Matthew Roper
June 2008
Computer Science

Application-specific Thread Schedulers

## Abstract

This dissertation describes CATAPULTS, a domain-specific language for creating and testing application-specific user-level thread schedulers. Using a domain-specific language to write user-level thread schedulers provides three advantages. First, it modularizes the thread scheduler, making it easy to plug in and experiment with different thread scheduling strategies. Second, using a domain-specific language for scheduling code helps prevent several of the common programming mistakes that are easy to make when developing thread schedulers. Finally, the CATAPULTS translator has multiple backends that generate code for different languages and libraries. This makes it easy to prototype an application in a high-level language and then later port it to a low-level language; the CATAPULTS translator will take care of generating the appropriate code for both the prototype and the final version of the program from a single scheduler specification. Using CATAPULTS, we have been able to improve the performance of applications in many areas of computing, including Internet servers, embedded systems, and distributed systems.

Professor Ronald A. Olsson
Dissertation Committee Chair

**Application-specific Thread Schedulers**

By

MATTHEW ROPER
B.S. (University of California, Davis) 2002

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

---

Professor Ronald A. Olsson, Chair

---

Professor Premkumar T. Devanbu

---

Professor Raju Pandey

Committee in charge

2008

# Contents

# List of Figures

# List of Tables

# Acknowledgments

The path to a PhD is a long and difficult journey, and my success would never have been possible without the help and support of numerous people. My first and most important acknowledgment must go to my advisor, Dr. Ron Olsson whose support and guidance has been instrumental to my success. Dr. Olsson has always been available to meet with on short notice and has spent countless hours carefully reading over and responding to my work. His attention to detail is unparalleled, and his advice has always been incredibly sharp and insightful. I am very grateful to have had an advisor willing to invest so much time and effort into my success.

I would also like to thank my fellow graduate students in the Systems Lab. Their energy and enthusiasm has been a great inspiration. Special thanks to current students Jeff Wu, Gary Wassermann, and Chris Bird, and to former students Stoney Jackson, Brian Toone, and Eric Wohlstader for all the feedback and support provided over the years.

I also owe a huge debt of gratitude to all of my triathlon training partners and teammates. Athletics were an excellent balance to the academic side of my life and having friends to push me in the pool, on the bike, or out on a long run has a much larger impact on academic success than most people realize. Special thanks to Bryan Pro, Sinclair Yeh, Brandon Zipp, Amy Snodgrass, Max Biessmann, Adam Schaal, and Matt Davie for all of your support through the years and for making me remember the athletic and social sides of my life when I got bogged down in my research.

Also, a large thank you to Jason Cheung, Glen Sanford, Jeff Yuen, and Eric Farraro for their undergraduate research work on the CATAPULTS project. Their work and ideas added a lot to the CATAPULTS project and their patience with the never-ending stream of bugs they uncovered was much appreciated.

Finally, I owe a huge thanks to my parents for supporting my decision to pursue a PhD and for their patience as I sometimes dropped off the face of the earth for months at a time. Their love and encouragement through the long years was crucial to my success.

## Abstract

This dissertation describes CATAPULTS, a domain-specific language for creating and testing application-specific user-level thread schedulers. Using a domain-specific language to write user-level thread schedulers provides three advantages. First, it modularizes the thread scheduler, making it easy to plug in and experiment with different thread scheduling strategies. Second, using a domain-specific language for scheduling code helps prevent several of the common programming mistakes that are easy to make when developing thread schedulers. Finally, the CATAPULTS translator has multiple backends that generate code for different languages and libraries. This makes it easy to prototype an application in a high-level language and then later port it to a low-level language; the CATAPULTS translator will take care of generating the appropriate code for both the prototype and the final version of the program from a single scheduler specification. Using CATAPULTS, we have been able to improve the performance of applications in many areas of computing, including Internet servers, embedded systems, and distributed systems.


Professor Ronald A. Olsson
Dissertation Committee Chair

# Chapter 1

# Introduction

Multi-threaded application designs have become a very popular model for many types of applications. In general purpose computing, multi-threaded designs are often employed by highly concurrent Internet servers such as Apache 2.0 web server [1], MySQL [4], and OpenLDAP [5]. Threads are a popular model for these types of applications because they offer an intuitive programming style, similar to multi-process applications, while providing performance similar to that of event-driven applications. Although highly concurrent multi-threaded applications scale much better than their multi-process counterparts, performance can suffer if the algorithm used to schedule the multitude of threads makes poor decisions. Mutlithreaded designs are also widely used in embedded systems where control software is generally responsible for handling several concurrent tasks (e.g., driving different pieces of hardware while performing background calculations during spare computing cycles). This model is intuitive to program in because it allows each task to be programmed in relative isolation and makes it easy to follow the flow of control inside the task. Threads can either be scheduled cooperatively, where each thread has control of the processor until it explicitly yields it, or preemptively, where context switches are triggered at regular intervals by a timer interrupt. Regardless of which type of threading is used, the algorithm used to schedule threads can have a significant impact on the overall performance of the system.

With the limited resources available on an embedded system, the overhead of inefficient context switching is much more noticeable than it would be on a workstation, which has much more processing power.

This dissertation presents our system for developing application-specific schedulers, CATAPULTS (***C***reating ***A***nd ***T***esting ***AP***plication-specific ***U***ser ***L***evel ***T***hread ***S***chedulers). Modifying schedulers for threading libraries has historically been a difficult undertaking because scheduling is a cross-cutting issue and the relevant code is usually not collected as an easily pluggable component. Generating specialized schedulers for a specific application can improve performance not only by minimizing inappropriate context switches, but also by speeding up the scheduling algorithm itself; i.e., information such as thread priority or number of activations should only be tracked and processed if an application actually needs it to make good scheduling decisions. Unnecessary bookkeeping can be eliminated.

Our novel approach uses a domain-specific language for writing application-specific schedulers. It provides three major benefits. First, all scheduling code is collected into a single, replaceable component. The programmer need only fill in the body of various scheduling events (e.g., "new thread," "thread blocked on I/O," "thread terminated," etc.) in an aspect-oriented manner. Second, using a domain-specific language allows much better static analysis to be performed than if the scheduler were directly written in a lower-level language such as C (as most embedded applications are). For example, it is impossible to "lose" a reference to a thread using our language. Finally, using a domain-specific language allows multiple translation backends to be developed in order to target different threading libraries or programming languages.

Our work with CATAPULTS targets three major areas of computing. The first target is highly concurrent Internet servers (web, database, mail, etc.) that run on uniprocessor machines or virtual machines. These types of applications are very widely used and can gain a large benefit from the use of custom schedulers. Our second area of focus for CATAPULTS is applications for embedded systems. Custom schedulers allow such applications

to make optimal use of the very limited resources available to them, possibly allowing the use of cheaper hardware or reducing power usage. Our final target for CATAPULTS is "distributed" applications. We use the term "distributed" loosely to describe not only applications with components physically distributed across a network, but also systems that are logically distributed at the operating system level (e.g., a web server and database server running on the same physical machine that share an intelligent high-level scheduler).

The rest of this dissertation is organized as follows. Chapter 2 presents background material on the design of concurrent applications and threading alternatives. Chapter 3 describes the purpose, primary goals, and design of the CATAPULTS system. Chapter 4 introduces the CATAPULTS language through the use of an example. Chapter 5 provides details on the CATAPULTS domain-specific language. Chapter 6 discusses the front-end parser for the CATAPULTS language. Chapter 7 describes our implementation of CATAPULTS on unix-like platforms and our experience applying CATAPULTS to high traffic Internet servers. Chapter 8 examines our use of CATAPULTS on embedded systems. Chapter 9 explores our extension of CATAPULTS to distributed environments. Chapter 10 discusses common scheduler designs and tradeoffs we encountered in our work with CATAPULTS and explains the steps required to port CATAPULTS to new platforms. Chapter 11 surveys related threading research. Chapter 12 concludes the dissertation and covers possible areas of future work.

We introduced CATAPULTS and reported initial results of CATAPULTS for embedded systems in Reference [39]. Another paper, describing CATAPULTS' use on Internet servers, has also been submitted for publication. This dissertation contains additional details about our overall approach, a more extensive background review, more examples, further information on the CATAPULTS implementation, further experimental results, and additional discussion. It also presents our work with CATAPULTS on distributed systems.

# Chapter 2

# Background

## 2.1 Models For Highly Concurrent Applications

Concurrent applications, such as Internet servers or multimedia applications, generally follow one of three main design strategies: multi-process, event-driven, or multi-threaded [11, 23].

### 2.1.1 Multi-Process Applications

A multi-process application deals with concurrency by spawning multiple OS processes (e.g., by using the `fork()` system call on UNIX-like operating systems). Multi-process applications are popular for a number of reasons. First, all modern operating systems provide a means of spawning additional processes, so programs that make use of multiple processes are easy to port between operating systems; they require few, if any, modifications when moving to a new system. Second, multi-process applications provide a layer of safety by isolating the different parts of a program from each other. Since each process has its own address space, it is more difficult for a misbehaving process to corrupt the memory of other tasks and bring down the entire application; graceful recovery is easier to accomplish than in other concurrency models. Finally, since each process is a separate

kernel-level object, the application will get the full benefit of the kernel process scheduler and I/O scheduler, both of which have been heavily tweaked and optimized.

However, this model is the most heavy-weight of the possible strategies for dealing with concurrency. Process context switching is a (relatively) expensive operation and further overhead is incurred if separate processes need to communicate with each other or share global data. Furthermore, the memory requirements of a multi-process application are higher than in other models since some data must be duplicated.

### 2.1.2 Event-Driven Applications

Event-driven servers run as a single process and use non-blocking I/O and event notification mechanisms such as `select()` or `poll()` to internally multiplex control between active connections. Because there is no kernel-level context switching, event-driven servers are light-weight and perform very well with low memory and CPU usage, making them ideal for use on low-resource embedded systems. However, event-driven servers suffer from a number of limitations. First, event-driven applications run inside a single OS process, which means they can only submit a single I/O request to the kernel at a time.[1] This prevents them from taking advantage of the operating system's drive head scheduling or multiple hard drives, thus damaging performance. Furthermore, Internet servers such as web servers often need to handle a large number of open network connections; since each network connection requires a file descriptor, the supply of descriptors available to a single process can be exhausted quickly. Another factor that can limit the performance of event-driven servers is the event notification model used; most event-driven servers use `select()`, which does not scale well over large numbers of connections (although it is the most portable mechanism). Use of less-portable event models (such as epoll on Linux [33] or Kqueue on FreeBSD [32]) can provide much better performance. Finally, event-driven

[1] The use of asynchronous I/O (AIO) could overcome this limitation, but at this time, AIO is still immature and poorly supported by many popular operating systems.

applications are difficult to develop and debug since the control flow of a single task is now distributed throughout the program.

### 2.1.3 Multi-threaded Applications

The use of threads is the third main concurrency model and provides a compromise between the ease of programming of process-based applications and the high performance of event-driven applications. Threads use fewer system resources and usually have faster context switches than processes, which allows them to scale better as the concurrency of a system increases; the performance of threaded applications can even match or exceed the performance of event-driven applications [15]. Threads also provide a programming model similar to that of process-based applications, which makes threaded applications much easier to develop and maintain than event-based applications. Threads can be further subdivided into kernel-level threads and user-level threads, as discussed in Section 2.2.

## 2.2 Types of Threads

Threads can be divided into two categories: kernel level threads and user-level threads. Kernel-level threads, or "Light Weight Processes" as they are sometimes called, are created by system calls into the OS kernel and are scheduled preemptively by the kernel. This provides some of the same benefits as using processes; each thread can perform I/O or other blocking operations without suspending the entire application, and threads are scheduled with the operating system's highly tuned scheduler (which also allows them to be distributed among multiple processors). Unlike processes, kernel threads share an address space, which makes inter-thread communication easier (although thread synchronization is required to prevent race conditions). The disadvantage of using kernel-level threads is that each context switch requires a transition into kernel code and then back again, which is a relatively expensive operation. Even though the kernel scheduler is highly tuned for opti-

mal system performance, it does not have any knowledge of the application-level semantics and can often make poor decisions about which of a program's threads to run next.

User-level threads differ from kernel-level threads in that they are completely transparent to the operating system kernel. This means that all context switches are performed in user space (either preemptively or cooperatively) by a separate library-specific scheduler without kernel intervention, which results in a significant performance gain. However this performance benefit comes at a price: user-level threads run inside a single kernel process so they cannot be distributed among multiple processors; furthermore, I/O or other blocking system calls must be wrapped or replaced with non-blocking versions so that a single thread does not block the entire application. Despite these limitations (which they share with event-driven applications), user-level threads are ideal for highly concurrent applications that are meant to run on uniprocessor systems.

Threading implementations should not be confused with threading interfaces (API's). The most well-known threading API is Pthreads, the POSIX threading API [7]. Pthreads can be implemented either by an operating system kernel, or by a user space threading library. For example, Linux 2.4 used a thread implementation called LinuxThreads [45] and exposed (via glibc) the Pthreads API to user applications. In Linux 2.6, the threading implementation was replaced with the Native POSIX Thread Library (NPTL) [21], but the interface with applications remained the Pthreads API. And user space libraries can also provide implementations of the Pthreads API; GNU Pth [22], a popular user space threading library, can also be compiled with a Pthreads API, which makes it easy to run multi-threaded applications on different thread implementations.

## 2.3 Importance of User-level Threads

Although most modern operating systems provide an interface for kernel-based threading, user-level threading is still an important area of research. Not only is user-level thread-

ing required for optimal performance of multithreaded applications (as shown by [15]), it is also the only form of threading available on some types of embedded systems (some of which do not even have a separate OS capable of managing tasks). Finally, the concepts developed for application-specific scheduling via user-level threading may also be applied to kernel threaded systems through the use of Loadable Kernel Modules (as described in Chapter 12).

It is important to note that unlike kernel-level threads, user-level threading libraries can usually only utilize a single processor. However many servers on the Internet (web, mail, etc.) are still run on single uniprocessor machines. If the traffic expected for such a server is low, the server may even be hosted on a machine that is a generation or two old and has been retired from its previous purpose. There is still a lot of interest in obtaining maximum performance from such aging hardware as is evident from the high demand for "light" operating systems and server software.

Although multi-core and SMP systems are becoming much more common for general computing and high-end Internet servers, this apparent trend does not preclude the use of user-level threading on such systems. Only very busy Internet servers truly require multi-core or SMP systems to operate effectively. Instead, this trend towards multi-processing has helped boost the adoption of server virtualization [29] as a platform for small or medium-sized Internet servers. Technologies like User Mode Linux [19] and the Xen virtual machine monitor [12] make it possible for several functionally separate Internet servers to be hosted on a single physical host machine. In these situations, each server hosted on a given physical machine (which is quite often multi-core or SMP) usually interfaces with a single virtual CPU. Such virtualization is very popular for low- and mid-level Internet server hosting since it provides much better hardware utilization than running several independent physical servers. Since applications running in these virtual environments are usually presented with just a single virtual CPU, user-level threading is an excellent programming model, despite the multiprocessing nature of the true physical machine.

Finally, true multi-processor systems may still utilize user-level threading effectively by forking an OS-level process for each processor in the system. Each of these processes may contain its own user-level threading subsystem. This multi-process approach is covered by CATAPULTS' model of "distributed" applications, as described in Chapter 9, and custom schedulers can still be developed to improve the performance of such systems.

# Chapter 3

# Design

The CATAPULTS scheduling language was designed with three major goals in mind: modularization and pluggability of scheduling logic, prevention of common programming errors encountered in schedulers, and portability across different scheduling libraries with different capabilities. This chapter discusses these three goals. A fourth important goal, good performance, is implicit in the design decisions made for CATAPULTS. The use of CATAPULTS to achieve this performance gain is still an optimization with a tradeoff though: although custom schedulers can result in higher performance, they do require additional work to write and test. This tradeoff is explored in more depth in Section 10.2.

## 3.1 Modularization

One reason thread schedulers are so hard to write is because scheduling logic is spread throughout the scheduling library. The more complex a scheduling library is, the more places scheduling code will have to be modified. For example, although the GNU Pth threading library isolates its main scheduling routine in a file `pth_sched.c`, making any large changes to the Pth scheduler (such as replacing Pth's new thread queue, ready queue, etc. with different data structures and thread organizations) can require code modifications to more than 20 files. This is because other routines in the Pth library (e.g., I/O functions

that place blocked threads onto a waiting queue) make assumptions about how threads will be scheduled and manipulate the scheduling data structures directly. Furthermore, if the developer wishes to actually change the data structures used to store threads (e.g., add a new queue for threads of a specific type), the modifications required become even more invasive. Simpler threading libraries, such as those used by embedded systems, may require fewer changes (especially if they do not handle other programming concerns such as I/O and event handling), but replacing a scheduler is usually still non-trivial since all uses of the scheduling data structures must be tracked down and possibly modified.

With CATAPULTS we overcome this problem by allowing the developer to write the scheduler specification independently from the rest of the threading library. The CATAPULTS translator will then weave the user-specified scheduling code into the rest of the threading library to create a version of the library that is specialized for the specific application. Thus, it is very easy to try out different scheduling strategies.

## 3.2 Error Prevention

User level threading libraries are often written in C since they generally need low-level access to operating system facilities such as signal processing or event handling. On embedded systems, C and assembly are common languages for threading libraries since these are the languages that are most often used for application development. Although such low-level languages are very powerful, they do very little to prevent programming errors, especially when manipulating complex data structures via pointers. When such errors occur in thread schedulers, they often take the form of a thread coexisting on two different data structures at once (essentially duplicating a thread) or of a thread's reference being "lost" by the scheduler. Even in higher-level languages these types of mistakes are often easy to make and they are often very hard to track down and debug since the exact scheduling conditions that trigger the bug may not be easy to reproduce.

CATAPULTS provides a small set of data structures for storing collections of threads: threadrefs, stacks, queues, and lists that are sortable on any thread attribute (e.g., priority). Although these are all very simple container types, they have proven sufficient for most schedulers in our experience. All of these containers are unbounded except for threadrefs, which are bounded containers with a single slot. For convenience, individual threadrefs can be grouped into arrays, but each element of the array must be accessed directly; the array itself is not considered to be a thread collection. CATAPULTS enforces the invariant that each thread in the system is contained in exactly one collection at any time; this is a strength of CATAPULTS because thread references can never be duplicated or lost due to programmer error (although they can be explicitly destroyed when no longer needed). The only way to add or remove a thread from a container is to use the thread transfer operator, whose syntax is `src => dest;`. Each type of thread container has predefined logic that specifies how threads are inserted and removed by the transfer statement (e.g., removal occurs at the end of queues, but at the beginning of stacks). When this transfer operator is encountered in a scheduler specification, the CATAPULTS translator attempts to verify statically that there will be at least one element in the source container; if this cannot be guaranteed, the CATAPULTS translator inserts runtime assertions into the generated code. Similar checks are made to ensure that a bare thread reference used as the destination of a transfer statement does not already contain a thread. All thread transfer operations fall into one of four cases and cause the following types of checks to be performed:

**threadref => threadref** Attempt to statically determine that the source threadref is full and that the target threadref is empty. If either of these cannot be determined with certainty, runtime assertions are inserted into the generated code.

**threadref => unbounded container** Attempt to statically determine that the source threadref is full. If unable to determine statically, a runtime assertion is inserted into the generated code.

**unbounded container => threadref** Attempt to statically determine that the source container contains at least one thread and that the target threadref is empty. If either of these cannot be determined with certainty, runtime assertions are inserted into the generated code.

**unbounded container => unbounded container** Attempt to statically determine that the source container contains at least one thread. If unable to determine statically, a runtime assertion is inserted into the generated code.

This syntax is slightly restrictive in the sense that it is not possible to transfer the entire contents of one container to another container without an *O(n)* loop. A scheduler hand-written in C could implement an entire container transfer in `O(1)` simply by swapping pointers. Fortunately, we find this situation to be very uncommon in practice. We also may be able to improve static analysis in the future to detect and optimize entire container transfers for this case.

It should be noted that due to CATAPULTS' use of callback-like event and query handlers, the empty or full status of a container can only be inferred intra-handler and not inter-handler. Since event handlers can be called in any order, the contents of all containers (with the exception of threadrefs used as parameters to a handler) are completely unknown at the start of an event handler. As thread transfers are performed, it will become statically apparent that some containers are not empty (i.e., they have had threads transferred into them) or that some threadrefs are definitely empty (they have had a thread transferred out of them). So in general, less than half of this kind of container checks can be done statically at compile time — only when a container has been previously operated on by the current event or query handler can any information about its contents be inferred. For situations where errors must be detected at runtime, the use of invariant blocks (described in Section 5.5) can help quickly track down the original source of the error.

## 3.3 Portability

One of the primary goals of CATAPULTS is to make it as portable and retargettable as possible. CATAPULTS is not restricted to generating code for any one threading library; different code generation modules can be plugged in to allow generation of scheduling code for different libraries or even different languages. At the moment we have mature backend code generators for Dynamic C (a C-like language with threading features that is used to program ZWorld's embedded Rabbit controllers [2]) and GNU Pth [22] (a powerful cooperative threading library for PC's); writing more backends for other languages should be equally straightforward.

The fact that CATAPULTS can compile to different target languages and libraries and has backends for both embedded systems and regular PC's is especially advantageous when simulating or prototyping a system on a PC and then re-writing a final version on the actual embedded hardware. CATAPULTS allows the programmer to develop a scheduler once and then (assuming a CATAPULTS code generation module exists for both languages), simply recompile to generate schedulers for both the prototype and final system, even though they are using different languages and threading libraries.

Ideally, CATAPULTS would be able recompile schedulers for different threading packages with no modifications to the scheduler specification at all. However, since CATAPULTS allows programmers to write callback routines for various scheduling events, it may be necessary to add code to the scheduler specification when switching to a more featureful output module. For example, a scheduler developed for use with Dynamic C need only specify callback code for a basic set of thread events (thread creation, thread selection, etc.). If that scheduler specification is then used to generate a scheduler for a more advanced threading library, such as Pth, additional code will need to be written to specify what actions to perform on Pth's more advanced scheduling events (e.g., OS signal received, I/O operation complete, etc.). Each CATAPULTS backend code generation module includes a list of the scheduling events that are required in order to create a complete

scheduler; if a code generation module is used with a scheduler specification that does not include one or more of the required events, an error will be returned and translation will stop.

Although both of the primary backends we have developed so far have used a cooperative approach to task switching, CATAPULTS is also applicable in preemptive environments. Schedulers for preemptive threading libraries take the same form as those for cooperative libraries except that they require the addition of a handful of additional event/query handlers to deal with the preemptive aspects of the library (e.g., "quantum expired" event, "timeslice remaining" query, etc.). An undergraduate researcher working with us developed a preliminary CATAPULTS backend for the preemptive Wth threading library[1] over a three month academic term. Although this backend did not receive heavy testing or full debugging, it showed that the development of backends for preemptive threading libraries is no more difficult than developing backends for cooperative libraries.

[1] Wth was a user-space threading library that performed preemptive task switching through the use of the `alarm()` system call. It was not entirely stable at the time we were working with it and the project appears to have been abandoned now. The project website no longer exists.

# Chapter 4

# An Introductory Example of a CATAPULTS Scheduler

CATAPULTS is most easily introduced by providing an example of applying it to a simple, hypothetical multi-threaded application: the embedded control system of a weather monitoring station. The application has to monitor several temperature sensors (which have to be checked with different frequencies), drive a display that changes when the temperature reaches a certain threshold, and perform various calculations while the hardware is idle. Such a situation is relatively easy to model in a multi-threaded application: one thread is assigned to each temperature sensor, one thread drives the output display, and one or more threads perform miscellaneous calculations during the processor's idle time.

Control systems of this form are common applications for languages on embedded systems such as Dynamic C [2], an extended subset of C that runs on Z-World's 8-bit Rabbit processors (Chapter 8 discusses Dynamic C and our implementation of CATAPULTS on it). Although straightforward to implement, a standard Dynamic C implementation as described would fail to utilize the processor fully because Dynamic C's native thread scheduler uses a simple first come, first serve algorithm. Even though some threads do not need to run as often as other threads or only really need to run when certain application-level con-

ditions occur, the Dynamic C scheduler has no such knowledge. Its inefficient scheduling results in unnecessary context switches and additional overhead. In our weather monitoring example, the "slow" sensors will be queried for information as often as the "fast" sensors, even though they will not be ready to report information each time.

Using CATAPULTS can make such an application more efficient. It allows the programmer to quickly and easily create a thread scheduler tailored specifically for this application. Figures 4.1–4.8 show the scheduler specification (some minor details are omitted to save space).

Our example scheduler begins with a thread definition section, shown in Figure 4.1. It specifies what attributes the scheduler should track for each thread. In this example, only a single attribute ("state") is declared to track the status of a thread (i.e., whether the thread is new, running, suspended, blocked on I/O, etc.).

```
thread {
  int state; // new, running, suspended, etc.
}
```

Figure 4.1: Thread attribute declarations

Next, the scheduler declares which per-thread application variables should be imported into the scheduler. Importing an application variable into the scheduler allows the scheduler to monitor the variable for changes made by the application and also allows the scheduler to modify the variable's contents (which is a useful way of communicating information back to the application). Per-thread variables imported this way can be referenced exactly like regular thread attributes in event handler code. Application variable imports are discussed in depth in Section 5.2. Figure 4.2 illustrates the per-thread imports for our example; a single application variable ("threadclass") is imported, which allows the scheduler to determine the scheduling class ("slow sensor", "display", etc.) to which a given thread belongs.

The scheduler specification must also include declarations for any global objects used

```
threadimports {
  // possible values are thread class constants
  // defined in data section
  int threadclass default 0;
}
```

Figure 4.2: Thread import declarations

by the scheduler, including both global variables and constants of primitive types (i.e., integers and floats) and thread containers (queues, stacks, etc., provided by CATAPULTS; see Section 5.1).

Figure 4.3 shows the data declaration section for our example scheduler. Several thread containers are declared to hold different classes of threads: new (i.e., just created) threads are placed on a stack (`NQ`), sensor threads are divided depending on their speed between two queues, a thread reference is used to hold the single thread that drives the display, and another queue is used to hold the calculation threads. Regular variables of primitive types (just integers in this case) are also defined here to keep track of the last time a thread of a specific class ran, and constants are defined for the different scheduling classes to which a thread can belong.

```
data {
  threadref current;       // current thread
  threadref next;          // next thread (named yield)
  stack NQ;                // new threads
  queue standard_sensors; // sensors
  queue slow_sensors;      // sensors monitored
                           //   less frequently
  threadref display;       // display driver
  queue calculations;      // calculation threads
  // Last time various thread types ran
  int last_display, last_sensor1, last_sensor2;

  const int UNKNOWNCLASS = 0,
            SENSOR1CLASS = 1, SENSOR2CLASS = 2,
            DISPLAYCLASS = 3, CALCCLASS    = 4;
  }
```

Figure 4.3: Global data declarations

Just as a scheduler may need to import thread-specific attributes from the application, it may also need to monitor or update regular (global) application variables. For CATAPULTS to link a general application variable with an identifier in the scheduler, the imported variable must be declared in an `imports` block, along with a default value to use in case the application does not register the variable or the variable needs to be used by the scheduler before the application has a chance to register it. In our example, a single global variable ("temperature") is imported from the application. This variable will be used later, in the scheduler's event handlers, to determine whether or not the display output thread should be run.

```
imports {
  int temperature default 0;
}
```

Figure 4.4: Application variable imports

The remainder of the scheduler definition consists of event and query handlers. These handlers, which resemble C functions, are callbacks that the base threading library has been modified to call when it needs to perform a scheduling action or get information from the scheduler (see Chapter 8). The difference between an event handler and a query handler is the type of action performed. Event handlers are used when the base threading library is directing the scheduler to perform a specific action (e.g., "suspend this thread"). Event handlers are intended to perform side effects by manipulating the scheduler's global data structures; they return no value. In contrast, query handlers are used when the internals of the base threading library need to know something about the scheduler (e.g., "how many threads are currently in the system?"); query handlers return a value and must not have any side effects. Figures 4.5–4.8 contain a subset of the example scheduler's event and query handlers (the full set of event and query handlers is not reproduced here to save space).

After writing an entire specification, such as that in Figures 4.1–4.8, the developer then runs the CATAPULTS translator on the specification. It produces a scheduler targeted for

```
event init {
  last_display = 0;
}

event newthread(t) {
  t => NQ;    // Place t on 'new thread' queue
}
```

Figure 4.5: Event handlers to initialize the scheduler and handle new thread creation events

a particular backend. The developer then links that scheduler together with the application code.

If the developer decided to prototype/simulate the system on a regular PC before actually developing the embedded Dynamic C version, the scheduler specification could be passed through a different CATAPULTS backend to generate scheduling code for whatever language and library was being used for the prototype.

```
event schedule {
  threadref tmp;

  // Move new threads to their appropriate containers
  // if we know what type of thread they are yet.
  foreach tmp in NQ {
    if (tmp.threadclass == SENSOR1CLASS)
      tmp => standard_sensors;
    else if (tmp.threadclass == SENSOR2CLASS)
      tmp => slow_sensors;
    else if (tmp.threadclass == DISPLAYCLASS)
      tmp => display;
    else if (tmp.threadclass == CALCCLASS)
      tmp => calculations;
  }

  // Update last run times
  last_display++; last_sensor1++; last_sensor2++;

  // Determine next thread to run:
  //  - run target of named yield, if any
  //  - run display if temperature >= 100 and display
  //    hasn't been updated in over 10 ticks
  //  - run regular sensor if none run in 3 ticks
  //  - run slow sensor if none run in 6 ticks
  //  - else run calculation thread

  // |next| = size of next  (|next| is 1 or 0 here)
  if (|next| == 1) {
    next => current;
  } else if (temperature>=100 && last_display>10) {
    display => current;
    last_display = 0;
  } else if (last_sensor1>3 && |standard_sensors|>0) {
    standard_sensors => current;
    last_sensor1 = 0;
  } else if (last_sensor2>6 && |slow_sensors|>0) {
    slow_sensors => current;
    last_sensor2 = 0;
  } else {
    calculations => current;
  }

  dispatch current;
}
```

Figure 4.6: The main scheduling event handler

```
event switch_out(t) {
  if (t.threadclass == SENSOR1CLASS)
    t => standard_sensors;
  else if (t.threadclass == SENSOR2CLASS)
    t => slow_sensors;
  else if (t.threadclass == DISPLAYCLASS)
    t => display;
  else if (t.threadclass == CALCCLASS)
    t => calculations;
}

event set_next_thread(t) {
  t => next;
}
```

Figure 4.7: Event handlers for context switching away from a thread and performing a named yield to a specific thread

```
query threads_ready {
  return |standard_sensors| + |slow_sensors| +
         |display| + |calculations|;
}
```

Figure 4.8: An example query handler that returns to the base threading library the number of threads currently ready to run in the system

# Chapter 5

# Language Details

This chapter provides details about the CATAPULTS language.

## 5.1 Data Types

CATAPULTS provides a typical set of primitive types. In addition, CATAPULTS provides several thread container types for organizing the threads in the system: `queue`, `stack`, `pqueue`, `pstack`, and `threadref`. A `pqueue` (or `pstack`) is similar to a `queue` (or `stack`), but its threads are ordered by a user-specified key. A `threadref` can hold at most one thread. As mentioned in Section 3.2, all threads must be present in one and only one of the scheduler's thread containers at any time and the thread transfer operator is used to move threads between containers. Each container type has a designated insertion point and removal point that controls how the transfer statement manipulates the container. The container types provided by CATAPULTS are:

**threadref** A thread container with a slot for a single thread. Using an empty threadref as the source of a transfer operation or using a full threadref as the destination of a thread transfer are errors.

**stack** Threads are both inserted at and removed from the beginning of the list. This data

structure is unbounded, but attempting to transfer a thread out of an empty stack is an error.

**queue** Threads are inserted at the end of the list and removed from the beginning. As with the stack, this data structure is unbounded, but using an empty queue as the source of a transfer operation is an error.

**pqueue** Threads are maintained in a sorted order based on the pqueue's designated sort key. An inserted thread will be placed after all other threads with the same sort key. Threads are removed from the beginning of the list. This data structure is unbounded, but using an empty pqueue as the source of a transfer operation will result in an error.

**pstack** Identical to a pqueue, except that an inserted thread will be placed *before* all other threads with the same sort key.

Collection types are declared using `type var;` syntax (e.g., `queue ready_threads;` or `threadref current_thread;`) with the exception of `pstack`'s and `pqueue`'s. These two data types maintain a sorted collection of threads so their definition must also specify the sorting key. The syntax is *`type var`* `[reverse] sortable on` *`attribute`*`;` where *`attribute`* is one of the attributes declared in the thread definition section of the scheduler. The threads will be sorted such that the thread with the highest sort key will be removed first, unless the `reverse` keyword is specified in which case the thread with the lowest sort key will be first on the queue.

CATAPULTS also allows arrays of both primitive types and thread containers. Note that an array of threadrefs is not considered a container itself and cannot be used as the source or destination of the thread transfer operation; instead, a specific element of the array must be indexed directly.

## 5.2 Imported Application Variables

The primary goal of CATAPULTS is to not only make it easier to write new thread schedulers in general, but to allow the development of *application-specific* schedulers for the absolute maximum performance on a specific application. Since optimal scheduling decisions often require knowledge about the internal state of an application, CATAPULTS provides a means for applications to register their internal variables with the scheduler. Once a variable is registered with the scheduler, it is linked with a corresponding variable declared in the 'imports' section of the scheduler specification (Chapter 4). Any changes that the application makes to the variable will immediately be visible through the linked scheduler variable and vice versa.

Imported application variables are the most controversial feature of CATAPULTS since mixing application-level data with scheduler logic can be seen as a dangerous entanglement of separate system levels. This optional feature provides a tradeoff to the application programmer: it becomes harder to change the application without also making changes to the scheduler, but performance can be significantly improved by making use of application-level information.

As seen in Chapter 4, CATAPULTS allows two types of variables to be registered (imported) with the scheduler: general (global) application variables and per-thread instance variables. Registering general variables is useful for providing the scheduler with information about the status or load of the system as a whole; common examples include the number of open network connections in a multithreaded Internet server or the number of calculations completed in a scientific application. In contrast, registering per-thread instance variables with the scheduler is useful for tracking information that the application stores for each thread. Per-thread instance variables are useful not only for monitoring information that the application would be tracking anyway (e.g., the number of packets that have been processed on a network connection for an Internet server), but also for specifically directing scheduler behavior from the application, e.g., `threadclass` declared in

Figure 4.2 and used in Figures 4.6 and 4.7.

Although registering variables requires some modification to the base application and removes the transparency of CATAPULTS, the modifications required are minimal; only a single registration statement is necessary near the beginning of the program for each variable that is to be registered with the scheduler. As a real-world example, when we modified our existing CoW web server [26] to use a CATAPULTS-generated scheduler for GNU Pth (as described in Section 7.2.1), we only had to add the following code near the beginning of main:

```
// Catapults-specific:
//   register # of open files with scheduler.
pth_schedvar_register("noof", &cow_noof);
```

and the following code to the routine that spawns the thread pool:

```
// Register our per-thread information
// with the scheduler
pth_threadvar_register("isresponding",
    OFFSETOF(thrresponse, isresponding));
pth_threadvar_register("bytesleft",
    OFFSETOF(thrresponse, bytesleft));
...
// Create the Catapults scheduler data
scheddata = malloc(sizeof(thrresponse));
if (scheddata == NULL) {
  perror("Failed to allocate scheduler data");
  exit(errno);
}
scheddata->isresponding = 0;
scheddata->bytesleft = 0;
pth_set_userdata(tid, scheddata);
```

A similar number of lines of code would have to be changed for other typical applications.

If an application tries to link an application variable to a scheduler variable that does not exist in the currently loaded scheduler, the registration command will cause a fatal error or will be silently ignored, as per user-specified option. The silent choice makes it easier to swap in and out schedulers that make use of different application variables. For example, a web server may try to register the number of open network connections with the

scheduler, but if it is executed with a scheduler that does not make use of this information, the registration command will be ignored.

## 5.3 Verbatim Statements and Expressions

Although CATAPULTS' portability between target languages and libraries is generally considered beneficial, it is sometimes a disadvantage, such as when the programmer wishes to make a library or system call that is not exposed through the CATAPULTS language. For example, a scheduler that uses a random number generator to make some of its scheduling decisions will need some means of generating random numbers even though CATAPULTS does not have any instructions to do this. Likewise, if the programmer wishes to write some values to the screen while debugging, he will need some way of generating output since CATAPULTS does not include any output commands.

To overcome these limitations, CATAPULTS provides verbatim statements and verbatim expressions, which allow the programmer to include a block of code (either a statement-level block or a single expression) of the target language directly in the scheduler. Thus the programmer can express anything that could be coded in the target scheduler's programming language at the expense of some portability (i.e., the verbatim statements and verbatim expressions will need to be re-written for each target language/library to which the scheduler is compiled).

## 5.4 Scheduler Inheritance and Aspect-Oriented Programming

Although the optimal scheduling algorithm varies from program to program, quite often large parts of a scheduler specification are identical between several different schedulers. For example, custom schedulers for a web server and a database server may use different

logic for selecting the next thread to run, but share the same code for events like blocking on I/O or suspending a thread. CATAPULTS provides two features to help leverage this realization: scheduler inheritance and scheduler aspects.

CATAPULTS supports scheduler inheritance: schedulers can be derived from other schedulers in a standard object-oriented manner. Only events or queries that need to behave differently from those in the base scheduler need be implemented and the new implementation will completely replace the behavior of the base scheduler. If the overridden events and queries need to make use of new scheduler variables or data structures, those can be defined in a data section – the data section of the derived scheduler will be combined with that of the base scheduler. Figure 5.1 shows a simple example.

```
scheduler newsched extends basesched {
    data {
        int newvar = 0;
    }

    event schedule {
        /*
         * new scheduling logic here will override
         * that in basesched.sched
         */
    }

    /* All other events/queries are unchanged */
}
```

Figure 5.1: Example of scheduler inheritance.

CATAPULTS also supports simple aspect-oriented programming: additional code can be run before or after the existing implementation of an event. These aspects can either be defined in a derived scheduler (as described above) or in an external scheduler whose name is specified on the command line. In addition to allowing new schedulers to be created from existing ones, scheduler aspects can also be used to debug or perform measurements on existing schedulers. For example, one of the first aspect bundles that we developed added additional code to capture timestamps before and after performing scheduling events so

that the time consumed by a scheduler itself could be measured. By defining this bundle of aspects in a separate file, we can apply these measurements to any existing CATAPULTS scheduler simply by specifying the aspects' filename on the command line.

## 5.5 Invariant Blocks

Although CATAPULTS schedulers make it much easier to write error-free schedulers, they can not completely remove the possibility of logic errors in scheduler design or simple coding mistakes. Occasionally a developer will write a scheduler that, due to an oversight on his part, will eventually try to remove a thread from an empty thread container or will try to store a thread in an already filled thread reference. Ideally (as described in Section 3.2) these problems will be caught by static analysis at compile time, but if static analysis is unable to do so and the error must be detected at runtime it can be difficult to work backwards and figure out exactly what the source of the error was. The ultimate outcome (read from empty thread container or write to non-empty threadref) is often the end result of a logic error earlier in execution. It may be the result of count variables becoming out of sync with container contents, it may be an incorrect flag getting set on a thread, or it might be the result of a thread being placed in the wrong container due to incorrect programming logic.

To combat these errors and ensure that they are detected as early as possible in the execution of a scheduler if they cannot be discovered at compile time, CATAPULTS provides a mechanism called "invariant blocks." An invariant block is an optional block of code that follows the `data` and `import` sections of a scheduler, but precedes the event handlers. An invariant block specifies a specific container and provides a block of code that should be executed to test the system validity after any change to that container is performed. A scheduler may include multiple invariant blocks if desired. These tests, which take the form of assertions in the generated C code, are only generated when CATAPULTS is run in debug mode. An example of an invariant block is given in Figure 5.2.

```
invariant WQ {
  foreach t in WQ {
    if (t.state != WAITING)
      {* assert(0 && ``Failure: non-waiting thread on WQ''); *}
  }
}
```

Figure 5.2: An example invariant block

# Chapter 6

# CATAPULTS Front-end Details

This chapter describes the front end parser for the CATAPULTS language. The CATAPULTS translator is fairly straightforward and similar to other language translators. Section 6.1 describes the implementation of the frontend. Section 6.2 discusses the static analysis performed by the translator.

## 6.1 Implementation

The CATAPULTS translator is written in Python using PLY (Python Lex/Yacc) [14]. PLY is a parser generator library for Python that makes compiler development relatively easy. Parse rules are provided as BNF rules in function comments and are read by the PLY initialization routine using Python's reflection capabilities. An example of a rule for an `if` statement is shown in Figure 6.1.

The function implementing each rule takes an array `t` as a parameter. `t` contains one element per grammar symbol matched by the rule. For terminal symbols, the corresponding `t[`*i*`]` value is the literal representation of the token. For nonterminals, `t[`*i*`]` contains the result of the parse rule function for that symbol.

The CATAPULTS front-end defines a collection of classes, all derived from a generic `ASTNode` class, to represent nodes of the Abstract Syntax Tree (AST) for a scheduler.

```
def p_standard_selection_statement_1(self, t):
    'selection_statement : IF LPAREN expression RPAREN statement'
    t[0] = ASTIfStmt(t[3], t[5])

def p_standard_selection_statement_2(self, t):
    """selection_statement : IF LPAREN expression RPAREN statement
                           ELSE statement"""
    t[0] = ASTIfStmt(t[3], t[5], t[7])
```

Figure 6.1: Example parser rules for an `if` statement

Each parse rule function saves its child node parameters and produces an AST node object for the rule. In the example in Figure 6.1, both functions will expect an `ASTExpression` object as `t[3]` and `ASTStatement` objects as `t[5]` and `t[7]` and will generate an ASTIfStmt object for higher level rules.

Once the AST for an entire scheduler has been built (with an `ASTScheduler` node at the top of the tree), the CATAPULTS parser performs several passes over the tree to perform the following operations:

- Global variable resolution
- Local variable resolution
- Lint (type checking, static analysis)

## 6.2 Static Analysis

The translator uses very simple propagation-based static analysis to track the various invariants described in Section 3.2. Specifically, this static analysis is used to track the following information:

- presence or absence of threads on a container or in a thread reference
- failure to store a thread passed as a parameter into a permanent container in an event handler that requires this (e.g., `newthread(t)`)

- failure of a query handler to return a value
- failure of an event handler to produce a side effect
- code following a dispatch or return statement
- statically known variable values

As discussed in Section 5.1, CATAPULTS generates runtime assertions in the generated code for scheduler code whose behavior it cannot guarantee statically.

Some of the traits tracked by static analysis (such as whether a block of code saves the event handler parameter to a permanent container) can be true (“YES”), false (“NO”), or dependent on what branch of code gets taken (“MAYBE”). Figure 6.2 shows part of the static analysis routine for an `if` statement.

```
# Does this 'if' statement save the parameter of the event
# handler to a permanent container?
if not self.falsebranch:
  self.savesparam = self.truebranch.savesparam
else:
  if self.truebranch.savesparam == self.falsebranch.savesparam:
    self.savesparam = self.truebranch.savesparam
  else:
    self.savesparam = "MAYBE"
```

Figure 6.2: Example of static analysis to check whether an `if` statement saves an event handler parameter to a permanent container.

# Chapter 7

# CATAPULTS for Internet Servers

## 7.1 Implementation

GNU Pth [22] is a relatively advanced cooperative threading library written in C. In addition to providing traditional thread facilities (create thread, suspend thread, yield, etc.), Pth also provides thread-aware wrappers for I/O operations, per-thread signal handling, message ports, and more. The many advanced features of Pth make it a large, complex library and scheduling-related code is spread over Pth's many source files. To integrate CATAPULTS with Pth, we made modifications to allow Pth to load custom schedulers at application startup time from `.so` files. When an application linked against our modified version of Pth begins execution, it will check the contents of the `PTH_SCHED` environment variable. If this variable is not empty, `.so` will be appended to its contents and the resulting filename will be loaded via the `dlopen()` library call. If the `PTH_SCHED` variable is not set, the original, builtin scheduler will be used.

Loading schedulers at runtime from shared libraries introduces a slight amount of additional overhead since scheduling code that was embedded directly in the Pth library before is now replaced with an invocation of a function pointer that is loaded from a shared library. Fortunately we found the overhead of calling scheduling code via a function pointer rather

than executing it directly to be minimal and we measured no decrease in performance after instrumenting the Pth library.

The one disadvantage of using this shared library technique is that it poses a security risk for applications that run setuid. A malicious user could add any code he wanted to the scheduler after CATAPULTS has translated it to C code, but before it has been compiled into a `.so` library. This code would then be executed with target user's permission when the application tried to perform regular scheduling actions. This security problem is somewhat similar to the one that arises when `LD_PRELOAD` is honored for setuid programs (a situation that is no longer allowed by most modern operating systems) [43]. However, we expect that CATAPULTS will be most applicable to server-type applications; these types of applications generally do not need to be started by regular users and are unlikely to be marked as setuid.

## 7.2 Experience

This section describes our experience, including performance results, in using CATAPULTS to write custom schedulers for real-world Internet server applications.

### 7.2.1 CoW Proxy Server

The first Internet server to which we applied CATAPULTS scheduling was the CoW proxy server [26]. CoW (the "COoperative multithreaded Web server) can function either as a regular web server, or as a caching reverse proxy server for another web server. For the purpose of this experiment we ran CoW in proxy server mode; since pages are cached in RAM, this ensured that the workload did not become disk-bound. Nevertheless, similar results are obtained when CoW runs in regular web server mode as long as the web server's working set of requested documents can fit into the operating system's disk cache.

CoW, when run in proxy mode, utilizes: a main "accept" thread, which is responsible

for processing incoming network connections; a "backend" thread, which is responsible for fetching pages that are not found in the cache from the backend web server; and a pool of "handler" threads, each of which is responsible for processing a particular web request. CoW is written with the GNU Pth and when initially developed, CoW relied on the generic first-come, first-served scheduler built into the Pth library. Although this scheduler provided adequate performance (CoW's performance was comparable to that of other popular servers such as Apache [1] and Boa [20]), we identified a number of situations in which poor thread scheduling was damaging performance:

- If there are no idle handler threads remaining in the thread pool, switching to the "accept" thread is undesirable since no new threads can be accepted at this time.
- Likewise, if the number of open file descriptors has reached the OS-imposed limit, switching to the "accept" thread is useless since the `accept()` system call will fail until some of the file descriptors are freed.
- For static web requests, it is easy to estimate how close a handler is to completing the processing for a web request. Since performance can be improved by four or five times under a shortest-connection-first scheduling algorithm [18], using the default, progress-unaware scheduler results in a significant performance hit.

Using CATAPULTS, we developed a customized scheduler that addresses these issues and yields quicker response times. Appendix B shows the scheduler specification. During compilation, CATAPULTS was able to statically determine that 24 out of 32 (75%) thread transfer checks (as described in Section 3.2) were safe; no runtime assertions were inserted for these transfers. As the figures show, the application-specific scheduler for CoW is relatively simple. For each thread, the scheduler tracks what state the thread is in (running, suspended, etc.) and whether the thread is the main application thread (which accepts new connections and dispatches web requests), the backend thread, or a handler thread responsible for processing requests. It also makes use of application-level thread variables

that indicate whether a thread is currently in the process of responding to a web request and, if so, how many bytes it still has to return. Threads that are ready to run are divided between two lists – one sorted list for threads that are currently responding to web requests and one queue for threads that are performing other tasks, such as parsing a request. The list of responding threads is sorted by how many bytes are remaining in the response. Finally, CoW imports two application-level global variables to help make scheduling decisions: `noof`, which tracks the number of open file descriptors (necessary so the scheduler knows when no more network connections can be accepted), and `idlehands`, which tracks the number of idle handler threads remaining in the thread pool. (The use of application-level variables by the scheduler represents a design tradeoff between separation of concerns and performance increase potential; see Section 10.2 for details.)

The entire CATAPULTS scheduler specification consists of 169 lines of CATAPULTS code, including comments and whitespace. In addition, we added/modified about 20-30 lines of C code in the CoW application code in order to get it to run with the new CATAPULTS scheduler. These modifications, shown in Figure 7.1, consisted of registering application variables with the scheduler.

We benchmarked the performance benefit of using the CATAPULTS scheduler by running the httperf [35] HTTP benchmark against CoW, both with and without the CATAPULTS scheduler. There are many variables that can be tuned while benchmarking a web or proxy server; one very important variable to vary is the size of files being served. Different web sites will serve different types of content with varying file sizes; for example, sites that are primarily text will generally serve much smaller files than those with lots of multimedia documents or downloadable files. Serving larger files clearly places greater strain on a web server since additional computing power is required to read, copy, and transmit an entire web document. In order to analyze whether our CATAPULTS scheduler would allow a web server to serve larger files with less degradation of performance, we structured our experiment as follows. We had a set of 20 machines on the LAN each send

```
// near the beginning of main:
  //   Catapults-specific:
  //     register # open files with sched
  pth_schedvar_register("noof", &cow_noof);
  pth_schedvar_register("idlehands", &numidlehands);

// in the routine that spawns the thread pool:
  //   Register our per-thread information with the scheduler
  pth_threadvar_register("isresponding",
                         OFFSETOF(thrresponse, isresponding));
  pth_threadvar_register("bytesleft",
                         OFFSETOF(thrresponse, bytesleft));
  ...
  // Create the Catapults scheduler data
  scheddata = malloc(sizeof(thrresponse));
  if (scheddata == NULL) {
    perror("Failed to allocate sched data");
    exit(errno);
  }
  scheddata->isresponding = 0;
  scheddata->bytesleft = 0;
  pth_set_userdata(tid, scheddata);
```

Figure 7.1: C code added to CoW application code to register application variables with the scheduler.

web requests to the web server at various request rates until $1000 \times (20 \times rate)$ requests had been made. We measured how the response time varied as server load increased (since improving page response time is the primary purpose of a caching reverse proxy server). The results provided below reflect reflect the experiment run against a CoW server with a thread pool of size 512. The client machines used in the experiment were dual processor 2.8 GHz Pentium IV Xeon's with 1 GB of RAM, running hyperthreading-enabled Linux kernel 2.6.17 (as packaged by Fedora Core 5). The server machine used was a 866 MHz Pentium III with 256 MB of RAM, running Linux kernel 2.6.10 (as packaged by Debian). Although the network and client machines were open to other use, we ran our experiments at a time when other traffic would be minimal. The results show the median results of running the entire experiment five times.

Figure 7.2 shows our results. Our application-specific scheduler resulted in quicker

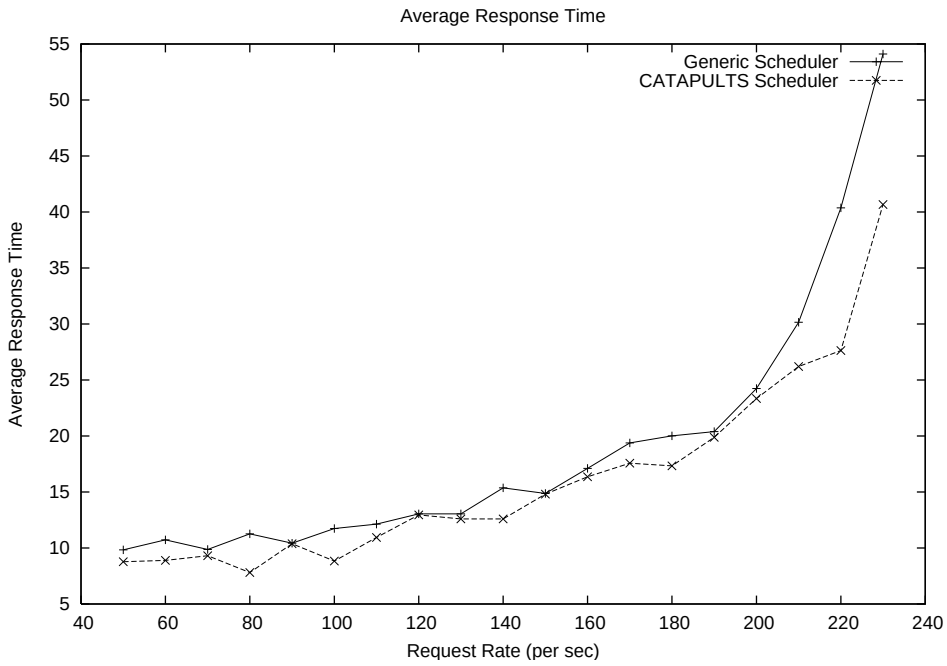



Figure 7.2: HTTP response time for CoW proxy server

response times during periods of heavy loads (reductions of about 13–32% at the highest request rates).

### 7.2.2 MySQL Database

We also developed a CATAPULTS scheduler for the MySQL database server [4]. MySQL is written using the pthreads API and can be compiled against Pth's implementation of pthreads. MySQL uses a separate thread per connection and also has some additional threads for advanced features, such as replication.

#### Initial Design

We initially set out to write a scheduler that would improve the performance of MySQL in general, regardless of the actual application making use of the database. By default MySQL uses the MyISAM table format, which performs table-level locking. A complex query that makes use of several database tables will require a significant number of table

locks (shared read locks for tables that are being read, exclusive write locks for tables being updated). These widespread locks are likely to prevent further progress in other threads that are performing simpler queries involving the same tables, so if these complex queries are given higher priority, they will finish quicker and unblock several other threads in the system.

The scheduler we developed based on this strategy maintained a single list of ready threads and sorted that list on the "locking factor" of the thread. We calculate the locking factor of the thread as $readlocks + 5 \times writelocks$. Readlocks, although not as important as writelocks, are far more numerous so we expected this expression to roughly balance the importance of writing over reading. The changes required to MySQL itself so that it can provide the data needed by the CATAPULTS scheduler were relatively simple. We simply created a per-thread structure to contain MySQL's write lock and read lock counts, updated those counts on lock/unlock operations, and then registered the values as per-thread imports. The code is similar to that we added to CoW (described in Section 7.2.1).

After making the changes described above and implementing the CATAPULTS scheduler, we benchmarked the system using DBT2 [6], an open-source implementation of the industry-standard TPC-C database benchmark. Database servers have an incredible number of settings that can be tuned for maximum performance (e.g., various internal buffer sizes or caching policies) so we just used the default server configuration. The system running the database server used in the experiment was a 2.8 GHz dual-processor Intel Xeon with 1 GB of RAM and running Linux kernel 2.4.21 as packaged by Red Hat Enterprise Linux. We found that our performance using the specialized MySQL scheduler was not noticeably better than using a regular first-come, first-served scheduler implemented in CATAPULTS. After some further exploration, we realized that it was not going to be possible to speed up single query operations in MySQL while using a cooperative threading library like Pth.

MySQL and other pthreads applications assume that their implementation of pthreads is preemptive and do not bother to explicitly yield at any point during execution. Pth forces

the yield on any kind of I/O operation, but MySQL also does a lot of CPU-intensive work as well (e.g., joining tables). Under a preemptive threading library, context switches would happen during this time and performance would benefit from a scheduler that could continue to run an important transaction or recognize the presence of a higher-priority query and switch directly to that. However Pth's cooperative multithreading causes these long stretches of CPU-based work to execute atomically, and it is only possible to switch to a higher priority thread when disk I/O occurs. Furthermore, it turns out that the I/O-based yields do not benefit performance as much as hoped either; even when using the largest test database that we could generate with our available diskspace (i.e., 25 warehouses under DBT2), the data being requested most often by queries was small enough to fit into either the OS disk cache or MySQL's application-level query cache. Thus, most of the I/O operations became non-blocking and also failed to result in thread yields. The effects of caching were very noticeable — benchmark performance rose from approximately 80 New Order Transactions Per Minute (NOTPM) when run against a "cold" database to about 250 NOTPM after running the benchmark just a few times.

**Final Design**

Although improving the general performance of MySQL was not successful with our initial design, we found that we were able to improve the performance of the actual target application making use of MySQL by using a different approach. Instead of trying to speed up single queries, we focused on identifying important application-level transactions and giving priority to all queries of those transactions. The DBT2 benchmark performs several different types of transactions and identifies the "new order" transactions as the important transactions. The benchmark measures how many New Order Transactions Per Minute (NOTPM) can be performed in the presence of the other, less important queries in the system, so our focus with the new MySQL scheduler was to specifically improve these transactions.

We made minor modifications to the MySQL command parsing/execution engine to expose the list of table and field names used in a query as global string variables. We then registered these strings as application imports so that the CATAPULTS scheduler could identify the first and last queries of a New Order transaction. Higher priority was given to threads determined to be executing a new order transaction, possibly at the expense of the unimportant queries. Although tailoring a tool to gain better performance on a specific benchmark is usually frowned upon, in this case it is really the whole point of the experiment. The performance of MySQL over all real-world situations is irrelevant — it is the performance of the end-user application that really matters, so that is what we need to tune our system for. In this case, the end-user application is the DBT2 benchmark, but the same approach would be used to write a scheduler for a real-world database application such as a large database-backed website.

The scheduler described above consists of 144 lines of CATAPULTS code, including comments and whitespace. The code, which appears in Appendix C, uses the CATAPULTS mechanisms in ways similar to the code for the CoW scheduler (Section 7.2.1). CATAPULTS was able to statically recognize 19 of the 26 thread transfer checks (73%) to be statically safe, as described in Section 3.2; no runtime checks had to be inserted for these transfers.

Table 7.1 summarizes our performance results. We obtain a 6.5% increase in performance by using the MySQL-specific scheduler as opposed to a generic, lightweight first-come, first-served scheduler. As before, we expect that the performance increase would be much more noticeable if a preemptive threading library were used instead of a cooperative library.

### Qualitative Results

The previous two sections described schedulers that we wrote for the CoW multi-threaded web server and the MySQL database server. In both cases, developing the sched-

Table 7.1: The average results from running DBT2 against MySQL a total of five times, for five minutes each (variances were small).

| Scheduler | New Order Transactions Per Minute (NOTPM) |
|---|---|
| Generic First-come, First-served | 242.46 |
| CATAPULTS MySQL scheduler | 258.35 |

ulers themselves once we had decided what scheduling algorithm to use was very easy; we were able to take the schedulers from idea to implementation in under an hour. Instead, what we found took a very long time was developing an understanding of the internal workings of an unfamiliar code base. Before developing the MySQL scheduler, we had to explore the internal architecture of the system and figure out where the system was tracking various pieces of information that we wanted to make use (e.g., number of locks, fields involved in a query, etc.). For CoW, an application that we wrote ourselves, this step was eliminated since we were the original authors of CoW and were already familiar with the internal architecture. In general, the application developer should have the knowledge about the structure of the application and some ideas about what kinds of thread scheduling might improve the application's performance.

Although our ease of developing schedulers with CATAPULTS could be partially influenced by the fact that we are the authors of the tool, we also received positive feedback from four undergraduate students who were not involved in the initial development of CATAPULTS. They were able to understand and start using the CATAPULTS language very quickly and each wrote several schedulers that are now included as part of the CATAPULTS test suite.

# Chapter 8

# CATAPULTS for Embedded Systems

While our Pth backend (Chapter 7) illustrated the the benefits of using CATAPULTS with a large complex, threading library, we also wanted to apply CATAPULTS in an embedded environment. For this purpose we chose to develop a CATAPULTS backend for Dynamic C, an extended subset of C that is used to program Z-World's 8-bit Rabbit devices.

## 8.1 Implementation

Dynamic C includes builtin cooperative multithreading in the form of costatements and cofunctions. Costatements and cofunctions allow a programmer to structure his program as a loop of state machines. Each costatement or cofunction can be considered a thread, but the system only allows a round-robin scheduling policy for the threads. Although it is possible to use tricks to accomplish dynamic scheduling in Dynamic C [37], doing so requires invasive changes to the application itself, which results in confusing code and does not integrate well with CATAPULTS. Instead we chose to integrate CATAPULTS with the cmthread.lib threading library that we had previously written for Dynamic C. cmthread.lib is a substitute for Dynamic C's language-level multithreading and provides an API that is more consistent with other popular threading API's such as Pth or Pthreads. cmthread.lib

also provides better performance in many cases.

Dynamic C applications run in an embedded environment with no operating system, so dynamically loading schedulers at runtime as our Pth implementation does is neither possible nor advantageous. Instead, our Dynamic C backend generates a custom version of the cmthread.lib library that contains the custom generated scheduling code inline. This approach has the additional advantage of eliminating the indirect function calls via function pointers used in the Pth backend; although the overhead of calling scheduling code was too small to measure on a regular PC, we expect that it would have played a much larger role in the embedded environments that Dynamic C is used on.

The modifications to cmthread.lib to make it work with CATAPULTS are minor: about 100 new lines of code were added to the original 457 lines. Because this new code is being generated by CATAPULTS, its formatting sometimes splits what would normally be one line of code over several. So, a fairer estimate is about 50 lines of new code. Also, a good portion of this code is functions that simply do callbacks.

## 8.2 Experience

In order to measure the benefit of using CATAPULTS on an embedded application, we simulated the weather monitoring station example described in Chapter 4. Since we do not have access to real weather monitoring hardware, we wrote a Dynamic C application with the appropriate control logic and replaced actual sensor and display hardware I/O with small loops. The CATAPULTS scheduler specification described in the example in Chapter 4 was used to control thread scheduling. The complete specification (including the minor details omitted in Figures 4.1-4.8) was a total of 174 lines of code and was translated into 546 lines of Dynamic C. In contrast, the original cmthread.lib library on which CATAPULTS' output is based is a total of 457 lines of Dynamic C code. Although space is a scarce resource on embedded systems, this size increase is quite reasonable considering

Table 8.1: Comparison of CATAPULTS threading library and generic cmthread.lib

| | Lines of Code | Size of Compiled Code | Duration of Simulation |
|---|---|---|---|
| Generic cmthread.lib | 457 | 21120 bytes | 76.508 sec |
| Generated CATAPULTS library | < 546+517 | 23808 bytes | 66.837 sec |

how much more sophisticated the generated scheduler is than the simple first-come, first-serve scheduler in cmthread.lib. The CATAPULTS library also links with another 517 line auxiliary library that contains implementations of the various thread container types provided by CATAPULTS. The Dynamic C compiler will only link in the functions from this auxiliary library that are actually used by the specific application, so only a couple hundred of these lines are likely to be included in any given application. So, comparing only lines of code is somewhat misleading; comparing code size is more useful. After compiling the simulation application along with the threading library, the total code size downloaded to the Rabbit processor was, as shown in Table 8.1, 23808 bytes when the generated CATAPULTS library was used as compared to 21120 bytes when the generic cmthread.lib was used (i.e., a 12.7% increase in size).

### 8.2.1 Weather Monitoring Station Application

To measure the performance difference between the CATAPULTS generated scheduler and generic cmthread.lib scheduler, we executed the control simulation until a total of 10000 executions of the "calculation" threads had run and then measured the total runtime. When using the generic cmthread.lib, we allow threads to notice that they have no work to do and yield immediately; this eliminates the additional overhead of useless hardware I/O, but still incurs the overhead of an unnecessary context switch. As shown in Table 8.1, the simulation completed almost 10 seconds faster when using the CATAPULTS-generated scheduler (a 12.6% speedup).

Table 8.2: Comparison of CoW web server throughput (with 4 handler threads) under generic and CATAPULTS schedulers.

| number of clients | generic scheduler (bytes/sec) | CATAPULTS scheduler (bytes/sec) | Improvement in Throughput (percent) |
|---|---|---|---|
| 1 | 4245.23 | 5958.96 | 40.37% |
| 2 | 7359.28 | 12028.24 | 63.44% |
| 4 | 16181.50 | 18571.56 | 14.77% |
| 6 | 20567.03 | 21488.52 | 4.48% |
| 8 | 23407.21 | 24906.37 | 6.40% |
| 10 | 7451.62 | 26542.34 | 256.20% |
| 12 | 6477.88 | 16513.81 | 154.92% |

### 8.2.2 CoW Web Server

Section 7.2.1 described the CoW web server for PCs. Another version of CoW runs on the Rabbit processor. This version of CoW uses the standard Dynamic C first-come, first-serve scheduler. Although CoW provides satisfactory performance with this generic scheduler, we realized that performance could be improved by using an application-specific scheduler. So, we developed a CATAPULTS scheduler for this version of CoW.

Unlike PC-based threading libraries like GNU Pth, the `cmthread.lib` library on which CoW is built does not have a notion of "blocked on I/O" threads. This is because the TCP interface provided by Dynamic C requires manual pumping and polling by the application; there is no operating system to monitor the sockets and raise I/O events to the threading library. This deficiency means that all CoW handler threads are always on the ready queue and that during execution, there is a context switch into each handler thread. Quite often it is determined that no socket events have occurred (no new data for a reading thread, no new connection for a listening thread, etc.) and another context switch happens immediately. This extra context switching is wasteful; our CATAPULTS scheduler allows us to eliminate this cost.

CoW uses a separate thread to perform the necessary pumping/polling of all the TCP sockets. Only after an iteration of this thread will handler threads see changes in the status of their TCP sockets. Our CATAPULTS scheduler makes use of this knowledge as follows:

Table 8.3: Comparison of CoW web server throughput (with 8 handler threads) under generic and CATAPULTS schedulers.

| number of clients | generic scheduler (bytes/sec) | CATAPULTS scheduler (bytes/sec) | Improvement in Throughput (percent) |
|---|---|---|---|
| 1 | 4460.36 | 5390.08 | 20.84% |
| 2 | 8738.18 | 9621.48 | 10.11% |
| 4 | 15653.21 | 17587.51 | 12.36% |
| 6 | 21623.12 | 21712.17 | 0.41% |
| 8 | 26289.43 | 23062.81 | -12.27% |
| 10 | 33543.06 | 26981.74 | -19.56% |
| 12 | 38085.71 | 30544.68 | -19.80% |

- When a handler thread context switches out, instead of putting the thread back on the ready queue, the scheduler now checks to see what action the thread is currently performing. If the thread is performing a socket operation, the thread is moved to a separate queue (LISTENQ for listening threads, READQ for reading threads, etc.). This thread can make no further progress until its socket operation completes, so it is isolated from the ready queue.

- When the TCP driver thread finishes a pump/poll run and is switched out, the scheduler performs additional logic before selecting the next thread to run; it checks for updates in the status of sockets on the blocked queues (i.e., new connections on listening thread sockets, new data on reading thread sockets, etc.). If such socket events have occurred, then the corresponding thread is moved back to the ready queue.

The complete CATAPULTS scheduler specification for CoW appears in Appendix D.

We benchmarked CoW with both the generic scheduler and the specialized CATAPULTS scheduler using a varying number of web clients and the server running with either four or eight handler threads. Although eight threads may not seem like a large number for a regular web server, the limited memory available on the Rabbit processor will not support the TCP buffers and stack space required for more handler threads. We benchmarked with up to 12 simultaneous clients to ensure a backlog of pending requests at the upper end of

the benchmark. Tables 8.2 and 8.3 present the results.

It is interesting to note that while the CATAPULTS scheduler always provided a speedup in the four handler thread case, it only provided a speedup for lower numbers of web clients in the eight handler thread case. This makes sense intuitively. First, consider the eight handler thread case. Part of the performance benefit of the CATAPULTS scheduler is that threads that are listening for incoming connections are not scheduled in the ready queue with threads that are actually processing requests. As the number of web clients increases, the server will become overloaded and all handler threads will be working all the time and will not spend any time waiting on the listening queue, thus erasing the benefit of isolating listening threads. Furthermore, since these benchmarks were run across a 100 mbit switch, there was very little network latency between the server and the clients, so threads spent little (if any) time waiting on the read queue and write queue. The additional bookkeeping overhead of moving handler threads to these separate queues and immediately back actually decreased the throughput of the system. We expect that if we were to run these tests across a larger network, reading and writing delays would allow threads to actually spend significant time on the reading and writing queues, which would result in significant performance gains for all numbers of web clients. Performance also goes down from the 10 to 12 client case since the server is completely saturated and additional pending connections just require additional bookkeeping overhead by the TCP system.

In contrast, a performance gain was always measured when only four handler threads were used. In this case, the lower number of threads allowed the TCP thread to cycle back and run again more quickly. The lower thread cycle time made it possible for the TCP thread to run again before reading and writing operations had completed for some of the handler threads, so the use of separate reading and writing queues provided an advantage, even on the low latency network.

# Chapter 9

# CATAPULTS for Distributed Applications

This chapter discusses our work with CATAPULTS on distributed systems and applications. We use the term "distributed" loosely here; it applies to any application or system that is broken into multiple components such that each component executes in a separate address space and maintains its own threading system. Three main types of real-world systems fall under this definition:

- *"Classic" single program distributed applications* — This category includes any application that consists of several processes split across a network. Such applications often employ multithreading to handle the concurrency at each network node. An example of this type of application is a typical MPI/PVM program.

- *Multi-process Multi-threaded applications on a single physical machine* — This category includes applications that run on a single physical machine, but use OS-level process creation to split themselves into separate address spaces. Each address space contains its own threading subsystem and communicates with the other address spaces via named pipes, shared memory mapping, temporary files, or other interprocess communication mechanisms provided by the operating system. This model is

used by applications such as Internet servers (web, database, email, etc.) to overcome OS-level limitations (e.g., the limit on the maximum number of file descriptors a single process can open). It can also be used for prototyping and testing truly distributed applications in a simpler setting.

- *Cooperating applications* — This final category describes systems that are composed of multiple independent, but communicating multi-threaded applications. A common example is a web server and a database server used together to serve an interactive website.

| Category | Machines | Programs |
|---|---|---|
| Classic | multiple | single |
| Multi-process multi-threaded | single | single |
| Cooperating applications | single or multiple | multiple |

Table 9.1: General categories of distributed applications

Table 9.1 summarizes the types of applications supported by distributed CATAPULTS. In all of these cases, multithreading is handled locally by each component. It would be possible to write separate CATAPULTS schedulers for each component in order to improve their performance individually, but such schedulers would not be able to take into account the global state of the system in order to properly load balance work between nodes or adapt to the producer/consumer behavior of various nodes in the system. In order to improve full-system performance, CATAPULTS supports the creation of high-level schedulers that manage the behavior of a distributed system as a whole in order to provide optimal overall performance.

This chapter discusses the details of our design, implementation, and experimentation with distributed CATAPULTS. The term "CATAPULTS" in this chapter refers to our extended version of CATAPULTS for distributed systems. The term "classic CATAPULTS" refers to our non-distributed version described in the preceding chapters. The rest of this chapter is organized as follows. Section 9.1 gives a high-level overview of the design of a

distributed scheduler. Section 9.2 introduces the language extensions to support distributed schedulers and motivates our work by describing three example distributed applications for which we developed schedulers. Section 9.3 explores the implementation details of the compiler and runtime system CATAPULTS uses for distributed applications. Finally, Section 9.4 provides experimental results of applying CATAPULTS schedulers to our three example applications.

## 9.1 Design of Distributed CATAPULTS Schedulers

CATAPULTS views a distributed system as a collection of ***nodes***. A node is a component with a distinct address space and a self-contained threading system. Furthermore, nodes in a distributed system are grouped into ***families***; nodes within a family share the same scheduling logic for their local scheduling and track the same statistics and thread attributes. The relationship between nodes and families in a distributed CATAPULTS scheduler is similar to the relationship between classes and objects in object-oriented programming; a family is a specification that describes the behavior and data tracked by a node, but an application may have multiple nodes that all use that same specification, but track their own instances of local data. Figure 9.1 shows an example of how a distributed scheduler is organized. This example system has a total of six nodes: three use a round-robin scheduling algorithm for their node-level scheduling, one uses a priority-based algorithm, and the final two nodes each use a lottery algorithm. Since some of the nodes use the same scheduling algorithm, there are only three node families defined by this scheduler.

Distributed applications can take many forms. Some applications involve static nodes that are present for the duration of system execution, while others allow nodes to join and leave the system mid-execution (e.g., as new hardware is plugged in or turned on). Some applications involve a fixed or maximum number of active nodes, while others allow an unbounded number of nodes to join the execution (e.g., applications like SETI@home).

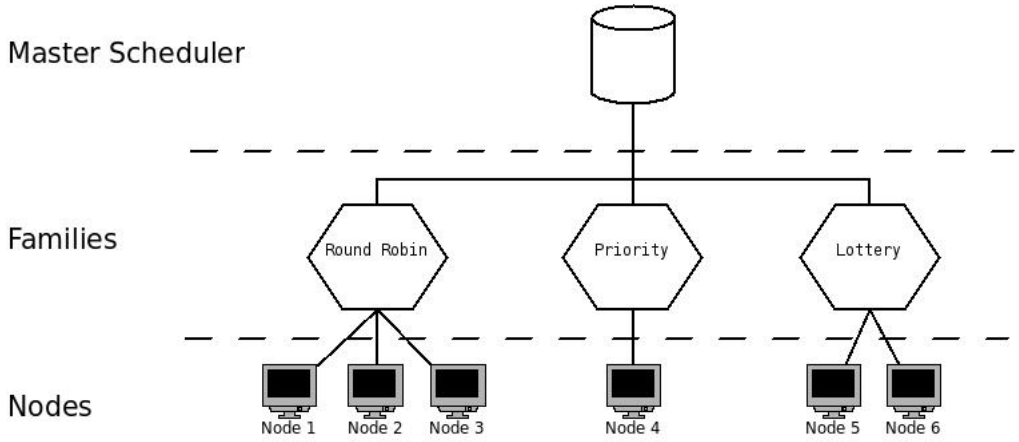



Figure 9.1: Example layout of a distributed CATAPULTS scheduler

Our design for CATAPULTS is general enough to suit all such application designs.

A CATAPULTS scheduler for a distributed system is designed as a collection of communicating schedulers. Each high-level system scheduler involves multiple node schedulers (one for each node family) called ***nodespecs*** and one high-level general scheduler called the ***master***. All of these cooperating subschedulers are able to communicate and share various global scheduling data. Distributed nodespec schedulers are syntactically identical to classic CATAPULTS schedulers; any classic scheduler can be used as a nodespec scheduler without modification. Variables defined in a nodespec's `data` section are referred to as ***node global variables***. Distributed schedulers can define higher-level variables, called ***distributed global variables***; there is only a single instance of each global variable. Distributed global variables are accessible by all node schedulers. When a variable name is encountered in the event handler of a distributed scheduler, the scheduler tries to resolve it as a distributed global, node global, and then local variable in that order. For clarity, node schedulers may optionally specify that the variable belongs to the distributed global namespace by referencing it as "`GLOBAL.`*varname*" rather than just "*varname*." Node schedulers are also capable of referencing node global variables of other nodes using the syntax "*nodename.varname*" where *nodename* is a node declared in the master scheduler's `nodes` section (see Section 9.2 for details).

The master, implemented separately, describes how the system as a whole should function and respond to scheduling events. The master identifies the various node families that will be recognized by the scheduling system and describes what actions, if any, should be performed when new nodes join the system or when current nodes terminate and leave the system. It also defines the distributed global variables that are accessible to all of the nodespecs, as well as default implementations of any node-level scheduling events.

## 9.2 Distributed Scheduler Examples

This section describes three examples of distributed applications for which we have developed CATAPULTS schedulers. The syntax extensions added to the CATAPULTS language to support distributed applications are introduced via the code of the first example; full code for the other two examples is available in Appendices E–F. The first system is a typical website system consisting of a CoW webserver [26] and a MySQL database. Our second application is a simulation of a distributed embedded system for a security system. Our final application is a synthetic test that measures the ability of a CATAPULTS scheduler to enforce "balance" on a distributed application in terms of disk usage, network connections handled, etc.

### 9.2.1 Example 1: Web Application System

Our first distributed experiment for CATAPULTS was a typical web application system that involved two nodes, a web server (node #1) and a database server (node #2). Although these are two distinct applications, programmed by different groups of people, they work together closely to generate the interface a user sees when he visits a website. As such, performance can be improved by writing a CATAPULTS scheduler to allow the two applications to work together more efficiently.

The webserver software used in this test was a modified version of CoW (previously

introduced in Section 7.2.1). We compiled CoW in webserver mode with the maximum threadpool size disabled so that CoW would continue to create new threads to handle incoming requests, no matter how heavily loaded the server was. This behavior matches the default setting for many popular web servers, including Apache. MySQL was used as the database software used in this experiment. The web application running on top of this system was a simple web application consisting of several static files (80% of all incoming web requests) and several pages generated dynamically from the database (20% of all web requests). Table 9.2 gives the exact frequency and average file size of each type of document requested. Although all web applications are different, the relative request frequencies used were selected to match the traits of a real-world calendar and scheduling web application [38]; we analyzed log files for 12 months of operation of the application to determine the typical request size and type distribution.

| **Document Type** | Average Size (kbytes) | Request Frequency |
|---|---|---|
| GIF image (static) | 2.1 | 35% |
| CSS stylesheet (static) | 0.6 | 20% |
| Generated HTML (dynamic) | 9.6 | 20% |
| JPG image (static) | 26.2 | 15% |
| JS script (static) | 5.9 | 10% |

Table 9.2: Breakdown of web document types requested

In order to develop an effective scheduler for this web application, we first need to determine our performance goal. Without using a CATAPULTS scheduler, we found that static documents were always returned within hundredths of a second, but as the MySQL database became overloaded dynamic request performance deteriorated and response times for an individual dynamic document grew to several seconds. Since the dynamic content is the primary content on the site and the static documents are primarily images and style information, which we consider less important, we made our goal to improve the system's response time for dynamic content requests. Slower response times for static requests were considered an expected and acceptable tradeoff in this setting.

The scheduler that we developed attempts to improve the responsiveness of dynamic content by limiting the number of request threads actively attempting to query the database. The database is a bottleneck in this system because of the massive I/O associated with performing large, complex queries on the database. Our custom scheduler tracks the number of threads *blocked on I/O* inside the database. If the number of I/O-blocked threads in the databases surpasses a specific threshhold, the webserver stops scheduling any threads that have been determined to be associated with dynamic requests to prevent them from reaching the point where they submit more work to the database server. Until that limit

```
else if (dbload < DBMAX && |RQ_dynamic| > 0)
    RQ_dynamic => current;
else
    RQ_static => current;
```

Figure 9.2: Logic in web server scheduling event that manages database load.

is reached, web server threads associated with dynamic content requests will be given priority over those responsible for static content requests. Note that this does not limit the number of active threads in use by the database server, but only those actively blocked on I/O operations. The number of database threads performing computation (e.g., filter and join operations on data already loaded into memory) is unimportant. Figure 9.2 shows the crucial code for this scheduling logic.

Figures 9.3 through 9.7 illustrate the key components of the master scheduler for this application. Master schedulers contain the following sections, described in more detail below:

- scheduler name
- node family declaration
- node declaration
- distributed global variables *(optional)*

- node arrival handler
- node departure handler *(optional)*

Before any of these sections are defined, a distributed scheduler specification must first specify the name of the scheduler on the scheduler definition line, shown in Figure 9.3. The name chosen is used internally by the scheduler to make sure all nodes that connect to the system belong to the same application. It is also used to name various temporary and intermediate files, as described in Section 9.3.

```
distributed scheduler webapp {
```

Figure 9.3: Scheduler name

The first section of a master definition is a node family declaration block, shown in Figure 9.4. This section specifies the filename of each nodespec subscheduler used by the application and associates it with a symbolic name. For example, the nodespec for the database node is implemented in file `webapp.database.subsched` and is given the name symbolic name `database`. The example in Figure 9.4 consists of just two nodes and each belongs to its own node family with unique scheduling logic.

```
nodefamilies {
  "./webapp.database.subsched" : database;
  "./webapp.webserver.subsched" : webserver;
}
```

Figure 9.4: Node family declarations

A node declaration section, illustrated in Figure 9.5, follows the node family declarations. This section declares the data structures used to organize the collections of nodes present in the system. As previously stated, distributed application designs vary greatly. In applications where all nodes are static and known at design time, each node can simply be declared as a variable of type “node” using C-style syntax. However, other applications may not know how many nodes will be present while the system executes, or nodes may

join and leave the system. In these cases, nodes may be organized into simple collections (stacks and queues) just as threads in a classic scheduler or node scheduler may be organized into collections (as previously discussed in Section 5.1). Although nodes may also be placed into `pstacks` and `pqueues` that are sorted based on the value of node global variables, these containers will not be re-sorted when the node global variables change value.

The declaration of nodes in a distributed scheduler closely mirrors the declaration of threads in a classic CATAPULTS scheduler, as shown in Chapter 4. Whereas threads can either be declared directly using `threadref`'s or placed in container types, nodes can either be declared directly as `node`'s or placed in the same types of containers.

For this example, the number of nodes and their purposes are known statically, so the node declaration section simply defines two `node` containers.

```
nodes {
    node dbnode;
    node wwwnode;
}
```

Figure 9.5: Node declarations

The next section of a distributed master scheduler is a block of distributed global variables (illustrated in Figure 9.6). Variables declared in this section are accessible to all nodespec subschedulers and changes made to their values at one node will be reflected at other nodes (details of how these updates take place are given in Section 9.3.4). This example declares two global variables (to track the load on the database server and the number of active threads in the web server) and one constant.

The final two sections of a master scheduler are an `arrival` block and a `departure` block. The syntax of each of these blocks mirrors that of an event or query handler in a classic CATAPULTS scheduler and takes the node leaving or departing the system as a parameter. An example of an `arrival` block is given in Figure 9.7. In order to determine

```
global {
    int dbload;     // DB "load";  # of threads blocked on IO
    int webthreads; // # of webserver threads

    const int DBMAX = 5;  // max allowed load on DB
}
```

Figure 9.6: Distributed global variables

the purpose and properly categorize a new connecting node, arrival blocks can use a new `is` keyword to determine the node family associated with the new node. The expression "*n* `is` *type*" will return true if the connecting node's family was given the symbolic name *type* in the family declaration, otherwise it will return false.

```
arrival(n) {
    if (n is database)
        n => dbnode;
    else
        n => wwwnode;
}

departure(x) {
    /* noop in this example */
}
```

Figure 9.7: Arrival and departure blocks

`arrival` blocks are required to store the incoming node in one of the containers defined in the `nodes` block. `departure` blocks are not required to take any action and are often empty if no cleanup work is required by the scheduling logic, as is the case in this example.

### 9.2.2 Example 2: Embedded Application Simulation

Our second experiment for distributed CATAPULTS was a simulated embedded system for camera and sensor monitoring. The system had several components:

- Multiple sensor/camera nodes that take snapshots of their environment at periodic

intervals (e.g., every 5 seconds).

- A computation node that contains a small circular buffer of temporary storage. This node is responsible for encoding/compressing the data and writing it to permanent storage (assumed to be a network-accessible disk).

- A web server node that allows the system's owners to watch current video feeds from the system, examine archived snapshots, or control various cameras.

Although this is not a real-world application, we feel that it is representative of the organization and design of embedded systems used in settings such as zoos (to monitor the activity of animals in their cages) and museums.

Most embedded systems are designed so that the hardware available meets the computation needs of the software. However, systems like the one described here often have components whose computation needs vary greatly and cannot be accurately predicted in all cases. In this example, the web server nodes are susceptible to visitor traffic and may require more network bandwidth, CPU time, or I/O activity than the system can sustain. The amount of work done at the computation node to encode or compress data from the cameras and sensors may also change depending on the amount of activity the cameras are experiencing. If the encoding scheme is diff-based, more computation will be required to encode images that change significantly from frame to frame than images that are identical for long periods of time. Some applications could build fixed limits on the resource usage of "unknown" nodes like the webserver, but these hard coded limits may actually restrict the performance of the system at times when the resources really do exist to support the increased needs.

In our example application, the greatest potential source of failure for the system is that the computation node will fall behind on the encoding process and that the temporary storage space for raw data will fill up. At this point, new data from the cameras and sensors will be lost, or else old data will be overwritten before it has been encoded and flushed to

permanent storage. This could become a problem if the web server saturates the network enough to impede the transfer of data between the sensors/cameras and computation node, or if the web server generates enough disk I/O that encoded data cannot be written to permanent storage quickly enough.

Our scheduler for this application, which is shown in its entirety in Appendix E, solves the overflowing buffer problem in two ways. First, it restricts scheduling of the web server's "accept" thread as the temporary storage buffer fills up. When the number of web request threads being processed exceeds a given threshhold, the accept thread is no longer scheduled, which prevents new requests from being accepted and placing additional load on the system. The actual threshhold varies dynamically with the temporary storage buffer's current utilization. Second, our scheduler also manipulates the rate at which new data is accepted from the cameras and sensors. The cameras are already timed at the application level to take a snapshot every 5 seconds, but if the thread responsible for driving the camera does not wake up once 5 seconds has elapsed, the next snapshot will be delayed or dropped. Although a 5 second interval is desirable, we believe that occasionally delaying or dropping a frame to slow the rate at which the temporary buffer fills is preferable to letting the buffer completely fill and losing all data until the situation is resolved.

### 9.2.3 Example 3: Load Balancing Simulation

Our final experiment with distributed CATAPULTS was a synthetic benchmark designed to test CATAPULTS' ability to manage load balancing. Many multi-threaded applications break themselves into multiple OS-level processes in order to effectively utilize multiple processors, disks, or other resources. Our experiment is a multi-process multi-threaded application that reads requests from the network, performs some processing, and then returns a result; this type of model is typical for Internet servers such as web, database, or mail servers. Our goal is to write a scheduler that will enforce equal distribution of work across all of the OS-level processes. Since the server socket is created before the applica-

tion forks into multiple OS-level processes, all processes listen on the thread and attempt to accept new connections. When a new connection arrives, the process that actually wakes up and reads it off the socket is determined by the OS.

Our scheduler, shown in Appendix F, is very simple. Given $n$ nodes in the system, $r$ requests currently being processed across all nodes, and a user-defined threshold $T$, the network accept thread of a node will be suspended if that node is processing more than $\frac{r}{n} + T$ requests. By suspending the accept thread, no new requests will be taken on and other nodes will be able to "catch up." This ensures that no node takes on the majority of the work, leaving other nodes underutilized.

## 9.3 Scheduler Implementation

### 9.3.1 Overview

Our implementation of CATAPULTS targets distributed systems residing on a single physical machine and uses GNU Pth for thread management at each node. Nodes in our system are separate OS-level processes; they can either be forked copies of a single application or processes spawned by independent, cooperating applications. In addition to each application-level process involved in a distributed application, our implementation also spawns one additional process, the ***manager***, for storing distributed global variables and facilitating inter-node communication. The version of GNU Pth that we use has been heavily modified to communicate with other nodes in the system; details are given in Sections 9.3.3 and 9.3.4.

Although our implementation resides on a single physical machine and uses OS pipes to pass messages, the implementation could also be extended to distributed applications operating across a network by replacing the pipes with TCP sockets. While network communication is significantly more expensive than local message passing, proper handling of inter-node communication is still very important under our single machine model since

every message passed incurs several additional system calls to read from and write to the communication pipes.

### 9.3.2 Parsing and Compilation

Parsing and compiling a distributed scheduler is significantly more complicated than processing a classic CATAPULTS scheduler. Distributed global variables defined in the master scheduler can be referenced in nodespec schedulers. Conversely, node-specific global variables defined by a nodespec can be referenced in the master scheduler, or even in other nodespec schedulers if their source node is named directly. Thus the parsing, type checking, and compilation of a distributed scheduler must be done as a single, complete operation and cannot be broken into separate pieces.

In order to compile a distributed scheduler, the name of the master scheduler is passed to the CATAPULTS parser on the command line. The specified scheduler file is loaded and parsed to generate an abstract syntax tree (AST). Since variables in this AST may refer to variables defined in nodespec schedulers (which have not yet been parsed), type checking cannot yet take place. Instead, the list of filenames and symbolic names for nodespecs utilized by the scheduler are extracted. Each nodespec filename is passed to a separate helper program that loads the nodespec and parses it to generate an AST. The resulting AST is then serialized using Python's `pickle` module and written to a temporary file. Once the helper script has been run on each node scheduler, all of the AST's are loaded into memory and a complete, scheduler-wide symbol table is generated. Type checking and semantic analysis are then performed on the subschedulers for the master and all nodespecs. Finally, separate `.c` files are generated for each subscheduler.

The `.c` file for each nodespec subscheduler (named *`name`*`.subsched.c`, where *name* is the symbolic name provided in the master scheduler specification) is compiled into `.so` shared libraries and loaded at startup time on each node as described in Section 7.1. The `.c` file for the master scheduler is compiled into a standalone manager exe-

cutable. A unique identifier consisting of the name of the master scheduler (as specified on the "`distributed scheduler` *`name`*" line) and the timestamp at which parsing was performed is embedded into each of the `.c` files. This identifier is compared at startup on each node to make sure that all nodes and the manager process are using the same version of the scheduler specification; if even a single node is accidentally executed with a different version of the scheduler, the results would be unexpected and inconsistent. It is important to note that although nodespec implementations are re-usable by different distributed schedulers, the `.c` and `.so` files into which they parse and compile are not reusable; the original specification must be re-compiled in order to be used in a different distributed scheduler.

### 9.3.3 Scheduler Initialization and Node Coordination

Coordinating the initialization of a distributed system is a complicated process. Coordinating the initialization of a communicating, multi-node threading system that is layered on top of a distributed application that has no awareness of its presence can be even more complicated.

Proper initialization of a distributed CATAPULTS scheduler requires that the manager process be started before any of the nodes are initialized, but makes no further restrictions on the arrival/departure patterns of the nodes in the system. Although mechanisms such as barriers for initialization coordination are generally considered an application-level concern, it is possible to mimic them at the scheduler level by simply not making any application threads eligible for scheduling until all expected nodes are present.

The first action taken by the manager process upon startup is the creation named pipe called `.`*`name`*`-fifo` where *name* is the name of the master scheduler. It then waits for messages from nodes to start arriving over the pipe. The manager is a single-threaded process that simply waits for messages to arrive and then processes them; thus it spends the majority of its time blocked on a `select()` system call and has negligible effect on other processes running on the machine.

When an application node initializes, it also creates a named pipe of its own under the filename `.name-client-pid` where *name* is again the name of the master scheduler and *pid* is the process ID (PID) of the node. It then opens the manager's pipe and writes a short message identifying itself as a new node in the system, providing the proper version-checking identifier, and specifying its PID. Once the manager has the PID for the connecting node, it opens the node's pipe and uses that as a channel to communicate with the node. Thus in a system with $n$ nodes, the manager has $n$ pipes open for writing and 1 pipe open for reading.

### 9.3.4 Communication Between Nodes and Manager

#### Models

Distributed schedulers rely on communication between the manager and nodes for effective scheduling. This communication consists of inter-node commands (e.g., suspension of remote threads) and the transfer of variable data. CATAPULTS provides a distributed shared memory (DSM) interface to distributed global variables and accesses to these variables are translated into inter-node messages by the CATAPULTS runtime system. Variable data makes up the majority of the communication in a distributed scheduler. The DSM interface provided by CATAPULTS can be implemented using one of two models:

- *Request-based model* — Under this model, variable data is stored in a single location and must be requested each time a node needs to read or write its value. Distributed global variables are stored at the manager and node globals are stored at the individual nodes. The overhead associated with this model is two communication operations (one pipe write from the source to the destination, and one pipe write from the destination to the source) for each access to a distributed variable. This type of centralized, request-based model is used by other DSM implementations, such as PerlDSM [34].

- *Broadcast/Cache model* — Under this model, a separate copy of variable data is

stored at every node in the system that may access it. When such a variable is updated at any node in the system, the update is broadcasted to all nodes in the system so that they may update their locally cached copies. When distributed variable data is accessed by a node, it uses its locally cached copy. The overhead associated with this model is $n$ communication operations, where $n$ is the number of nodes in the system.

Clearly neither of these models is superior to the other since the relative cost of communication depends on the variable access patterns of a scheduler. A scheduler that accesses global and cross-node variables frequently, but rarely updates their values will perform poorly due to the high volume of traffic on the communication channel, but using the broadcast/cache model provides minimal overhead. In contrast, another scheduler that updates global variables frequently, but only accesses them occasionally will perform much better with the request-based model. The tradeoffs between the two models are similar to those faced by Linda [24] implementations where the implementation must decide whether the tuple space should be stored centrally, or distributed among the physical nodes of the system [17, 40]. CATAPULTS allows distributed schedulers to be compiled with either type of DSM model. We currently allow the user to select the desired model by passing a command line switch to the CATAPULTS compiler, but in the future we would like to extend CATAPULTS to attempt automatic detection of the optimal communication model.

### Synchronization

As in any concurrent system, it is important to ensure that shared variables maintain a consistent state and that race conditions and deadlock can be avoided. In our experience, the vast majority of shared variables in CATAPULTS schedulers represent counts (e.g., number of threads in some state, number of times some scheduling condition has occurred, etc.). As such, updates to the values of these variables are usually performed by autoincrement/autodecrement or by the arithmetic assignment operators (+=, -=, etc.). All of these operations on shared variables are atomic in CATAPULTS; when such an operation is per-

formed at a node, a single autoassignment message is sent to the destination rather than separate fetch and set messages. Thus it is not possible for local schedulers at other nodes to access the value of a variable between the time it is read and updated by one of these operations.

Another case where synchronization is necessary is if multiple variables need to be updated and kept in sync. An example of such a situation would be two count variables to track the number of threads in the system with and without a specific trait. If a thread gains or loses the trait, both variables need to be updated, but the update must be atomic so that the sum of the two totals always matches the total number of threads in the system. If two separate assignment operations are sent to update each variable, it is possible that another node in the system could read both of their values between the time the first assignment arrives and the second assignment arrives, thus creating a race condition. Although we have not come across any schedulers that need this type of update functionality, one simple, but primitive solution is synchronization using only shared-variable programming techniques [11] (no additional mechanisms for synchronization): a third variable at the destination node can be used as a lock variable. As already mentioned, the autoincrement operation is atomic so it can be used as a "test and set lock" operator. Before entering the critical section and updating the multiple distributed variables, a node will first perform an `l = sem++` operation, where `l` is a local variable, and `sem` is a distributed variable acting as a semaphore. If `l` receives a non-zero value, then another node is already operating in the critical section and updating the distributed shared variables, so the current node will have to wait, using a busy-waiting loop. If `l` receives a value of 0, then no other nodes are operating on those variables and the current node may enter the critical section. When finished with the updates, the node may simple perform `sem = 0` to unlock the critical section and allow other nodes to access those variables.

The DSM interface provided by CATAPULTS is very simple. The request-based model described above provides sequential consistency [31], while the broadcast/cache model

only provides causal consistency [9]. Far more sophisticated work exists in other DSM implementations; relaxed memory consistency models such as release consistency [28, 27] and entry consistency [16, 41] can be used to reduce network traffic on variable updates, although they require the explicit use of acquire/release instructions or the manual association of data with guard variables. Other DSM implementations allow data to be dynamically migrated at runtime based on the operation of the system [42]. In the future, we may try to incorporate some of these improved memory consistency models or data migration techniques to CATAPULTS, but so far our initial experience indicates that our simple DSM implementation is satisfactory.

## 9.4 Experimental Results

This section experimental results from applying CATAPULTS schedulers to the three distributed applications discussed in Section 9.2.

### 9.4.1 Web Application System

We tested our scheduler by bombarding the web server with web requests matching the statistics shown in Table 9.2. From the web application's point of view, this models the type of traffic that would be experienced during traffic spikes (e.g., if the web application were linked from a popular news website). Thirty client machines were used to submit the requests. We performed two benchmarks: one had 3000 total web requests and the other had 15000 total web requests. Each experiment was run multiple times and the average response times are given in Tables 9.3 and 9.4.

As the results show, our custom scheduler significantly improved the performance of dynamic requests, bringing the average response time down by over a second to roughly a quarter of its original time. This performance was gained at the expense of slower responses to static requests, but we consider this tradeoff acceptable since the dynamic content is what

| Document Type | Default Scheduler | Custom Scheduler |
|---|---|---|
| Generated HTML (dynamic) | 1.724 | 0.450 |
| GIF image (static) | 0.015 | 0.380 |
| CSS stylesheet (static) | 0.014 | 0.372 |
| JPG image (static) | 0.020 | 0.528 |
| JS script (static) | 0.019 | 0.525 |

Table 9.3: Benchmark 1 (3000 web requests): average response time (sec) for various requests

| Document Type | Default Scheduler | Custom Scheduler |
|---|---|---|
| Generated HTML (dynamic) | 1.729 | 0.447 |
| GIF image (static) | 0.037 | 0.395 |
| CSS stylesheet (static) | 0.040 | 0.372 |
| JPG image (static) | 0.039 | 0.527 |
| JS script (static) | 0.038 | 0.528 |

Table 9.4: Benchmark 2 (15000 web requests): average response time (sec) for various requests

the users of this web application actually care about (i.e., the calendar events requested by users' queries).

### 9.4.2 Embedded Application Simulation

We tested our embedded application simulation by providing its web interface with periodic spikes of traffic as displayed in Figure 9.8. As previously explained, overall performance of the system is not our concern, but rather its ability to withstand unexpected spikes of traffic. Without a CATAPULTS scheduler, the large spikes of web traffic placed too much load on the system and the application was unable to fully encode all of the data from the cameras before the temporary buffer ran out of free space. Such buffer overflows resulted in data loss. Figure 9.9 shows the utilization of the temporary buffer over the course of the experiment. When we applied the scheduler described in Section 9.2, the system correctly adapted to spikes in web traffic by throttling back the number of requests accepted. The less space available in the temporary buffer, the fewer requests accepted; when the buffer has no more room ("Queue Remaining" hits 0 in the figure), no requests

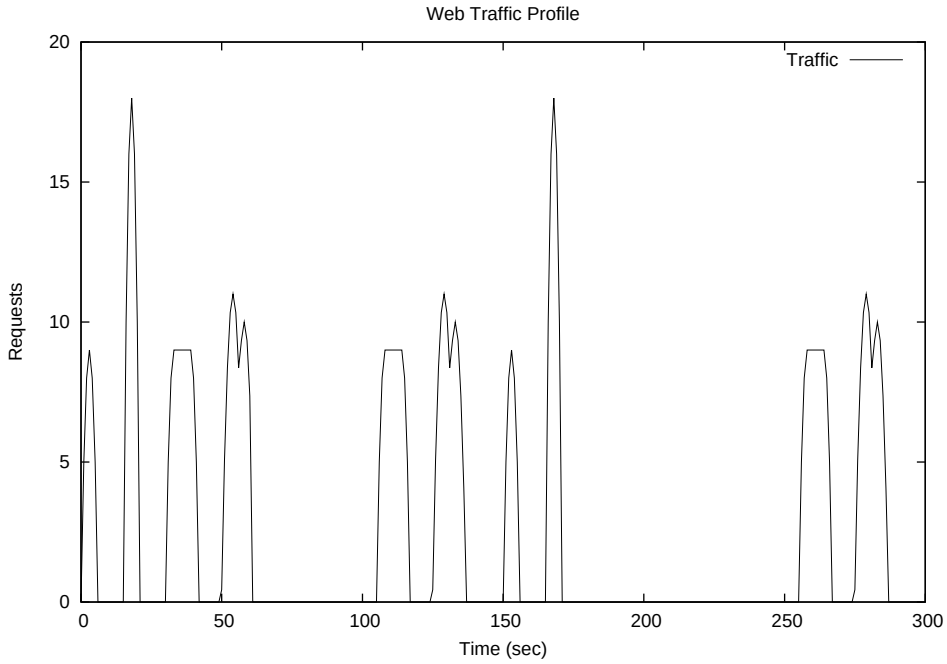



Figure 9.8: Input traffic provided to embedded application's web interface

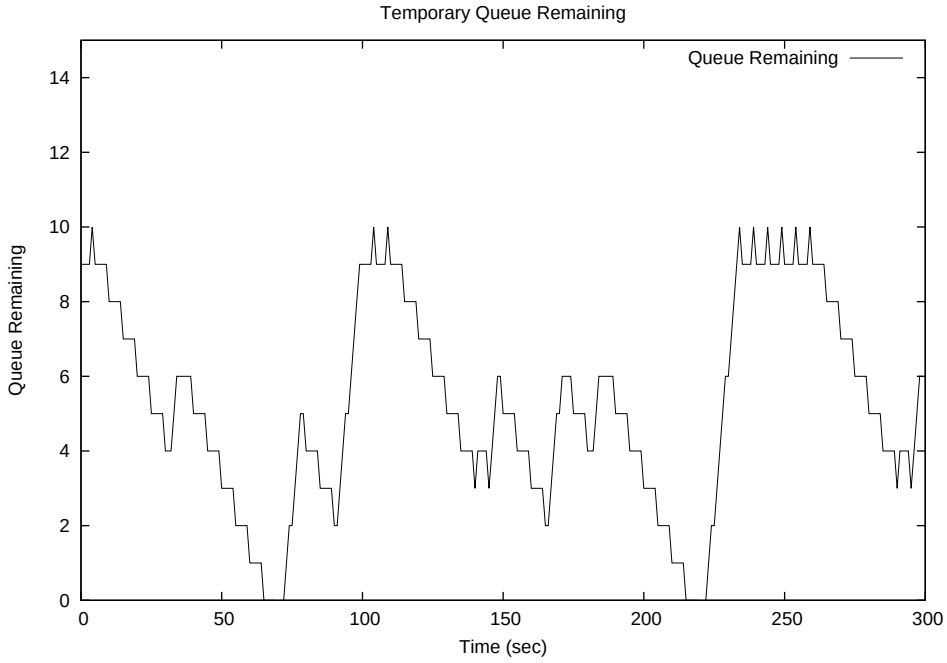



Figure 9.9: Temporary buffer usage without a custom scheduler

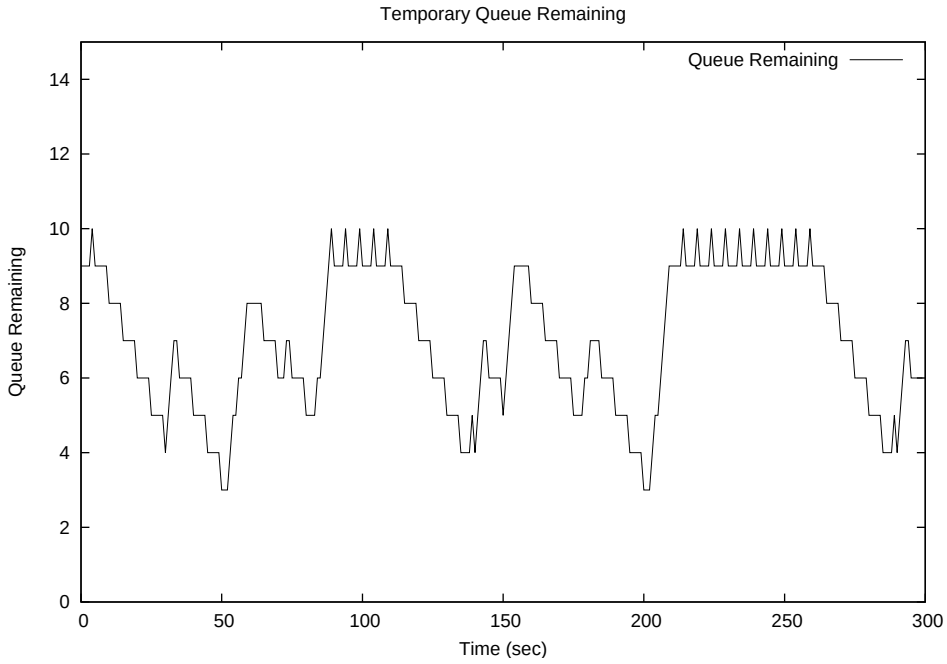


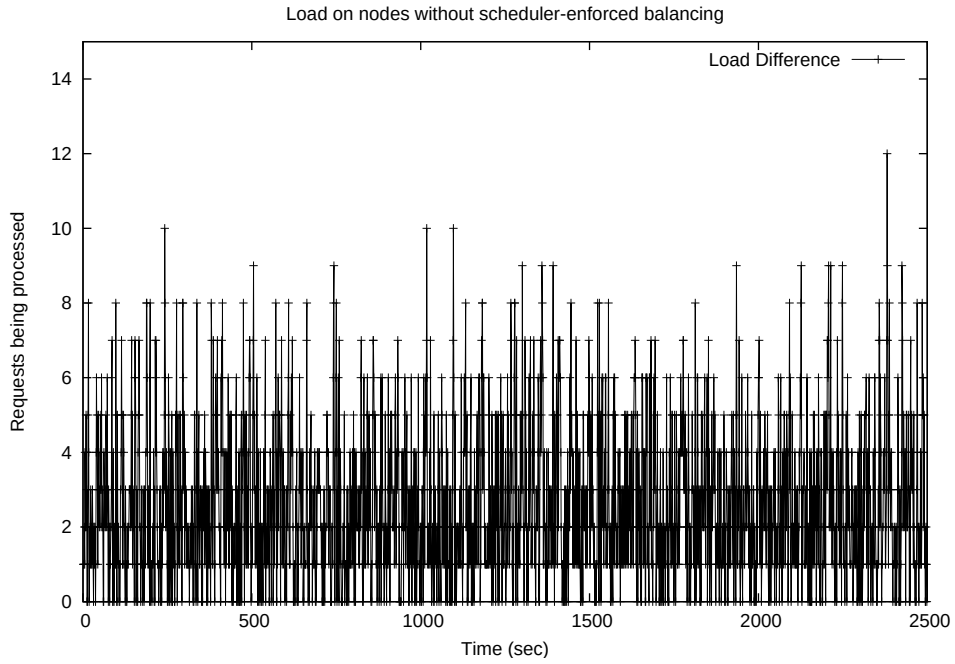

Figure 9.10: Temporary buffer usage with a custom scheduler

can be handled. Figure 9.10 shows the buffer's usage after applying our custom scheduler. As the figures show, the custom scheduler was successful at preventing the temporary storage buffer from filling up and resulting in data loss.

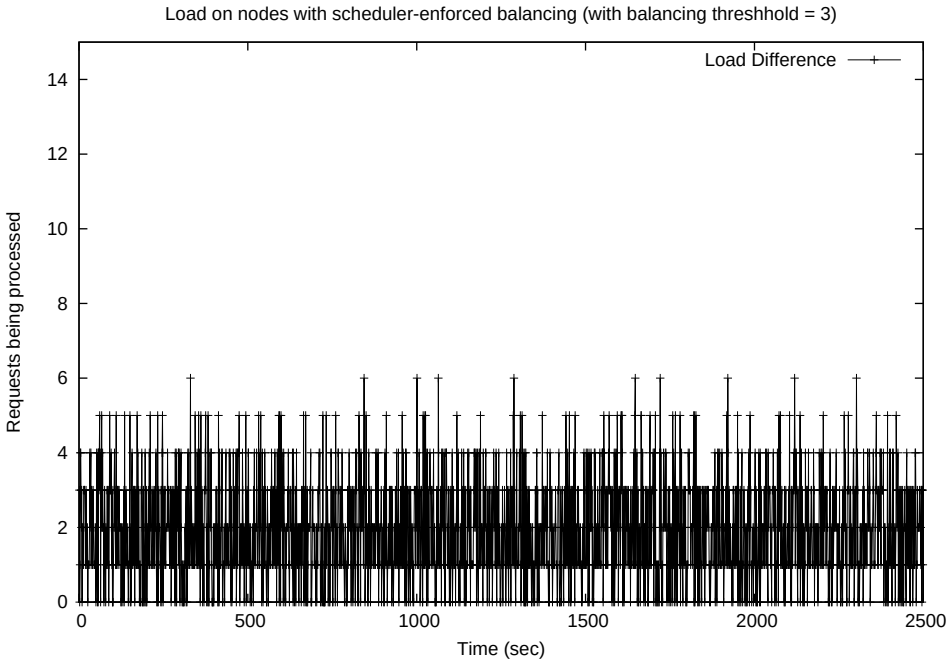
### 9.4.3 Load Balancing Simulation

We tested our load balancing simulation by providing input requests every 0.25 seconds. Each request took between 0.25 and 5.0 seconds to process. When executed without a CATAPULTS scheduler, both OS-level processes had a chance of accepting an incoming request and the two processes had the potential to become very unbalanced in terms of workload. When a load-balancing CATAPULTS scheduler (shown in Appendix F) was used with a maximum load difference threshhold of 3, the workload of the two processes stayed much closer. Figures 9.11(a) and 9.11(b) illustrate the behavior. Interestingly, although a load difference threshhold of 3 was enforced by the CATAPULTS scheduler, the two processes would occasionally get farther out of sync. This usually happened when

several requests at a node complete at roughly the same time (thus dropping that process' load) and are not replaced quickly enough by new incoming requests. The average load difference when load balancing was not performed was 2.68, although the difference spiked up to a maximum of 12. When load balancing was applied, the average load difference dropped to 1.78 and the maximum difference observed was 6.




(a) No CATAPULTS scheduler




(b) Load-balancing scheduler

Figure 9.11: Difference in server load with and without a CATAPULTS scheduler

# Chapter 10

# Discussion

## 10.1 Multi-threaded Application Designs

Although threads provide a very open-ended programming model, we found it interesting that most multi-threaded applications follow one of two very distinct designs. These designs are important to recognize because the optimal scheduling strategy and source of performance improvement differ between the two.

The first design, the "server pool" application design, is generally used for highly concurrent Internet servers (web/proxy, database, mail, etc.). The CoW web server (both for general computing and for embedded systems) and the MySQL database server discussed in Section 7.2 follow this design. Applications following this design make use of a large number (hundreds or thousands) of threads, almost all of which perform the same operations on different sets of data, network connections, or users. In this kind of system, performance is usually improved by prioritizing the order that "worker" threads execute such that those closest to completion can finish quickly and free up resources for other workers. To accomplish this prioritization, a CATAPULTS scheduler for such an application can use one or more `pqueue` or `pstack` containers, sorted on a value that indicates how close a worker thread is to completion. Multiple sorted containers are useful if a worker thread

passes through multiple stages of operation so that threads can also be prioritized based on their present stage. For example, threads in our CoW web server example go through a request processing stage and then a response stage. The largest challenge when developing a scheduler for this type of application is dealing with any additional non-worker threads. For example, an Internet server may have one thread that is simply responsible for accepting new network connections; such a thread does not fit into the same prioritization scheme as the rest of the threads since it never completes its task; additional logic must be added to the scheduler to determine how often such auxilary threads need to run and properly interleave their executions with the executions of worker threads.

The second significant application design, the "heterogeneous task" design, is characterized by a smaller number of threads than the server pool design with most of the threads performing different, largely unrelated, tasks. The weather monitoring station example from Section 8.2.1 follows this design. Embedded systems often use this type of application design (e.g., different threads may be responsible for driving various pieces of mechanical hardware), as do multimedia applications where unrelated tasks such as decoding, video output, sound output, general application logic, etc. are all handled by separate threads. Applications of this type are more varied in purpose than those of the server pool design and do not measure performance in terms of requests handled per unit time, so it is more difficult to give a general metric for measuring performance. Since the definition of "performance" is much more application-specific in this case, performance improvement strategies also depend highly on the purpose of the application. When developing a custom scheduler for this type of application, it is important to consider the purpose of individual threads and how their roles can affect the performance of the application as a whole. Important threads that need to operate with high latency should be given higher priority than less important threads; threads that only need to run at well-defined intervals (e.g., a thread refreshing a video display) should only be considered for scheduling at the appropriate times; threads with data dependencies on other threads in the system should be scheduled

accordingly. Using this kind of application-specific knowledge, application performance can be improved, regardless of how "performance" is measured — work completed, power consumed, etc.

## 10.2 Expressiveness versus Safety Tradeoff

Allowing application programmers to replace the thread scheduler, a very highly tuned component of most software systems, is a controversial approach. Although errors introduced in the scheduling specification can result in poor performance or instability, well-written schedulers can result in significantly improved performance. The use of CATAPULTS is an optimization with a tradeoff: higher performance at the cost of additional work writing a CATAPULTS scheduler and less assurance of stability. We expect applications to be written without regards for the scheduler, and then, if higher thread performance is necessary, an application-specific scheduler can be written and plugged-in. The most "dangerous" feature of CATAPULTS is the use of imported application variables (discussed in Section 5.2) since it allows direct interaction between the application and the scheduler. Importing application variables is an optional feature that allows more specialized scheduler development at the cost of tighter coupling between the application and scheduler; the application developer can decide whether this tradeoff is worthwhile for the specific application. Even if application-specific schedulers that import application variables are not desired, performance can often be enhanced simply by selecting an appropriate generic scheduler for the application. In this case, the scheduler can be developed and fine-tuned by a third party, which makes the use of a CATAPULTS scheduler just as safe and easy as using the original, built-in scheduler.

## 10.3 CATAPULTS Visualization

Thread scheduling is a complicated subjected and not an area that most application programmers have much experience with. CATAPULTS schedulers are generally developed and maintained by applications programmers, so we have developed an additional visualization tool to help programmers understand the behavior of a CATAPULTS scheduler at a high level. Our visualization system is actually developed as an additional backend for CATAPULTS. Instead of generating C code that can be compiled into a scheduler, it instead outputs specifications for Graphviz [3] which can then be compiled into diagrams that illustrate how threads are transferred between containers by the scheduler. See Figure 10.1 for an example. Although this diagram does not convey the full semantics of a scheduler, it does allow a programmer unfamiliar with the scheduler to quickly get an idea of how threads are organized and handled. Arrows between nodes (thread containers) in the graph indicate that the event handler indicated in the label performs a thread transfer between the two containers; bold arrows indicate a transfer that occurs inside a loop and dashed arrows indicate one inside a conditional statement. Since thread parameters to event handlers are often already in a thread container, the visualization tool attempts to determine to which container the parameter belongs. It does so by analyzing the conditions of any conditional or looping statements enclosing the thread transfer operation. The diagram generated can help the programmer detect bugs in the code.[1] For example, if threads are transferred into a container and never transferred out (or vice-versa), then the scheduling logic is likely flawed.

[1] It would also be interesting to explore how to create CATAPULTS schedulers graphically with a GUI design tool. Such a tool would be especially attractive to engineers of mechanical systems who do not have a lot of experience with software development.

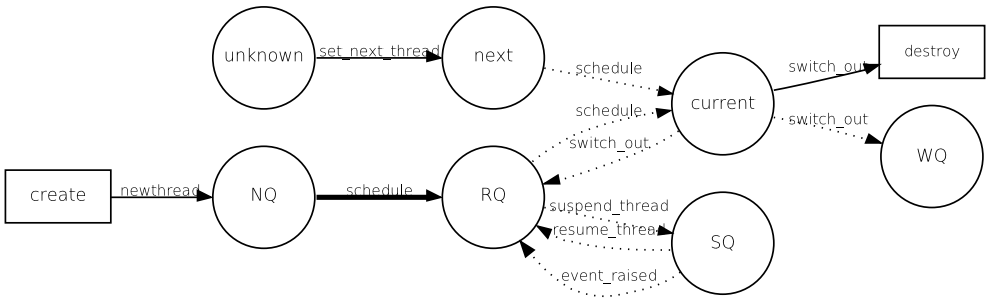



Figure 10.1: An example diagram generated by the CATAPULTS visualization backend

## 10.4 Extending CATAPULTS With New Backends

As described earlier, CATAPULTS comes with backends for both the GNU Pth and `cmthread.lib` threading libraries. Developing backends for additional threading libraries is relatively straightforward. The process consists of two stages: making minor modifications to the new target threading library and writing the backend driver code for CATAPULTS. These steps need be performed only once for a threading library and then any number of applications can be compiled against the modified threading library to take advantage of custom CATAPULTS schedulers.

Below, we discuss both of these steps and use a simple CM threading library (`libsimplecm` [8]), which we had already developed for other work, to illustrate the changes. The `libsimplecm` library is very similar to the `cmthread.lib` library we developed for embedded systems, but `libsimplecm` targets general purpose computers rather than embedded systems. Specifically, `libsimplecm`'s thread operations include creating, suspending, resuming, and destroying threads as well as named and unnamed yields between threads.

### 10.4.1 Library Modifications

The first step toward making it possible to use CATAPULTS schedulers with `libsimplecm` is to make some minor modifications to the `libsimplecm` implemen-

tation. As in most threading libraries, scheduling logic is intermixed with low-level thread manipulation code. For example, Figure 10.2 shows the C code to create a thread in the `libsimplecm` library.

```
thread_t thread_create_with_size(void (*func)(void*),
                                 void* parm, int stacksize) {
    Thread* newthread = (Thread*)malloc(sizeof(Thread));
    if (newthread == NULL)
        return NULL;
    newthread->state = NEW;

    newthread->userdata = NULL;
    newthread->coro = co_create(func, parm, NULL, stacksize);
    if (newthread->coro == NULL) {
        free(newthread);
        return NULL;
    }

    /* Add to ready queue */
    if (RQ) {
        newthread->next = RQ;
        newthread->prev = RQ->prev;
    } else {
        RQ = newthread->next = newthread->prev = newthread;
    }

    return (thread_t)newthread;
}
```

Figure 10.2: `libsimplecm` C code to create a thread

This function includes code to handle low-level thread creation (malloc'ing a thread structure, creating a coroutine, etc.) as well as scheduling logic (re-linking of pointers to add the thread to a ready queue). Fortunately, it is usually relatively easy to identify code that implements scheduling logic (even if no comments or documentation are available) by looking for blocks of code that manipulate internal thread container structures. Once the blocks of scheduling logic are identified, they can be replaced with function calls. For example, the code in Figure 10.2 might be changed to that in Figure 10.3. Note that the `sched_newthread` function in Figure 10.3 could be either a pointer to a function from

```
thread_t thread_create_with_size(void (*func)(void*),
                                 void* parm, int stacksize) {
    Thread* newthread = (Thread*)malloc(sizeof(Thread));
    if (newthread == NULL)
        return NULL;
    newthread->state = NEW;

    newthread->userdata = NULL;
    newthread->coro = co_create(func, parm, NULL, stacksize);
    if (newthread->coro == NULL) {
        free(newthread);
        return NULL;
    }

    /* Perform scheduling logic for new thread */
    sched_newthread(newthread);

    return (thread_t)newthread;
}
```

Figure 10.3: Modified `libsimplecm` C code to create a thread

a dynamically loaded scheduling library (i.e., the technique used by the Pth backend, described in Section 7.1), or the name of a scheduling function that will be statically compiled or linked into the threading library (as the `cmthread.lib` library does).

### 10.4.2 CATAPULTS Coding

Developing the actual CATAPULTS backend module is the most time-consuming aspect of adding CATAPULTS support for a new threading library; fortunately, this task need be performed only a single time. Each CATAPULTS backend target is implemented in a separate Python module. The existing backends for Pth and `cmthread.lib` can be used as a template for developing new backends. The backend module is responsible for implementing a `genCode()` function, which, when given a node of a scheduler's abstract syntax tree, will generate the appropriate implementation in the output scheduler. Although the `genCode()` routine can be implemented in many different ways, the method used by the Pth and `cmthread.lib` backends is very simple:

- Determine the type of AST node passed as a parameter (the frontend makes this available in a 'nodetype' field of the object).
- Lookup the node type in a 'code_gen_funcs' associative array to get a reference to the function responsible for generating code for that specific node type.
- Execute the code generation function, which will call `genCode()` recursively on its sub-nodes.

For example, Figure 10.4 shows the code generation function for an `if` statement.

```
def genIfStmt(ifs, lvalue):
    ret = "if (%s) " % genCode(ifs.cond)
    ret += genCode(ifs.truebranch)
    if ifs.falsebranch:
        ret += "else " + genCode(ifs.falsebranch)
    ret += "\n"

    return ret
```

Figure 10.4: Code generation function in CATAPULTS backend (a Python module) for an `if` statement

# Chapter 11

# Related Work

Very little work has been done in the area of domain-specific languages for writing schedulers. The most closely related project is Bossa [13], a system for generating Linux kernel schedulers using a domain-specific language. Although Bossa is similar in nature to CATAPULTS, it aims to solve a different set of problems. Since Bossa deals with operating system schedulers instead of application-level schedulers, its primary focus is on safety rather than performance or expressibility. In Bossa, all operations are guaranteed to be safe, but this limits the overall power of the language. For example, Bossa does not allow any form of unbounded loop; in contrast, CATAPULTS provides traditional `for`, `while`, and `do` loops for cases where a safer `foreach` loop does not suffice. Our compiler will generate a warning if it cannot be sure that the loop will terminate.

CATAPULTS also differs from Bossa in that Bossa is tightly coupled with a specific target language and platform (i.e., it generates Linux kernel C code). CATAPULTS allows different backends to be written for different target platforms and languages.

Modularizing scheduling code has also started to receive some attention from Linux kernel developers. A recent Linux kernel patch [30] separates all scheduling logic out into a separate kernel source file, thus making it much easier to replace the kernel scheduler. Although it appears that this pluggable scheduler framework is unlikely to be accepted into

the mainline kernel, it has received notable support and is being developed as an external patch to the kernel. This pluggable scheduler framework provides some of the benefits that systems such as Bossa or CATAPULTS do — modularization and ease of replacement — but lacks the portability and safety benefits that can be obtained from using a domain-specific language like CATAPULTS. Had it existed early enough, the pluggable scheduler framework would have been an excellent foundation on which to build Bossa or other kernel-based frameworks.

Other user-level scheduling work includes superd [36], a user-level daemon that provides coarse-grained process scheduling control for Unix systems, and scheduler activations [10], a framework by which an OS kernel can operate more efficiently with user-level threads. Superd's focus is on restricting the set of kernel-level processes that can run at a given time in order to guarantee that various process classes get specific shares of the processor time. Fine-grained scheduling of intra-application threads is not possible. Scheduler activations are a means by which an operating system kernel can provide notifications ("activations") to a multiprocessor-aware user-level threading library when scheduling decisions are to be performed, while still retaining control over physical processor allocation. Scheduler activations move all intra-application scheduling to user-space, so this approach to OS design would integrate nicely with CATAPULTS-generated schedulers.

Other applications of domain-specific languages for embedded systems include Hume [25]. Hume aims to provide a language for programming embedded systems that includes high-level features such as automatic memory management, exception handling, and polymorphic types, while guaranteeing application resource usage and timing behavior. Hume is intended for actual application development and although threads are provided, their scheduling cannot be changed from the builtin round-robin algorithm.

Our work with CATAPULTS in distributed environments is somewhat related to J-Orchestra [44], a system for partitioning centralized applications to run in a distributed manner across a network. Although J-Orchestra can split threads from the original cen-

tralized application across multiple nodes, it does not modify the scheduling algorithm used at individual nodes. Our work with CATAPULTS would complement J-Orchestra well; it would be interesting to integrate CATAPULTS into the J-Orchestra system to semi-automatically generate a distributed scheduler for a newly partitioned application.

# Chapter 12

# Conclusion and Future Work

Multi-threaded designs are becoming increasingly popular in modern computing. Despite exponential gains in computing power, efficiency and optimal use of system resources remain an important issue for both large scale applications (e.g., Internet servers that process thousands or millions of requests per day) and small scale embedded applications. Extensive work is done to optimize such systems for maximum performance. We feel that our work with CATAPULTS provides a key component of performance optimization that has previously been overlooked. Performance gains may vary depending on the complexity of the scheduling algorithm required by a given system and by the penalty incurred by inefficient thread scheduling, but we have shown that developing custom thread schedulers with CATAPULTS is relatively straightforward and can provide significant performance gains. Moreover, using CATAPULTS provides additional safety and portability that many other optimization techniques do not provide.

This dissertation presented the CATAPULTS language for writing application-specific schedulers. We examined the design of CATAPULTS in Chapter 3 and illustrated how our language allows modularity and portability while maintaining key safety properties. Chapters 4 and 5 provided an in-depth description of the CATAPULTS language features and introduced the syntax via a real-world example. The following chapters examined

our use of CATAPULTS in three important areas of computing: highly concurrent Internet server (Chapter 7), embedded applications (Chapter 8), and distributed systems (Chapter 9).

In the future it would be interesting to examine the partially automatic generation of schedulers by analyzing application source code, or by using genetic algorithms to narrow in on the optimal scheduling strategy for an application. We would also like to explore the area of schedulers for embedded systems more deeply and allow CATAPULTS schedulers to guarantee soft or hard realtime requirements. Our work with distributed schedulers could also be extended to more systems, especially those with true distribution across a network.

Another area that we plan to explore is applying CATAPULTS to applications that use kernel-level threads instead of user-level threads. Since the majority of existing multi-threaded software makes use of the pthreads API (which on modern systems is implemented with kernel threads), all such existing applications would now be able to make use of application-specific CATAPULTS schedulers. To do this, a CATAPULTS backend would be developed that generated the source code for a Loadable Kernel Module (LKM) for the Linux operating system. This module could then be loaded into a running Linux kernel and intercept any process scheduling decisions made by the main Linux kernel scheduler. If the process/thread selected for execution belongs to a "group" of threads being managed by the kernel module (where a "group" would represent the threads of a given application), the module would perform its own scheduling algorithm to select a different thread from that group, if appropriate. This would allow an application to specify the scheduling of its own threads without affecting the scheduling of the rest of the processes/threads running on the system. This approach differs from the Bossa project [13] in that Bossa generates entire OS schedulers while CATAPULTS would be generating sub-schedulers that override the primary scheduler's decision for small groups of threads.

We would also like to work on additional tools to assist in the development of CATAPULTS schedulers. We already have a scheduler visualization tool (described in Section 10.3), but we would like to take the idea a step farther and allow the development of

schedulers through a GUI interface. Such a tool would be of interest to hardware engineers of embedded systems who do not have very strong programming backgrounds.

# Appendix A

# CATAPULTS Language Specification

This appendix provides an EBNF specification for the CATAPULTS grammar.

## A.1 Classic CATAPULTS

The following is the grammar for classic CATAPULTS.

<*scheduler_def*> ::= [ **"partial"** ] **"scheduler"** <*id*> [ **"extends"** <*id*> ] **"{"** [ <*threadblock*> ]
[ <*thrimportblock*> ] [ <*datablock*> ] [ <*importblock*> ] {<*invariant*>}*
{<*event_or_query*>}* {<*aspect*>}* **"}"**

<*threadblock*> ::= **"thread"** **"{"** {<*thread_declaration*>} **"}"**

<*thread_declaration*> ::= **"int"** <*id*> **";"** | **"float"** <*id*> **";"**

<*datablock*> ::= **"data"** **"{"** {<*declaration*>}* **"}"**

<*importblock*> ::= **"imports"** **"{"** {<*import_declaration*>}* **"}"**

<*thrimportblock*> ::= **"threadimports"** **"{"** {<*import_declaration*>}* **"}"**

<*declaration*> ::= <*declaration_specifiers*> <*init_declaration*> {, <*init_declarator*>}* **";"**

<*import_declaration*> ::= <*type_specifier*> <*id*> **"default"** <*expression*> **";"**

<*declaration_specifiers*> ::= [ **"const"** ] <*type_specifier*>

<*init_declarator*> ::= <*id*> [ **"="** <*assignment_expression*> ] | <*id*> [ <*sorttype*> ] **"sortable"** **"on"** <*id*>

<*sorttype*> ::= **"reverse"** | **"increasing"** | **"decreasing"**

<*init_declarator*> ::= <*id*> **"["** <*expression*> **"]"**

<*type_specifier*> ::= **"int"** | **"float"** | **"threadref"** | **"queue"** | **"stack"** | **"pqueue"** | **"pstack"** | **"time"**

<*event_or_query*> ::= <*event_or_query_keyword*> <*id*> [ **"("** <*param*> **")"** ] <*compound_statement*>

<*event_or_query_keyword*> ::= **"event"** | **"query"**

<*invariantcode*> ::= **"invariant"** <*id*> <*compound_statement*>

<*aspect*> ::= <*aspect_keyword*> <*id*> [ **"("** <*param*> **")"** ] <*compound_statement*>

<*aspect_keyword*> ::= **"before"** | **"after"**

<*return_statement*> ::= **"return"** [ <*expression*> ] **";"**

<*verbatim_statement*> ::= **"{*"** ... **"*}"**

<*param*> ::= <*id*> | **"fixed"** <*id*>

<*constant*> ::= <*fconst*> | <*tconst*> | **"true"** | **"false"**

<*variable*> ::= <*id*> | <*id*> **"["** <*expression*> **"]"**

<*statement*> ::= <*expression_statement*>
| <*selection_statement*>
| <*iteration_statement*>
| <*jump_statement*>
| <*dispatch_statement*>
| <*transfer_statement*>
| <*compound_statement*>
| <*return_statement*>
| <*destroy_statement*>
| <*verbatim_statement*>

<*transfer_statement*> ::= <*variable*> **"=>"** <*variable*> **";"**

<*expression_statement*> ::= [ <*expression*> ] **";"**

<*compound_statement*> ::= **"{"** {<*declaration*>}* {<*statement*>}* **"}"**

<*selection_statement*> ::= **"if"** **"("** <*expression*> **")"** <*statement*> [ **"else"** <*statement*> ]

<*iteration_statement*> ::= **"while"** **"("** <*expression*> **")"** <*statement*>
| **"for"** **"("** [ <*expression*> ] **";"** [ <*expression*> ] **";"** [ <*expression*> ] **")"** <*statement*>
| **"do"** <*statement*> **"while"** **"("** <*expression*> **")"** **";"**
| **"foreach"** <*id*> **"in"** <*variable*> <*statement*>

<*jump_statement*> ::= **"continue"** **";"** | **"break"** **";"**

<*dispatch_statement*> ::= **"dispatch"** <*variable*> **";"**

<*destroy_statement*> ::= **"destroy"** <*variable*> **";"**

<*expression*> ::= <*assignment_expression*>

<*assignment_expression*> ::= <*logical_or_expression*>
| <*postfix_expression*> <*assignment_operator*> <*assignment_expression*>

<*assignment_operator*> ::= **"="** | **"*="** | **"/="** | **"%="** | **"+="** | **"-="**

<*logical_or_expression*> ::= <*logical_and_expression*>

<*logical_or_expression*> ::= <*logical_or_expression*> **"||"** <*logical_and_expression*>

<*logical_and_expression*> ::= <*equality_expression*>

<*logical_and_expression*> ::= <*logical_and_expression*> **"&&"** <*equality_expression*>

<*equality_expression*> ::= <*relational_expression*>
| <*equality_expression*> **"=="** <*relational_expression*>
| <*equality_expression*> **"!="** <*relational_expression*>

<*relational_expression*> ::= <*additive_expression*>
| <*relational_expression*> **"<"** <*additive_expression*>
| <*relational_expression*> **">"** <*additive_expression*>
| <*relational_expression*> **"<="** <*additive_expression*>
| <*relational_expression*> **">="** <*additive_expression*>

<*additive_expression*> ::= <*multiplicative_expression*>
| <*additive_expression*> **"+"** <*multiplicative_expression*>
| <*additive_expression*> **"–"** <*multiplicative_expression*>

<*multiplicative_expression*> ::= <*unary_expression*>
| multiplicative_expression **"*"** unary_expression
| multiplicative_expression **"/"** unary_expression
| multiplicative_expression **"%"** unary_expression

<*unary_expression*> ::= <*postfix_expression*>
| **"++"** <*unary_expression*>
| **"– –"** <*unary_expression*>
| **"-"** <*unary_expression*>
| **"!"** <*unary_expression*>
| **"|"** <*variable*> **"|"**
| **"*"** <*variable*>

<*postfix_expression*> ::= <*primary_expression*>
| <*variable*> **"."** <*id*>
| <*postfix_expression*> **"++"**
| <*postfix_expression*> **"–"**

<*primary_expression*> ::= <*constant*> | <*variable*> | **"("** <*expression*> **")"**" | **"(*"** . . . **"*)"** | <*iconst*>

## A.2 Distributed CATAPULTS

The following rules are added or replaced when compiling a distributed scheduler.

<*scheduler_def*> ::= **"distributed" "scheduler"** <*id*> **"{"** <*familyblock*> <*nodesblock*> <*globalblock*>
<*arrivalblock*> <*departureblock*> <*event_or_query_list*> **"}"**

<*familyblock*> ::= **"nodefamilies" "{"** <*familydeflist*> **"}"**

<*familydef*> ::= <*filename*> **":"** <*id*> **";"**

<*nodesblock*> ::= **"nodes" "{"** {<*nodedecl*>}* **"}"**

<*nodedecl*> ::= **"node"** <*nodedeclvar*> **";"**
| **"stack"** <*nodedeclvar*> **";"**
| **"queue"** <*nodedeclvar*> **";"**
| **"pqueue"** <*nodedeclvar*> **"sortable" "on"** <*id*> **";"**
| **"pstack"** <*nodedeclvar*> **"sortable" "on"** <*id*> **";"**

<*nodedeclvar*> ::= <*id*> [ **"["** <*iconst*> **"]"** ]

<*globalblock*> ::= **"global" "{"** <*declaration_list*> **"}"**

<*type_specifier*> ::= **"int"** | **"float"** | **"threadref"** | **"queue"** | **"stack"** | **"pqueue"** | **"pstack"** | **"time"** | **"node"**

<*statement*> ::= <*expression_statement*>
| <*selection_statement*>
| <*iteration_statement*>
| <*jump_statement*>
| <*select_statement*>
| <*transfer_statement*>
| <*compound_statement*>
| <*verbatim_statement*>

<*arrivalblock*> ::= **"arrival" "("** <*id*> **")"** <*compound_statement*>

<*departureblock*> ::= **"departure" "("** <*id*> **")"** <*compound_statement*>

<*logical_or_expression*> ::= <*variable*> **"is"** <*variable*>

# Appendix B

# Scheduler for CoW Proxy on Linux

The following figures provide the scheduler used for the CoW proxy server discussed in Section 7.2.1.

```
/*
 * cow.sched
 *
 * A specialized scheduler for CoW (the COoperative-multithreading Webserver)
 */

scheduler cow {
    thread {
        int state;
        int ismain;
        int bytesleft_internal; /* copied from thread import */
    }

    threadimports {
        int isresponding default 0;       /* handler in response state? */
        int bytesleft default 0;          /* response bytes remaining */
    }

    data {
        threadref current;      /* current thread */
        threadref next;         /* next thread (named yield) */
        queue RQ;               /* ready threads */
        pqueue RQ2 reverse sortable on bytesleft_internal; /* response queue */
        queue WQ;               /* waiting for event */
        queue SQ;               /* suspended */

        int created_main = 0;   /* Have we spawned main yet? */

        const int NEW = 1;
        const int READY = 2;
        const int WAITING = 3;
        const int DEAD = 4;
        const int SUSPENDED = 5;
    }

    imports {
        int noof default 0;               /* Number of open files */
        int idlehands default 1;          /* Number of idle handlers */
    }


    event init {
        created_main = 0;
        {* fprintf(stderr, "CoW scheduler initialized\n"); *}
    }
```

Figure B.1: First part of CoW scheduler

```
event schedule {
    /* Find next thread to run */
    if (|next| == 1)
        next => current;
    else if (|RQ2| > 0)
        RQ2 => current;
    else {
        threadref tmp;

        RQ => tmp;
        if (tmp.ismain == 1 && (noof >= 1000 || idlehands == 0)) {
            /*
             * This is the main thread, but we have too many files open
             * to accept any new connections.  Just move it back to the
             * end of the RQ and schedule the next one instead.
             */
            tmp => RQ;
            RQ => tmp;
        }

        tmp => current;
    }
    dispatch current;
}

event newthread(t) {
    if (created_main == 0) {
        t.ismain = 1;
        created_main = 1;
    } else {
        t.ismain = 0;
    }

    /*
     * Unlike the PTH scheduler, we don't keep a separate NQ; we just
     * stick threads on the RQ immediately.
     */
    t.state = NEW;
    t => RQ;
}
```

Figure B.2: Second part of CoW scheduler

```
event switch_out {
    int resp;

    if (1 == (resp = current.isresponding))
        /*
         * Copy thread import into an internal variable that RQ2 can be
         * sorted on
         */
        current.bytesleft_internal = current.bytesleft;

    /* Did the last thread end? */
    if (current.state == DEAD)
        destroy current;
    else if (current.state == WAITING)
        current => WQ;
    else if (resp == 1)
        current => RQ2;
    else
        current => RQ;
}

event event_raised(t) {
    t.state = READY;
    if (t.isresponding == 1)
        t => RQ2;
    else
        t => RQ;
}

event set_next_thread(t) {
    t => next;
}

query threads_new       { return 0; }
query threads_ready     { return |RQ| + |RQ2|; }
query threads_waiting   { return |WQ|; }
query threads_suspended { return |SQ|; }
query threads_total     { return |RQ| + |RQ2| + |WQ| + |SQ| + 1; }

event set_waiting(fixed tid) {
    tid.state = WAITING;
}
```

Figure B.3: Third part of CoW scheduler

```
event set_dead(fixed tid) {
    tid.state = DEAD;
}

query is_new(tid) {
    return tid.state == NEW;
}

query is_ready(tid) {
    return tid.state == READY;
}

query is_waiting(tid) {
    return tid.state == WAITING;
}

query first_waiting {
    return *WQ;
}

query can_switch_to(tid) {
    return (tid.state == NEW || tid.state == READY);
}

event suspend_thread(tid) {
    tid.state = SUSPENDED;
    tid => SQ;
}

event resume_thread(tid) {
    tid.state = READY;
    if (tid.isresponding == 1)
        tid => RQ2;
    else
        tid => RQ;
}
```

Figure B.4: Fourth part of CoW scheduler

# Appendix C

# Scheduler for MySQL Database

The following figures provide the scheduler used for the MySQL database server discussed in Section 7.2.2.

```
scheduler pth {
    thread {
        int state;
        int is_neworder;
    }

    data {
        threadref current;      /* current thread */
        threadref next;         /* next thread (named yield) */
        queue reg_ready;        /* regular ready threads */
        queue new_order_ready;  /* new order ready threads */
        queue WQ;               /* waiting for event */
        queue SQ;               /* suspended */

        int table_list_internal = 0;
        int field_list_internal = 0;

        const int READY = 1;
        const int WAITING = 2;
        const int DEAD = 3;
        const int SUSPENDED = 4;
    }

    imports {
        int table_list default 0;
        int field_list default 0;
    }

    event init {
        /* no-op */
    }

    event newthread(t) {
        t.state = READY;
        t.is_neworder = 0;
        t => reg_ready;
    }
```

Figure C.1: First part of MySQL scheduler

```
event schedule {
    /* Find next thread to run */
    if (|next| == 1)
        next => current;
    else if (|new_order_ready| > 0)
        new_order_ready => current;
    else
        reg_ready => current;
    dispatch current;
}

event switch_out {
    /* Copy imported variable for easier use in verbatim statements */
    table_list_internal = table_list;
    field_list_internal = field_list;

    /* Transaction start trigger? */
    if ((* field_list_internal &&
            0 == strncmp((char* )field_list_internal, "w_tax", 5) *))
        current.is_neworder = 1;

    /* Find finish trigger */
    if ((current.is_neworder == 1) &&
            (* field_list_internal &&
               0 == strncmp((char* )field_list_internal, "1", 1) *) != 0)
        current.is_neworder = 0;

    /* Cleanup */
    {* *((char*)field_list_internal) = *((char*)table_list_internal) = '\0'; *}
    field_list = field_list_internal;
    table_list = table_list_internal;

    /* Did the last thread end? */
    if (current.state == DEAD) {
        destroy current;
    } else if (current.state == WAITING) {
        current => WQ;
    } else if (current.is_neworder == 1) {
        current => new_order_ready;
    } else {
        current => reg_ready;
    }
}

event event_raised(t) {
    t.state = READY;
    if (t.is_neworder == 1)
        t => new_order_ready;
    else
        t => reg_ready;
}
```

Figure C.2: Second part of MySQL scheduler

```
event set_next_thread(t) {
    t => next;
}

query threads_new       { return 0; }
query threads_ready     { return |new_order_ready| + |reg_ready|; }
query threads_waiting   { return |WQ|; }
query threads_suspended { return |SQ|; }
query threads_total     { return |new_order_ready| + |reg_ready| + |WQ| + |SQ| + 1; }

event set_waiting(fixed tid) {
    tid.state = WAITING;
}

event set_dead(fixed tid) {
    tid.state = DEAD;
}

query is_new(tid) { return false; }

query is_ready(tid) {
    return tid.state == READY;
}

query is_waiting(tid) {
    return tid.state == WAITING;
}

query first_waiting {
    return *WQ;
}

query can_switch_to(tid) {
    return (tid.state == READY);
}

event suspend_thread(tid) {
    tid.state = SUSPENDED;
    tid => SQ;
}

event resume_thread(tid) {
    tid.state = READY;
    if (tid.is_neworder == 1)
        tid => new_order_ready;
    else
        tid => reg_ready;
}
```

Figure C.3: Third part of MySQL scheduler

# Appendix D

# Scheduler for Embeddded CoW Web Server

The following figures provide the scheduler used for the embedded CoW web server discussed in Section 8.2.2.

```
/*
 * embcow.sched
 *
 * An application-specific scheduler for the embedded version of the CoW web
 * server.  The following optimizations are performed:
 *   - scheduler tests socket statuses of blocked threads immediately after
 *     each TCP thread run; blocked threads are not checked again until
 *     they're moved back to the RQ
 */

scheduler embcow {
    thread {
        int state;          // running, suspended, etc.

        // These were thread imports, but they don't have to be.
        int threadclass;  // handler, tcp, timeout
        int action;       // reading, writing, listening, working, etc.
    }

    data {
        threadref current;    // current thread

        queue RQ;
        queue SQ;             // suspended
        queue READQ;          // reading queue
        queue WRITEQ;         // writing queue
        queue LQ;             // listening queue

        const int HANDLER = 1, TCP = 2, TIMEOUT = 3;
        const int READY = 1, SUSPENDED = 2, BLOCKED = 3;
        const int WORKING = 1, LISTENING = 2, READING = 3, WRITING = 4, CLOSING = 5;
    }

    event init { /* noop */ }

    event newthread(t) {
        t.state = READY;
        t => RQ;
    }

    event schedule {
        RQ => current;
        dispatch current;
    }
```

Figure D.1: First part of embedded CoW scheduler

```
event switch_out {
    threadref iterthr;

    // Figure out which thread it was.
    if (current.threadclass == TIMEOUT) {
        current => RQ;
    } else if (current.threadclass == TCP) {
        // We just ran the TCP thread, so we need to see if we have any new
        // data on any of the sockets that are reading.
        foreach iterthr in READQ {
            if ((* sock_bytesready(iterthr->userdata) *) > 0 ||
                    (* sock_alive(iterthr->userdata) *) == 0 )
                iterthr => RQ;
        }
        foreach iterthr in WRITEQ {
            if ((* sock_tbused(iterthr->userdata) *) == 0 )
                iterthr => RQ;
        }
        foreach iterthr in LQ {
            if ((* sock_established(iterthr->userdata) *) != 0 ||
                    (* sock_alive(iterthr->userdata) *) == 0 )
                iterthr => RQ;
        }

        current => RQ;
    } else {
        // What's the thread doing right now?  If it's reading, we
        // don't want to put it back on the RQ since it can't run
        // again until at least after the TCP thread has run again
        // (and even then only if it actually received more data).
        // We'll put it on the reading queue instead.
        if (current.action == READING)
            current => READQ;

        else if (current.action == WRITING)
            current => WRITEQ;

        // If the thread is listening, put it on a listening queue.
        // Again, this can only be unblocked after the TCP thread
        // has run, but different conditions are required to move it
        // to the RQ.
        else if (current.action == LISTENING)
            current => LQ;

        // It must be working.  Keep it on the ready queue.
        else
            current => RQ;
    }
}

event suspend_thread(tid) {
    tid.state = SUSPENDED;
    tid => SQ;
}

event resume_thread(tid) {
    tid.state = READY;
    tid => RQ;
}
```

Figure D.2: Second part of embedded CoW scheduler

# Appendix E

# Scheduler for Balanced Server

The following figures provide the scheduler for the distributed embedded application described in Section 9.2.2.

```
distributed scheduler distemb {
    nodefamilies {
        "./camera.subsched" : camera;
        "./webnode.subsched" : web;
        "./tempbuff.subsched" : tempbuff;
    }

    nodes {
        queue cameras;
        queue webnodes;
        node buff;
    }

    global {
        int buff_remaining;
    }

    arrival(n) {
        if (n is tempbuff) {
            n => buff;
        } else if (n is camera) {
            n => cameras;
        } else if (n is web) {
            n => webnodes;
        }
    }

    departure(x) {
        /* noop */
    }
}
```

Figure E.1: Master scheduler for distributed embedded example

```
scheduler buff extends pth {
    imports {
        int buffleft default 0;
        int bufftot default 1;
    }

    before schedule {
        GLOBAL.buff_remaining = 100 * buffleft / bufftot;
    }
}
```

Figure E.2: Node scheduler for calculation node/temporary buffer

```
scheduler webnode extends pth {
    thread {
        int isaccept;
    }

    event schedule {
        /* Move all new threads to ready queue */
        while (|NQ| > 0)
            NQ => RQ;

        /* Find next thread to run */
        if (|next| == 1)
            next => current;
        else
            RQ => current;

        if (current.isaccept == 1) {
            // This is the accept thread.  Chance to actually run it
            // depends on remaining space on temp buffer.
            int r = (* rand() *) % 100;

            if (r > GLOBAL.buff_remaining) {
                // Don't let accept thread run right now;
                // choose something else
                current => RQ;
                RQ => current;
            }
        }

        dispatch current;
    }

    before newthread(n) {
        if (|RQ| + |NQ| + |WQ| + |next| == 0)
            n.isaccept = 1;
    }
}
```

Figure E.3: Node scheduler for web server node

# Appendix F

# Scheduler for Balanced Server

The following figures provide the load balancing scheduler discussed in Section 9.2.3.

```
distributed scheduler balanced {
    nodefamilies {
        "./balanced.subsched" : balnode;
    }

    nodes {
        queue bnodes;
    }

    global {
        int total_threads, total_nodes;
    }

    arrival(n) {
        n => bnodes;
        total_nodes++;
    }
    departure(x) { /* noop */ }
}
```

Figure F.1: Master scheduler for balanced server

```
scheduler balanced_node {
    thread {
        int state;
        int is_accept;
    }

    data {
        threadref current;      /* current thread */
        threadref next;         /* next thread (named yield) */
        queue NQ;               /* new threads */
        queue RQ;               /* ready threads */
        queue WQ;               /* waiting for event */
        queue SQ;               /* suspended */

        int numthreads;

        int now = 10;
        int NEW = 1;
        int READY = 2;
        int WAITING = 3;
        int DEAD = 4;
        int SUSPENDED = 5;

        /* Balancing point (max number of threads over average allowed) */
        int THRESHOLD = 3;
    }

    event init {
        numthreads = 0;
    }

    event newthread(t) {
        if (numthreads == 1)
            t.is_accept = 1;
        else
            t.is_accept = 0;

        t.state = NEW;
        t => NQ;
        numthreads++;
        total_threads++;
    }
```

Figure F.2: First part of nodespec for balanced server

```
event schedule {
    /* Move all new threads to ready queue */
    while (|NQ| > 0)
        NQ => RQ;

    /* Find next thread to run */
    if (|next| == 1)
        next => current;
    else
        RQ => current;

    // If this is the accept thread, check system balance
    if (current.is_accept == 1) {
        if (total_threads/total_nodes - numthreads > THRESHOLD) {
            // Put it back and grab another thread
            current => RQ;
            RQ => current;
        }
    }

    dispatch current;
}

event switch_out {
    /* Did the last thread end? */
    if (|current| == 0) {
        int dummy;

        // noop...last thread suspended itself and is already removed
        // from current
        dummy = 0;
    } else if (current.state == DEAD)
        destroy current;
    else if (current.state == WAITING)
        current => WQ;
    else
        current => RQ;
}

event event_raised(t) {
    t.state = READY;
    t => RQ;
}

event set_next_thread(t) {
    t => next;
}
```

Figure F.3: Second part of nodespec for balanced server

```
query threads_new       { return |NQ|; }
query threads_ready     { return |RQ|; }
query threads_waiting   { return |WQ|; }
query threads_suspended { return |SQ|; }
query threads_total     { return |RQ| + |NQ| + |WQ| + |SQ| + 1; }

event set_waiting(fixed tid) {
    tid.state = WAITING;
}

event set_dead(fixed tid) {
    tid.state = DEAD;
    numthreads--;
    total_threads--;
}

query is_new(tid) {
    return tid.state == NEW;
}

query is_ready(tid) {
    return tid.state == READY;
}

query is_waiting(tid) {
    return tid.state == WAITING;
}

query first_waiting {
    return *WQ;
}

query can_switch_to(tid) {
    return (tid.state == NEW || tid.state == READY);
}

event suspend_thread(tid) {
    tid.state = SUSPENDED;
    tid => SQ;
}

event resume_thread(tid) {
    tid.state = READY;
    tid => RQ;
}
```

Figure F.4: Third part of nodespec for balanced server

# Bibliography


[1] The Apache HTTP server project. http://httpd.apache.org/.

[2] Dynamic C user's manual. http://www.zworld.com/documentation/docs/manuals/DC/DCUserManual/index.htm.

[3] Graphviz – graph visualization software. http://www.graphviz.org/.

[4] MySQL. http://www.mysql.org.

[5] OpenLDAP. http://www.openldap.org.

[6] OSDL database test 2. http://www.osdl.org/lab_activities/kernel_testing/osdl_database_test_suite/osdl_dbt-2/.

[7] POSIX threads programming. http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html.

[8] SimpleCM threading library. http://www.cs.ucdavis.edu/~roper/catapults/simplecm/.

[9] Mustaque Ahamad, Phillip W. Hutto, Gil Neiger, James E. Burns, and Prince Kohli. Causal memory: Definitions, implementation and programming. Technical Report GIT-CC-93/55, Georgia Institute of Technology, 1994.

[10] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[11] G.R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison Wesley Longman, Inc., Reading, MA, 2000.

[12] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[13] L. Barreto and G. Muller. Bossa: A language-based approach for the design of real time schedulers. In *10th International Conference on Real-Time Systems (RTS)*, pages 19–31, 2002.

[14] David Beazley. PLY (Python Lex-Yacc). http://systems.cs.uchicago.edu/ply/.

[15] R. Behren, J. Condit, F. Zhou, G. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *Proceedings of the Ninteenth Symposium on Operating System Principles (SOSP)*, 2003.

[16] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. Technical report, Pittsburgh, PA, USA, 1993.

[17] S. R. Cannon and D. A. Brinkerhof. A stable distributed tuple space. In *HICSS '96: Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*, page 22, Washington, DC, USA, 1996. IEEE Computer Society.

[18] Mark E. Crovella, Robert Frangioso, and Mor Harchol-Balter. Connection scheduling in Web servers. In *1999 USENIX Symposium on Internet Technologies and Systems (USITS '99)*, Berkeley, CA, USA, 1999. USENIX Association. http://www.sagecertification.org/publications/library/proceedings/usits99/full_papers/crovella/crovella.pdf.

[19] Jeff Dike. A user-mode port of the Linux kernel. In *2000 Linux Showcase and Conference*, October 2000.

[20] Larry Doolittle and Jon Nelson. BOA web server, 2004. http://www.boa.org/.

[21] Ulrich Drepper and Ingo Molnar. The native POSIX thread library for Linux. http://people.redhat.com/drepper/nptl-design.pdf, 2003.

[22] Ralf S. Engelschall. GNU Pth - the GNU Portable Threads. http://www.gnu.org/software/pth/.

[23] Stephen Ferg. Event-driven programming: Introduction, tutorial, history. http://eventdrivenpgm.sourceforge.net/.

[24] David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[25] Kevin Hammond and Greg Michaelson. Hume: a domain-specific language for real-time embedded systems. In *Proceedings of the Second International Conference on Generative Programming and Component Engineering*, pages 37–56, 2003.

[26] Takashi Ishihara and Matthew Roper. CoW: A cooperative multithreading web server. http://www.cs.ucdavis.edu/~roper/cow/.

[27] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, 1994.

[28] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, 1992.

[29] Rawlson King. The emergence of server virtualization. http://www.thewhir.com/features/virtualization.cfm.

[30] Con Kolivas. Pluggable CPU scheduler framework, October 2004. http://groups-beta.google.com/group/fa.linux.kernel/msg/891f15d63e5f529d.

[31] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programm. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[32] Jonathan Lemon. Kqueue: A generic and scalable event notification facility. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, 2001.

[33] Davide Libenzi. /dev/epoll home page. http://www.xmailserver.org/linux-patches/nio-improve.html.

[34] Norman S. Matloff. PerlDSM: A distributed shared memory system for Perl. In *PDPTA '02: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 63–68. CSREA Press, 2002.

[35] David Mosberger and Tai Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67. ACM, June 1998. http://www.hpl.hp.com/personal/David_Mosberger/httperf.ps.

[36] Travis Newhouse and Joseph Pasquale. A user-level framework for scheduling within service execution environments. *Proc. of the 2004 IEEE International Conference on Services Computing*, pages 311–318, September 2004.

[37] Matthew Roper. Dynamic threading and scheduling with Dynamic C. http://www.cs.ucdavis.edu/~roper/dcdynthread/.

[38] Matthew Roper. UCDTri web calendar and scheduling system. http://www.ucdtri.com/workouts/monthview.php.

[39] Matthew Roper and Ronald A. Olsson. Developing embedded multi-threaded applications with CATAPULTS, a domain-specific language for generating thread schedulers. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2005. http://www.cs.ucdavis.edu/~roper/catapults/cases2005.pdf.

[40] Antony I. T. Rowstron and Alan Wood. An efficient distributed tuple space implementation for networks of workstations. In *Euro-Par '96: Proceedings of the Second International Euro-Par Conference on Parallel Processing*, pages 510–513, London, UK, 1996. Springer-Verlag.

[41] H. S. Sandhu, T. Brecht, and D. Moscoso. Multiple writers entry consistency. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, volume I, pages 355–362, 1998.

[42] Weisong Shi, Weiwu Hu, Zhimin Tang, and M. Rasit Eskicioglu. Dynamic task migration in home-based software DSM systems. In *Proc. of the Eighth IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-8)*, number 990004, Beijing, China, 1999.

[43] Mike Stevens. Linker hijacking. http://neworder.box.sk/newsread.php?newsid=4735.

[44] Eli Tilevich and Yannis Smaragdakis. J-Orchestra: Automatic Java application partitioning. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 178–204. Springer-Verlag, LNCS 2374, 2002.

[45] S. Walton. *LinuxThreads*, 1997. http://www.ibiblio.org/pub/Linux/docs/faqs/Threads-FAQ/html/.