Computational Systems Seminar, SS 2004
University of Salzburg, Department of Computer Science
Jakob-Haringer Str. 2, A-5020, Salzburg, Austria

The Context of Capriccio

Scalable Threads for Internet Services

By Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer

1st of July, 2004

Author: Claudiu Farcas
Academic Supervisor: Christoph M. Kirsch

Abstract

The survey covers the context of scalable threads models focusing on their usage for high performance Internet services and the improvements presented in the user level threading package Capriccio. It analyses the event vs. threads design methodologies, existing high performance server architectures, kernel and user level existing threading models and the usage of code analysis and smart compilers for better system performance.

Table of Contents:

1 Introduction

The current demand for high performance computing is pushing the hardware and software to new limits. The Internet services have and increasing demand for scalability and availability. More and more content is added online every day and users request for information is growing exponentially. Economics reasons push the need for automated tools and faster development of high performance applications such as database and web servers and web applications. It is becoming more important to get the maximum performance out of each system and reduce overall costs.

Traditional server-software design methodologies for Internet services such as one process per connection are doomed nowadays. The servers that hold the information have to deal with 1k-100k requests per seconds that generate a tremendous load on the application software, operating system and underlying hardware. For dynamic content not only one application is involved in producing high performance results but a group of applications. There are several bottlenecks in a real world information system (web content provider) composed of a mixed environment of databases, web servers and web applications. However, generally, the hardware is currently fast enough to meet the requirements of raw performance but the software is lagging behind.

2 Existing Models for High Performance Applications

The core of a high performance system is the operating system running on the system hardware and the application that provide the required services. Given the vast range of operating systems available and applications that provide similar services it is hard to decide, which solution is better and more important to understand why? However, at a closer look all systems designs have many common elements and we can group the approaches for high performance into two classes: events based and threads based.

The two models for developing applications are almost opposite in all aspects. The question for any given service is which model is better for developing an application for a specific service? The debates about a specific model are always counter balanced by arguments for the other model: programming skills required for high performance event based systems are confronted with easy understanding of threading code, high memory usage of multiple stacks required for threading are confronted with lightweight memory footprint of event based programs.

The advantages of one model are almost implicitly disadvantages for the other and debating on the best model is a long-term battle. However, in an empirical paper published in 1978, H.C. Lauer and R. M. Needham [LM78] revealed the two models as being duals of each other and that any system constructed according to a model can have a direct counterpart in the other model. We notice that the models presented in the paper are idealistic as there is no real system purely belonging to one class or the other. At lower levels, any given system is a mix of both models, and only the upper layer belongs to a specific model. Most developers that believe that a specific model is the solution to all problems generally neglect this aspect.

We can summarize the two models as message oriented and procedure oriented. The message-oriented system has generally a small number of processes that perform specialized operations and communicate their results using messages similarly with an assembly pipeline. The procedure-oriented system has generally many independent general-purpose processes that perform all operations on the data from the beginning to the end. We can share common resources between parallel streams of executions (processes or threads) via synchronizing techniques such as locks, semaphores, monitors, etc.

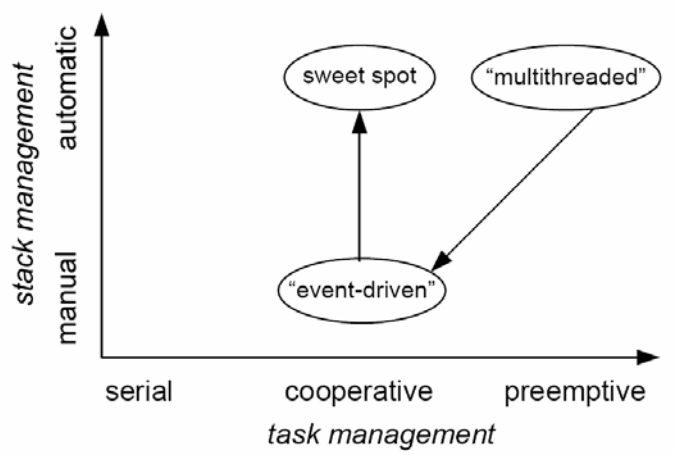

Figure 1 – Typical concurrency management [AHTBD02]

The behavior of a system designed according to any model has three major aspects:

- Execution time of the algorithms inside the application itself
- Computational overhead of the OS system calls
- Queuing, waiting times for resources that are influenced by external events, and scheduling decisions.

Changing a system designed according to a model into the other model maintains the code and execution time of the algorithms inside the application. Presuming that sending a message with allocation and queue overhead is timely equal with creating a new thread (forking) that involved similar operations, and watching a semaphore is also equal in time with waiting for new messages, and process switching and memory allocation is equally fast on both systems, the system will have identical performance no matter which model is used for its design. It is possible to adapt applications in order to exploit the benefits of both systems and avoid common problems by designing hybrid systems where both technologies can be used for improved performance (see Figure 1).

In the case of similar quality implementations of an application with both models, the only thing that can influence the performance of a specific model is the platform support. By platform, we understand the system hardware (CPU, memory architecture, storage mechanism) and the operating system with specific API and libraries to access the hardware resources. Some platforms are more adequate for threading models when true concurrency may benefit from multiple CPUs, others may benefit from the reduced memory footprint of the event model.

From the platform support point of view, there are at least the following aspects that are often neglected ("Four Horsemen of Poor Performance" [JD]):

- Data copies
- Context switches
- Memory allocation
- Lock contention

Data copies is generally the biggest performance killer and although most programmers avoid typical data copying operations there are lots of hidden or non obvious places where this could happen. An example of this would be a hashing function has all the memory-access cost of a copy and involves more computation [JD]. The best way to avoid data copies is to use a level of indirection such as pointers or descriptors (pointer and additional information needed to operate with the buffer such as length, offset; list pointers for fwd/back operations in a list). However, in certain cases copying operations are still unavoidable or even required if the effort to compensate for their effect is much higher (e.g., searching through long lists of pointers, breaking up large I/O requests, etc).

Most developers ignore the context switches because they presume that their impact on overall performance is negligible. This is true for simple applications and low CPU utilization but completely wrong in the case of high performance systems where not only the concepts but also their implementation has a great impact on the overall system performance. In the case of more active threads than actual processors, the increasing number of threads per processor leads also to an increasing number of context switches, and the system may spend more time switching between threads than actually executing their instructions. The most common way to decrease the number of active threads is by using the multiple connections per thread model (see Section 5). Splitting the server application into several stages such as listen, work, send results with an appropriate number of threads for each stage will improve the overall performance of the application and reduce thrashing [WCB01].

Memory management is one of the most common operations in any application, and contrary to popular beliefs, it is a time consuming task especially in busy systems that have a highly fragmented memory. One possible solution that may limit the flexibility of an application is the use of static preallocated memory, and sometimes a clever use of shared memory space for multiple objects. In addition, we can save frequent used objects (that are mostly allocated and soon after deallocated) onto a look-aside memory space instead of destroying them, and with a simple algorithm, we can check their existence before reallocation. Sometimes contiguous allocated space such as arrays may waste some memory compared with lists of objects but the access time performance is heavily improved (no need to walk through the list to find the N-th element). In the case of concurrent access requests to the same memory object, the locks may prevent one application from benefiting of concurrency. For a better synchronization that avoids waiting for resource release, we can use some clever cooperative multitasking technique.

Lock contention is a major problem when dealing with concurrent streams of execution. There are at least two kinds of locking: lightweight and simple but coarse grained that serialize the activities that logically are performed in parallel or complex and fine grained that reorder the requests for a resource for better utilization, loosing some performance with the

scheduling overhead. The right solution for an application is somewhere in between and finding it is not a trivial task. One possible idea is to build a staged application with separated data sets and avoid that two requests are in contention unless they are both in the same data set and in the same processing stage.

There are some other performance factors that have to be considered when choosing the right design model for an application, most of them deriving from the platform such as: storage subsystem, network protocol and TCP/IP stack implementation, page and cache size, signal based wakeup sequences, system calls overhead, etc. An idea is to develop a micro-benchmark application for that specific platform that measures the performance of each approach in the most sensitive areas [JD].

3 Events Based Model

The events based systems rely on signaling between different processes to transport data according to the application algorithm. Each process is handling a specific operation in the dataflow and we can assimilate the whole process with assembly pipeline. Depending on implementation, running processes can generate events via signals, messages, pipes, etc.

Generally, an event-based system is composed of an event monitor and some event handlers that express their interest in certain events. Handlers can be short-lived and running without preemption in the case of small operations (e.g., interrupt handlers for some I/O devices), or long-lived when performing time-consuming operations on their input data. In the last case, we can decrease the system load with cooperative multitasking or balancing on other CPUs.

3.1 Advantages

Most event-based systems have one execution stream and require no true CPU concurrency. The event monitor loops infinitely waiting for an event to happen and to pass it to the appropriate handler. We can easily use the idle time for background tasks without fearing that it may influence the performance of other processes.

There is no need for resource locking or synchronization, and hardly for preemption when handling one event at a time. This increased the utilization efficiency on single CPU system because there is no context switch or locking mechanism involved. The timing aspects of an application are only related to events and not to internal scheduling [Ous96].

The events are perfectly suited for graphic user interfaces and we can use the registration service provided by the event monitor to enforce security on the application. In such cases, the handler implements a single behavior such as copy, delete, undo, confirm, etc.

Distributed systems are using the events for handling input data with low overhead. Each handler is processing an input signal or a chunk of input data and sends immediately a reply with the results.

3.2 Disadvantages

In the case of long non-uniform handlers running time, when a handler takes a longer time to execute, the others are blocked and the application may become non-responsive. There are some workarounds available depending on the target application: start a sub-process to handle the work and use events just to query its completion, or provide mechanisms for cooperative multitasking where handlers can yield CPU to each other.

Another solution would be splitting a workload in several handlers and add completion events between different stages. However, as there is no state maintained between events (a handler must return) this makes implementation much harder and the internal algorithm becomes hard to understand, and in some cases very complicated.

The most common problem with events is the "stack-ripping" phenomenon [AHTBD02]. At each point that a blocking call could occur, the developer has to rip apart the stack and create a closure invoked via callback. This procedure obscures the intent of the program and makes maintenance or extension hardly possible. It also applies for saving live state of an event handler when the programmer has to keep track of all pairs call/return.

In some cases event driven I/O is not possible because of poor buffering or lack of OS facilities (e.g., async I/O) or CPU concurrency with task/handler preemption.

4 Threads Based Model

A thread-based system relies on multiple independent instances of a procedure regarded as execution streams that work using the same algorithm on similar data to achieve higher processing throughput. The threading model derives from the process model for true concurrency and takes advantage of multiple CPUs in high-end systems. The initial implementations consisted of lightweight processes that share some common data in user space.

The threading model fits well a variety of applications and is regarded as a solution to benefit from the real performance of a system. A single threaded application that requires more performance can be easily extended by creating multiple threads that perform the major operations in parallel. In the case of multi-CPU systems, we gain increased performance with little effort. Using the POSIX standard for development will also make platform migration easier.

There are two mapping alternatives for implementing threads in an operating system:

- 1:1 - easier and more efficient scheduling for one kernel thread per one user thread proved to be the fastest approach
- M:N – closer to logical programming model but the asymmetric distribution of user threads on kernel threads has complexity problems, requires more resources and the overall performance is lower than 1:1.

4.1 Advantages

Threads are generally more powerful than events and closer to the parallel applications. The concurrency in modern server application generally starts from independent requests, and similar pieces of code handle them in parallel. The algorithm behind an application and the data flow is easier to understand as the programmer does not need care about transferring states between different parts of the application (e.g., between different methods implemented in separated modules) that implement successive operations. This way a threaded application can have long-lived stateful handlers without blocking. Shared data allows for fast global configuration and easy access to common resources that are crucial for database applications.

With preemptive scheduling and independent execution, threading systems can handle overlapping requests and perform concurrent I/O. It is possible to implement dedicated threads for each operation that can be a bottleneck for the system such as disk operations or network communication. We can split the application into multiple parallel working threads and a couple of interfacing threads that communicate with the environment (fetching input data, storing, communicating results).

We can avoid the common problems with stack allocation and scheduling by using smart compilers and code analysis. Stack space can be fragmented into linked blocks and allocated on demand by checking the requirements of the application in advance (at compile time or during runtime). With resource monitoring it is possible to avoid long stalls for resources and thrashing [BCZNB03].

4.2 Disadvantages

Earlier implementations based on lightweight processes were slow and required large amounts of memory to operate. In addition, the available process IDs were running to their limits relatively soon. It was impossible to create and operate thousands of threads because of scheduler overhead and memory resources.

Accessing shared data requires synchronization, and the locking mechanism severely diminishes the true concurrency aspects of the threading programming model. However, without synchronization, the threads that access some common resources can generate race conditions and may easily corrupt data.

Using locking mechanism can lead to deadlocks when a circular dependency between threads appears during the application execution. With simple locking by monitoring the availability of a resource, the true concurrency benefits are lost. Fine-grained locking mechanisms may improve application performance but may increase system load because of scheduling and context switching additional overhead [Ous96].

It is hard to design independent modules that communicate efficiently because one thread deadlock can lead to unpredictable behavior of the following module. Debugging threading systems is hard because of data dependencies and timing issues.

Porting threads based applications to different platforms (e.g., UNIX to Windows) is difficult without a common API interface. Also standard libraries are not thread safe on all platforms and can lead to unpredictable behavior of the application.

Stack allocation for each thread can lead to high memory consumption and thrashing. Early implementations allocate fixed size stack space for each thread even if the stack requirements for that thread were minimal. The address space in this case is wasted on most 32-bit processors.

4.3 Kernel Threads

The kernel threads are the lowest level of parallel streams of execution integrated directly into the core of the operating system. Here there are the benefits of true concurrency and multi-processor support, direct access to hardware resources and OS internals. Their usage is mostly for concurrent processing of kernel jobs and separating several possible bottlenecks such as disk I/O and network communication.

However, the kernel threads are highly platform dependent and hardly portable. Programming kernel threads is generally difficult and debugging can be a problem because all operations are in kernel space and deadlocks can lead to complete system locking. Errors in memory allocation or code can lead to unpredictable behavior and system crashes. Most upgrades in the internal algorithms and other improvements are only possible with kernel upgrades that generally require an expensive restarting of the whole system.

4.4 User Threads

We see the user threads as a portable way to handle concurrency in most operating systems. There are many threading packages available for most operating systems and that can be changed on the fly without any side effect on the operating system's behavior. The OS does not need to be upgraded in order to support certain threading package or application. However, direct access to OS structures and scheduling is slower than with the kernel threads.

The cleaner programming model, and total logical decoupling from the kernel threads, makes them the most used threading model for highly concurrent applications. The resulting code is generally highly portable and flexible to adapt to various user-level threading packages.

With the various approaches in implementing a high performance user level threading package it is generally easier to use the latest enhancements and state of the art technology with an application than to wait for their integration into the next stable version of the operating system.

The use of user level threads reduces kernel mode switching and kernel-memory utilization. We can improve their scheduling according to the application logic by code analysis and smart compilers. This can also lead to better memory management that fits the application's needs with very low overhead.

However, the blocking system calls present in the application code may stall the entire execution of all tasks. Existing high performance threading packages such as Filaments [PG95] are still slow at dealing with a large number of blocking calls. For a fast recovery from this problem, we must convert all the blocking system calls to non-blocking equivalents that pool the resource to check its availability and perform the requested system call only when this will result in an immediate return. The additional work for this task (all user code must be generally parsed) can be mostly automated but it depends on special code analyzers and smart compilers to perform the additional changes needed.

In addition, the true concurrency support of the kernel threads is mostly lost and there is hardly a benefit from multi processor systems. Mapping of user level threads to kernel threads decreases the performance of the application and the existence of two schedulers for the same purpose (for kernel and user threads) introduces additional overhead. The cache memory footprint of the application is increased and there will be more hardware stalls for fetching new instructions and data from the main memory.

The scheduling of user threads is generally non-preemptive and the compiler must provide some sort of cooperative scheduling in order to emulate concurrency. Each thread has to yield periodically the CPU to other threads. In some cases when some threads contain CPU intensive operations this can result to stalls for the other threads.

Handling precompiled libraries or static compiled applications is difficult as there is no information about the internal policy for scheduling, their stack allocation requirements or other common resources. Also keeping in sync with kernel development is required in order to allow seamless compilation of the application with new operating system releases. This is not always possible and requires sometimes adaptations to the program source code [ST].

5 Real World Systems

The existing high performance systems involve a variety of technologies for handling multiple clients in an economic way. These include multiple concurrent execution streams such as multi-process (MP) and multi-threaded (MT), event based such as single process event-driven (SPED) and even a hybrid approach: asymmetric multi-process event-driven (AMPED) architecture.

The multi-process or multi-threaded models are using generally the same design methodology based on multiple concurrent streams of execution with minor differences in the way data is passed between processing units (for processes there is no need for synchronization mechanisms on data access as each one has its own private memory space), and resource management. By default for each new connection, a new thread or process is created and the connections are multiplexed by context switching between the units that cannot continue because of blocking and the ones that still can process data. Both models can benefit from OS support for overlapping disk activity and fast network processing, but also inherit performance hits because of context switching, forking a new thread/process overhead, inter process communication respectively synchronization delays.

When using the one thread per connection model the increasing number of active threads per processor will lead to an increased number of context switches that will severely

diminish the performance of the system. Limiting the number of active threads to the actual number of available processors in a system is a possible solution to this problem but the typical limitation of only one active thread will prevent the utilization of other CPUs and the application will become mostly network bounded. The well-known Apache webserver that handles currently most web sites worldwide evolved from the MP to MT model with minimal performance improvements mostly because of existing threading implementations. Another example is the lightweight Knot-C webserver [BCB03].

The SPED architecture relies on multiple independent stages of execution, activated by an event dispatcher. It handles a connection by passing the data through all necessary stages and requiring no context switching or synchronization. Typical implementation uses generally a single process along with non-blocking I/O and an event notification system based on primitives such as select or pool that determines when the corresponding system call (e.g., for network read/write) will execute without blocking. For multi-processor systems, running multiple SPED servers in parallel takes advantage of the additional hardware performance and provides excellent results. An example lightweight system based on SPED is the μServer [BO01] that in addition uses the multi-accept procedure [CM01] to drain the accept queue before performing the work for each connection.

The AMPED architecture used by Flash webserver [PDZ99] consists of several helper processes that perform most time consuming disk I/O operations on behalf of the mail event driven process that uses the SPED approach. The result is a fast webserver that outperforms most MP or MT based web servers.

6 Capriccio Approach

We designed Capriccio as a high performance lightweight user level threading system trying to overcome the previous drawbacks of similar threading packages. It scales well up to 100 thousands threads and aims for POSIX compliance for easy integration with existing applications. It also features cooperative scheduling, asynchronous I/O operations, linked stack management for reduced stack space and resource aware scheduling. Most of these features are available via code analysis, automatic source changes and compiler optimizations.

The Capriccio threading system comes to prove that threading model is adequate for high performance computing (in this case internet services) but the problem that prevented its high scale usage is the poor implementation of current threading packages [BCB03]. By reducing the common $O(n)$ operations in the scheduler and minimizing the number of context switches and additional kernel crossing it can match the performance of event based systems.

The linear control flow introduced by threads is not regarded as a limitation but a natural way of thinking and easier programming. The event-based systems have the possibility to implement any kind of control flow but in practice, applications use the same call/return mechanisms available with threads.

Synchronization problems inherent to any threading system can benefit of cooperative multitasking on single processor systems in the same way as the event based systems. In

addition, the scheduling advantage at application level of the event-based systems can be applied to the virtual processor model of the threading system.

6.1 Stack Management

A common problem when using threads is stack allocation. As each thread corresponds to an execution stream, we can consider it as an independent program and apply regular stack allocation policies similar with any other process of the system. This however leads to wasted address and memory space that can rapidly decrease the system performance.

Typical implementations on Linux allocated 2MB of stack space for each user thread. Handling just 1000 connections required more memory and address space than most computers had at that time. Other approaches used in languages such as Olden are using the concept of "spaghetti stacks" [BW73] - where multiple environments share a common stack subdivided in sub-stacks that are interleaved, or "stacklets" [GSC95, CB01] - that are allocated at runtime based on previous code analysis. It is possible not to use any stack at all with kernel threads [DBRD91] by packaging the entire state in a dedicated memory block in a similar way with the stack ripping mechanism used in the event model [AHTBD02].

Capriccio uses a better approach by splitting the stack space in small chunks that are allocated gradually on demand. In order to detect when a new stack chunk is needed the application source code is parsed to detect the best place to insert new stack allocation points. In this process, we create a weighted call graph where nodes represent the function calls of the application weighted by the amount of stack space required for a single stack frame that the function consumes, and the edges represent the logical flow of operations.

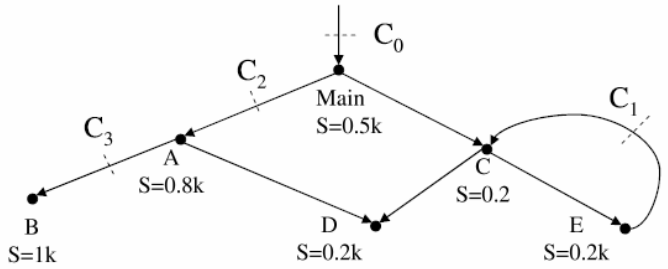

Figure 2 - Call graph annotated with stack frame sizes [BCZNB03]

Any path in this graph will correspond to a possible trace during the program execution and the sum of weights along the path will be equal with the total stack space required (see Figure 2). However, for most programs using recursion of function pointers it is not possible to detect at compile time the maximum stack space required during the execution of the program. This requires dynamic allocation and release of stack frames based on the evolution of the program. Capriccio executes all these operations in LIFO order allowing the application to reuse memory space and diminishing the overall memory consumption of the application.

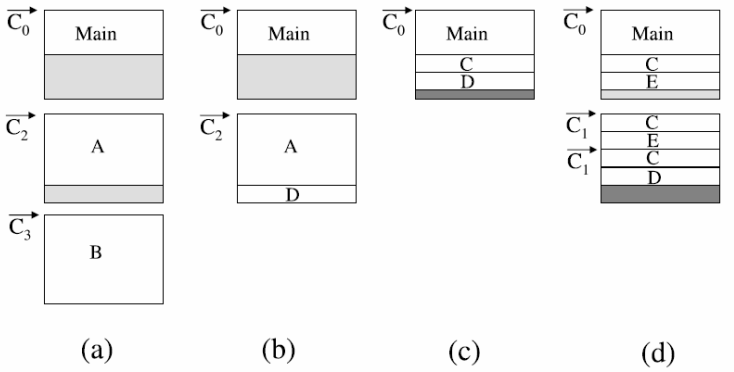

Figure 3 – Dynamic allocation/deallocation of stack frames [BCZNB03]

The checkpoints for inserting the stack management code are detected by performing a depth first search in the weighted call graph. Inserting checkpoints or stack management code before each call leads to low performance. The idea is to minimize the number of checkpoints and allocate at each checkpoint the right amount of stack space that will suffice until the program reaches another checkpoint. For recursive programs there are loops in the call graph that must be broken by inserting a compulsory checkpoint. In some cases, a small amount of stack space is wasted between successive checkpoints but this is a tradeoff for increased speed. Accessing the stack requires synchronization that Capriccio accomplishes via cooperative threading.

In the case of function pointers or external functions from precompiled libraries, it is not possible to detect the requirements of stack space in the weighted call graph. The solution in this case is to annotate the library or each possible function with an estimation of the amount of stack space required. This can be determined experimental via profiling tools but will not guarantee the behavior of the application.

This approach of dealing with the stack problem has a number of memory related advantages such as reduced stack footprint and virtual address space when running a large number of threads, reuse of memory space via dynamic LIFO allocation / deallocation mechanism that does not require a garbage collector. We also gain improved speed compared with traditional dynamic allocation on call and contiguous allocation at startup (performing malloc on fragmented memory space is expensive).

The disadvantages are coming from the additional step required to detect the right places for inserting the stack management code and library annotation in the case of precompiled libraries. However, most of these tasks can be fully automated and possibly integrated directly into compiler.

6.2 Scheduling

Scheduling the user level threads is problematic because when one thread calls a blocking system call the whole application will stop waiting for that system call to return. There is no preemption at user level code, and the only solution available is to use cooperative multitasking between threads. For the case of blocking system calls Capriccio introduces a novelty: an event monitor that transparently watches for the availability of each resource

needed and then performs the blocking system call. Although similar approaches were already described [FCEZ03], the Capriccio implementation is purely at user level and handles better the information gathered by monitoring the resource utilization level.

With this approach, all blocking system calls are rewritten at library level (libc 2.2 in Capriccio's case) and replaced with a call to the event monitor that yields the CPU to another thread preventing the overall stall. Only the thread that originally performed the system call will block until the result is available, and the remaining threads will successively yield the CPU generally in a round robin fashion.

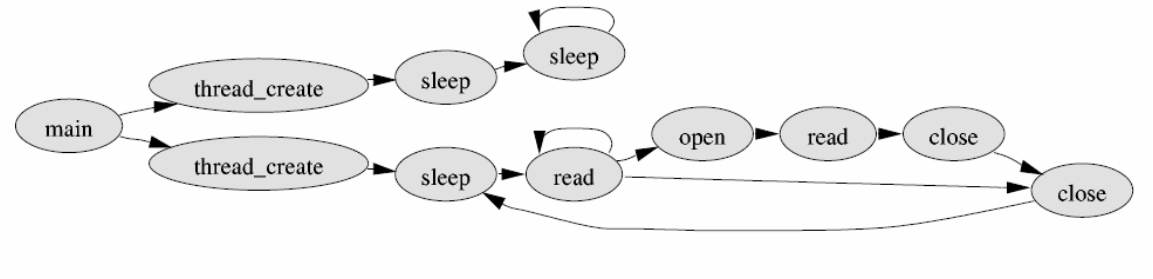

Figure 4 – Sample blocking graph for the Knot web-server [BCZNB03]

The application source is again analyzed in order to create the blocking graph that contains the information about the places where the threads of the program might block (see Figure 4). This is a better approach than retaining the instruction pointer of the blocking point because the transitions between blocking points allow for better dynamic prediction of the system behavior. The blocking graph allows the introduction of another novelty: a system resource monitor that keeps track of the past and current CPU, memory and other resources utilization (e.g., file descriptors) and tries to predict their future usage level.

At runtime, the system monitor will dynamically label each edge of the blocking graph with the exponentially weighted average time for the transition between the corresponding nodes (edge transition time). The nodes will also keep averages for the times spent on the edges outgoing from that node and information about resource changes that may indicate if the next node will decrease or increase certain resource utilization level.

Existing event-based system prioritize statically the event handlers and even SEDA [WCB01] is limited to event-handler queue length for scheduling the next handler. Capriccio goes further with resource aware scheduling based on the information gathered at runtime via the resource monitor and the weighted blocking graph. The strategy for optimal performance with Capriccio is to increase for each resource its utilization level until it reaches maximum performance and then throttle back by scheduling the threads that release that resource. When the resource utilization is low, the scheduler executes preferentially the threads that use the resource, and the other way around in the case of high utilization, it schedules preferentially the threads that release the resource before the ones that use the resource.

Detecting the maximum utilization level for each resource is however very difficult. With Capriccio, the level is determined by heuristics based on number of threads created and destroyed, number of files opened and closed, and early signs of thrashing (high page fault rate). In the case of normal web applications such as Apache, the procedure is sufficient and

assures a high performance level. However, in other cases and especially in multi processor systems or with virtual memory on disk this metrics will not predict the best utilization levels.

In addition, as in the case of event-based systems that rely on cooperative multitasking, we must detect the points in the program where threads should yield the processor and this is a time consuming task. In order to assist the programmer we developed a special profiling tool that provides access to internal data such as weighted blocking call. Another possible solution not implemented yet is to map the user threads on kernel threads.

6.3 Multi-processor Support

Multi processor systems bring more hardware performance that not always is used by the user level software mostly because the atomicity principle at the base of cooperative multi-threading is no longer valid. The preemptive threading model that applies for the kernel threads has to be adapted to the user level threading packages with an M:N mapping that is difficult to handle and implement.

6.4 Compiler Assistance

In order to implement a viable high performance user level threading package it is crucial to perform compile time code analysis, in order to detect yielding points, stack allocation points and blocking calls that must be replaced. All this operations can be done automatically with minimal changes to existing compilers or via dedicated tools. Also by code analysis, it may be possible to predict certain behavior and make application specific optimizations or CPU balancing in the case of multi-processor systems.

With Capriccio it is possible to perform compiler assisted and feedback based dynamically tuning of the threading package via live states regarding stack memory space, wasted stack, resource utilization levels, blocking calls path, etc. In the future it is expected to be able to generate the weighted blocking graph at compile time and warnings about possible non-necessary stack allocation for temporary parameters that with minor changes to the application algorithm can lead to smaller memory footprint, static detection of possible race conditions [BCB03].

7 Latest Trend

Improvements in the threading packages, kernel threads and kernel to user space "zero-copy" operations, system calls efficiency for critical performance (accept, select, pool/epool etc.) are now a priority for high performance system development. We regard compiler assisted code analysis and optimizations as possible future performance improvements factors mostly for threaded based designs.

More and more systems now use hybrid design models where both events and threads coexist in order to benefit from the advantages offered by multi processor and code clarity aspects of the threading model and easier memory management without synchronization of the event model (e.g., Flash webserver [WCB99]).

A great deal of work is recently invested in the storage subsystem because having a threading or event based system that can open/close 100k connections without being able to sustain them has no use. Current approaches are going into reordering random access requests into sequential requests from close related regions that can benefit from aggressive pre-fetching and disk read ahead internal caching and physical hardware properties, direct storage to network kernel operations for sending local files as fast as possible without user level overhead, etc.

8 Bibliography

[AHTBD02] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, J. R. Douceur. *Cooperative Task Management Without Manual Stack Management (or, Event-driven Programming is Not the Opposite of Threaded Programming)*. In Proc. of the 2002 Usenix ATC, 2002

[BCB03] R. Behren, J. Condit, E. Brewer. *Why Events Are a Bad Idea (for high-concurency servers)*. In Proc. of HotOS IX, Hawaii, 2003

[BCZNB03] R. Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. *Capriccio: Scalable Threads for Internet Services*. In Proc. of the 19th Symposium on Operating System Principles (SOSP-19), New York, 2003

[BO01] T. Brecht, M. Ostrowski. *Exploring the performance of select-based Internet servers*. TR HPL-2001-314, HP Labs, 2001.

[BW73] D. G. Bobrow, B. Wegbreit. *A Model and Stack Implementation of Multiple Environments*. Communications of the ACM, 1973

[CB01] P. Cheng, G. E. Blelloch. *A Parallel, Real-Time Garbage Collector*. In Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation, 2001.

[CM01] A. Chandra, D. Mosberger. *Scalability of Linux event-dispatch mechanisms*. In Proc. of the 2001 USENIX Annual Technical Conference, Boston, 2001.

[DBRD91] R. P. Draves, B. N. Bershad, R. F. Rashid, R. W. Dean. *Using Continuations to Implement Thread Management and Communication in Operating Systems*. In Proc. of the 13th ACM Symposium on Operating Systems Principle, ACM SIGOPS, 1991.

[DK] Dan Kegel's The C10K problem: *http://www.kegel.com/c10k.html*

[FCEZ03] R. Fowler, A. Cox, S. Elnikety, W. Zwaenepoel. *Using Performance Reflection in Systems Software.* In Proc. of the 2003 HotOS Workshop, 2003.

[GSC95] S. C. Goldstein, K. E. Schauser, D. E. Culler. *Lazy Threads, Stacklets, and Synchronizers: Enabling Primitives for Compiling Parallel Languages*. In Third

Workshop on Langauges, Compilers, and Run-Time Systems for Scalable Computers, 1995.

[JD] Jeff Darcy's online notes on high-performance server design strategies: *http://pl.atyp.us/content/tech/servers.html*

[LN78] H. C. Lauer, R. M. Needham. *On the Duality of Operating System Structures*. In Proc. Second International Symposium on Operating Systems, IR1A, 1978

[Ous96] J. Ousterhout. *Why Threads Are a Bad Idea (for most purposes)*. USENIX Technical Conference, 1996

[PDZ99] V. S. Pai, P. Druschel, W. Zwaenepoel. *Flash: An efficient and Portable Web Server*. In Proc. of the USENIX 1999 Annual Technical Conference, California, 1999

[PG95] W. Pang, S.D. Goodwin. *An Algorithm for Solving Constraint-Satisfaction Problems*. TR 95-04, ISBN 0-7731-0302-3, 1995

[ST] State Threads: *http://state-threads.sourceforge.net/docs/st.html*

[WCB01] M. Welsh, D. E. Culler, E. A. Brewer. *SEDA: An architecture for well-conditioned, scalable Internet services*. In Symposium on Operating Systems Principles, 2001