract nodes* represent a flow through multiple nodes which are composed from concrete nodes.

Errors are handled by error handlers. When a node returns a non-zero value, FLux checks to see if an error handler has been declared for the node. If no error handler exists, the current data flow is terminated.

Flux lets programmers use the *semantic type* of a node's output to express multiple possible paths by directing the flow of data to the appropriate subsequent node. A semantic type is a boolean function supplied by the Flux programmer that is applied to the node's output.

Concurrency constraints can be specified in Flux using arbitrary symbolic names. These constraints resemble locks and a node only runs when it has acquired all of the constraints. The Flux compiler enforces a canonical order for lock acquisition to avoid deadlock. Concurrency constraints in Flux are also reentrant, preventing deadlock that could occur due to multiple locking. Exposing concurrency constraints also enables the Flux compiler to generate more efficient and specialized code for different runtimes.

7.3.2 Compiler and Runtime Systems

The Flux compiler first reads in the Flux source and constructs a representation of the program graph. It then process the internal representation to type-check the program. Once the code has been verified, the runtime code generator processes the graph and outputs C code that implements the server's data flow for a specific runtime. Finally, the code is linked with the implementation of the server logic into an operational server. The current Flux compiler supports three different runtime systems: one thread per connection, a thread-pool system, and an event-driven runtime. [1]

7.3.3 Experiment and Evaluation

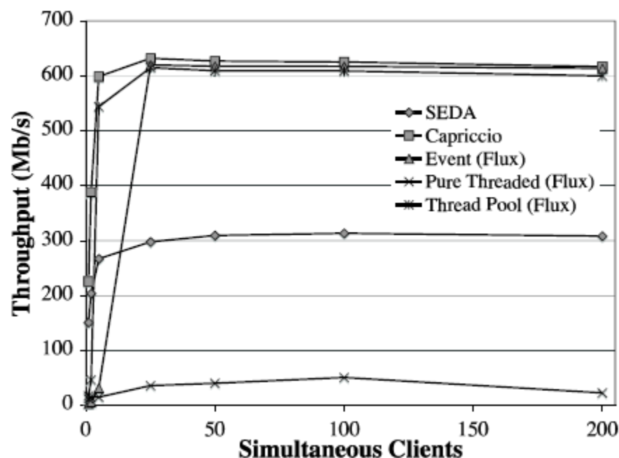

Figure 7.6:

A benchmark that simulates a number of clients requesting files from the server is implemented to load test Flux. Figure 7.6 shows the the result of comparing the throughput of a Flux web server with other implementations while using simultaneous clients. The Flux web server provides comparable performance to the fastest webserver (Knot), regardless of whether the event-based or thread-based runtime is used. Figure 7.7 shows the result of comparing the throughput of the Flux BitTorrent peer with CTorrent, an implementation of BitTorrent written in C. Prior to saturating the netwoek, all of the Flux servers perform slightly worse than the CTorrent server.

In addition to the language support, Flux can also generate discrete-event simulators that predict server performance. Figure 7.8 shows that the predicted server performance by Flux and the actual performance

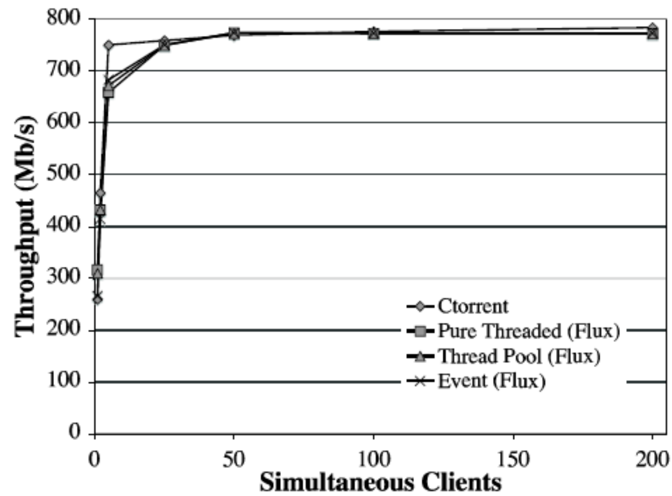

Figure 7.7:

for the image server match closely which indicates the effectiveness of the Flux's performance-predicting simulator.

References

[1] Brendan Burns, Kevin Grimaldi, Alex Kostadinov, and Mark Corner. Flux: A language for programming high-performance servers. Technical report, University of Massachusetts Amherst, 2006.

[2] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: scalable threads for internet services. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 268–281, New York, NY, USA, 2003. ACM Press.

[3] Carl A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management.* PhD thesis, Massachusetts Institute of Technology, September 1995.

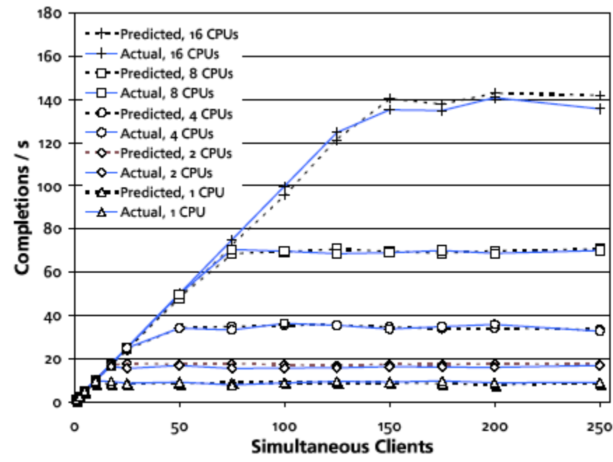

Figure 7.8: