Application-specific User-Level Thread Schedulers

Qualifying Examination Paper

Matthew D. Roper

roper@cs.ucdavis.edu

May, 2005

Department of Computer Science

University of California, Davis

Application-specific User-Level Thread Schedulers

Matthew D. Roper

roper@cs.ucdavis.edu

Abstract

This paper 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 our preliminary implementation of CATAPULTS for GNU Pth, we were able to significantly improve both the throughput and response time of CoW, a multi-threaded web server. We also applied our embedded system version of CATAPULTS to a typical embedded application and obtained a performance gain of about 12.6% at the expense of about 12.7% increase in code size. Future directions for the CATAPULTS project include extending the language to facilitate the development of schedulers for both realtime applications and distributed applications, as well as developing a metaprogramming model for generating CATAPULTS grammars specialized for a given threading package (GNU Pth, cmthread.lib, etc.) and application class (classic multithreaded, realtime, distributed, etc.).

Contents

List of Figures

List of Tables

1 Introduction

Multi-threaded designs have become a popular architecture for highly concurrent applications, such as Internet servers. Well-known examples of such multi-threaded applications include the Apache 2.0 web server [1], MySQL [2], and OpenLDAP [3]. 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.

This paper introduces CATAPULTS, a system for developing application-specific schedulers for user space threading libraries. 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. Our solution is to provide a domain-specific language for writing application-specific schedulers. This approach has 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. 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 primary target for CATAPULTS is highly concurrent Internet servers (web, database, mail, etc.) that run on uniprocessor machines since these applications will gain the most benefit from custom schedulers. Another important area that CATAPULTS has been applied to is applications for embedded systems has been applied to. We intend to further develop the CATAPULTS language to provide mechanisms for the development of schedulers for realtime applications and also facilitate the creation of schedulers for distributed multithreaded applications. Because realtime and distributed schedulers require significantly different features beyond what a traditional multithreaded application needs, we also intend to explore the use of metaprogramming to create dialects of the CATAPULTS language specialized for a specific application domain.

The rest of this paper is organized as follows. Section 2 presents background material on design of concurrent applications and on threading alternatives and gives an overview of related work. Section 3 provides an introductory example of a CATAPULTS scheduler. Section 4 describes the

purpose and design of the CATAPULTS system. Section 5 provides details on the CATAPULTS domain-specific language. Section 6 describes how the components of the system were implemented. Section 7 describes our experience using CATAPULTS on several applications. Section 8 provides a brief summary of our existing work and preliminary results. Section 9 discusses future directions for CATAPULTS, including distributed application scheduling, realtime scheduling, and using metaprogramming to generate a CATAPULTS grammar and parser that contains the exact subset of thread features required by a particular scheduler target. Section 10 provides a timeline for completion of the future work.

2 Background and Related Work

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.

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 [4] or Kqueue on FreeBSD [5]) can provide much better performance. Finally, event-driven 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; threaded applications can even match or exceed the performance of event-driven applications [6]. 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

[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.

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 optimal 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]. The 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 [8] 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) [9], but the interface with applications remained the Pthreads API. And user space libraries can also provide implementations of the Pthreads API; GNU Pth [10], 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 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 [11], 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 [12] 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 [13], a user-level daemon that provides coarse-grained process scheduling control for Unix systems; scheduler activations [14], a framework by which an OS kernel can operate more efficiently with user-level threads; and QuickThreads [15], a toolkit for building thread packages that allows arbitrary scheduling code. Superd focuses 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. QuickThreads is a toolkit that provides a very low-level set of thread operations for context switching between threads. Higher-level concerns, such as scheduling or stack allocation, are left to the threading library or

application, which makes it possible to develop application-specific scheduling routines.

3 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 [16], an extended subset of C that runs on Z-World's 8-bit Rabbit processors (Dynamic C and our implementation of CATAPULTS on it are discussed in Section 6.3). 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 conditions occur, the Dynamic C scheduler has no such knowledge. It schedules the threads in an inefficient manner, resulting 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 1 through 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 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.).

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

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

Figure 1: Thread attribute declarations

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 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.

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

Figure 2: Thread import declarations

The scheduler specification must also include declarations for any global objects used by the scheduler, including both global variables and constants of primitive types (i.e., integers and floats) and thread collections (queues, stacks, etc., provided by the runtime system; see Section 5.1).

Figure 3 shows this data declaration section for our example scheduler. Several thread collections are declared to hold different classes of threads: new (i.e., just created) threads are placed on a stack, 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.

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

```
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 3: Global data declarations

or not the display output thread should be run.

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

Figure 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 Section 6.3). 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 5-8 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 1 through 8, the developer then runs the CATAPULTS translator on the specification. It produces a scheduler targeted for 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

```
event init {
  last_display = 0;
}

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

Figure 5: Event handlers to initialize the scheduler and handle new thread creation events

```
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
  if (|next| == 1) {  // |next| = size of next
    next => current;  //   (|next| is 1 or 0 here)
  } 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 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 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 8: An example query handler that returns to the base threading library the number of threads currently ready to run in the system

different CATAPULTS backend to generate scheduling code for whatever language and library was being used for the prototype.

4 Design

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 ap-

plication 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.

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 section discusses these three goals. A fourth important goal, good performance, is implicit in the design decisions made for CATAPULTS and is discussed in Section 7.

4.1 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 (such as 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.

4.2 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 very simple (yet seemingly sufficient for most schedulers in our experience) set of data structures for storing collections of threads: threadrefs, stacks, queues, and lists that are sortable on any thread attribute (e.g., priority). 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.

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.

4.3 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 backend code generators for Dynamic C (a C-like language with threading features that is used to program ZWorld's embedded Rabbit controllers [16]) and GNU Pth [10] (a powerful cooperative threading library for PC's); writing more backends for other languages will be 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 must be specified 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.

5 Language Details

This section provides details about the language mechanisms used in the example in Section 3 and also introduces a few additional features that were not used in the example. This section can safely be skipped without affecting understanding of the rest of the paper.

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 4.2, 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

(Section 3). 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 Section 3, 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 2 and used in Figures 6 and 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 [17] to use a CATAPULTS-generated scheduler for GNU Pth (as described in Section 7.1), 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).

6 Implementation

This section describes the implementation of the CATAPULTS frontend and both of its existing backends. As noted earlier, CATAPULTS supports multiple backend code generators. Each is written as a separate module, which makes it easy to add new backends for other languages and libraries. We have currently implemented two such code generation modules for CATAPULTS: a GNU Pth backend (for PC's) and a Dynamic C backend (for embedded systems).

6.1 The Frontend

The CATAPULTS translator is written in Python using PLY (Python Lex/Yacc) [18]. The translator uses very simple propagation-based static analysis to track the various invariants described in Section 4.2. 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 4.2, CATAPULTS generates runtime assertions in the generated code for scheduler code that it cannot analyze statically.

6.2 The GNU Pth Backend

GNU Pth [10] 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) [19]. 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.

6.3 The Dynamic C Backend

While our Pth backend 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. Dynamic C includes builtin cooperative multithreading in the form of costatements and cofunctions, but only allows a round-robin scheduling policy for the threads. Although it is possible to use tricks to accomplish dynamic scheduling in Dynamic C [20], 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.

7 Experience

This section describes our experience, including performance results, in using CATAPULTS to write custom schedulers for two main applications: a web server on Linux and a weather station application on an embedded system. We have also used CATAPULTS to write custom schedulers for a variety of other applications, e.g., thread schedulers that use various prioritization schemes. We are currently developing CATAPULTS schedulers for MySQL [2] and a web server on an embedded system.

7.1 The CoW Web Server on Linux

To measure the performance benefits of using CATAPULTS on a real application, we adapted CoW, a cooperatively multithreaded web server that we had previously written, to use a CATAPULTS scheduler. CoW was developed with the GNU Pth threading library described in Section 6.2. When initially developed, CoW relied on the generic first-come, first-serve 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 [21]), we realized that better performance could be obtained if the scheduler was tailored to the specific requirements of CoW. We identified two cases in which more intelligent scheduling decisions could result in superior performance:

Table 1: Pages served by CoW running without and with a CATAPULTS application-specific scheduler

	Generic Scheduler	CATAPULTS scheduler	Improvement
5 handlers, 90K files	8658	10101	16.67%
20 handlers, 90K files	9807	10983	11.99%
100 handlers, 90K files	10917	10653	-2.48%
5 handlers, 250K files	5604	5760	2.78%
20 handlers, 250K files	5478	5109	-7.22%
100 handlers, 250K files	5184	5406	4.28%
5 handlers, 4 byte files	89883	86343	-3.94%
20 handlers, 4 byte files	202938	295425	45.57%
100 handlers, 4 byte files	463093	464457	0.3%

- If CoW is processing a large number (1024 on Linux systems) of HTTP requests simultaneously, it will exhaust its supply of file descriptors and be unable to accept any more network connections. In this case, context switching to the thread that accepts new connections is undesirable since no actual work can be performed by that thread; the useless context switch and networking system calls performed add a great deal of unnecessary overhead.

- The more quickly HTTP requests are processed and flushed from the system, the faster new requests can be accepted. By giving greater priority to request handler threads that are near completion, the overall throughput of the system can be significantly increased.

We developed a custom scheduler for CoW that addressed these issues (scheduler specification can be downloaded from Reference [22]). Only about 20-30 lines of code needed to be added/modified in CoW in order to get it to run with CATAPULTS; these modifications consisted of registering application variables with the scheduler.

We benchmarked the performance benefit of using the CATAPULTS scheduler by running the httperf [23] HTTP benchmark against against CoW running both with and without the CATAPULTS scheduler several times. We varied both the number of handler threads used by the server and the size of the files being requested. Our results are summarized in Tables 1 and 2 (the numbers given are averages of several runs with low variances).

For the most part, use of our application-specific scheduler resulted in more successful responses and quicker response times, although there were a few tests where performance decreased.

7.2 Weather Monitoring Station Application

In order to measure the benefit of using CATAPULTS on an embedded application, we simulated the weather monitoring station example described in Section 3. Since we do not have access to

Table 2: Average response time (in ms) of CoW running without and with a CATAPULTS application-specific scheduler

	Generic Scheduler	CATAPULTS scheduler	Improvement
5 handlers, 90K files	6014	4691	22.0%
20 handlers, 90K files	4570	1527	66.6%
100 handlers, 90K files	1893	1963	-3.7%
5 handlers, 250K files	10687	9708	9.2%
20 handlers, 250K files	7354	5698	22.5%
100 handlers, 250K files	4865	4248	12.7%
5 handlers, 4 byte files	732	794	-8.5%
20 handlers, 4 byte files	589	455	22.8%
100 handlers, 4 byte files	333	346	-3.9%

Table 3: Comparison of CATAPULTS threading library and generic cmthread.lib

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

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 Section 3 was used to control thread scheduling. The complete specification (including the minor details omitted in Figures 1-8) 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 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 3, 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).

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 execu-

tions 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 3, the simulation completed almost 10 seconds faster when using the CATAPULTS-generated scheduler (a 12.6% speedup).

7.3 Embedded CoW Web Server

We also adapted CoW [17], a cooperatively multithreaded web server, to use a CATAPULTS scheduler. The version of CoW that runs on the Rabbits uses the standard Dynamic C first-come, first-serve scheduler. We identified that we could obtain better performance if the scheduler was tailored to the specific requirements of CoW. In particular, the more quickly HTTP requests are processed and flushed from the system, the faster new requests can be accepted. By giving greater priority to request handler threads that are near completion, the overall throughput of the system can be significantly increased.

Figures 9 and 10 give the complete CATAPULTS scheduler specification for CoW. To give priority to handler threads that are near completion, the scheduler imports the variable `bytesleft` from each thread. The scheduler represents the ready list as a pqueue (see Section 5.1) sorted by `bytesleft_internal`. This scheduler variable, `bytesleft_internal`, mirrors the application variable `bytesleft`; it is updated each time a thread is context switched out by the `switch_out` event. This extra level of indirection is necessary because pqueues and pstacks can only be sorted on thread attributes, not on imported application variables.

We are currently getting CoW to work again on the Rabbits. The problem appears to be with the Dynamic C networking code, which has changed since the previous version with which CoW worked. This problem is not related to CATAPULTS since we have the same problem with non-CATAPULTS code. We expect to have fixed this problem and report for CoW the same kind of data as in Table 3 in the next version of this paper.

7.4 The MySQL Database

We are also in the process of developing a CATAPULTS scheduler for the MySQL database server. 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. We see two possible sources of speedup in implementing

```
scheduler cow {
  thread {
    int state;
    int bytesleft_internal;
  }

  threadimports {
    int bytesleft default 0;
  }

  data {
    threadref current;    // current thread
    threadref next;       // next thread (named yield)
                          // ready threads
    pqueue RQ sortable on bytesleft_internal;
    queue SQ;             // suspended
    int READY = 1, SUSPENDED = 2;
  }

  event init { /* noop */ }

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

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

Figure 9: CATAPULTS scheduler for the CoW web server, part 1 of 2

a MySQL-specific scheduler. First, giving priority to threads that have exclusive (write) locks on database tables will improve performance by quickly unblocking multiple threads that need shared (read) locks on those tables. Second, the threads that MySQL uses to implement advanced database features only need to be run when certain database events occur (e.g., a database replication thread need only be run after an update to the database). We are just now beginning work on our specification of the MySQL scheduler, and hope to have performance results for the next iteration of this paper.

```
  event switch_out(t) {
    t.bytesleft_internal = t.bytesleft;
    t => RQ;
  }

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

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

  query threads_ready     { return |RQ|; }
  query threads_suspended { return |SQ|; }
  query threads_total     { return |RQ| + |SQ| + 1; }
  query is_ready(tid) { return tid.state == READY; }
  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;
    tid => RQ;
  }
}
```

Figure 10: CATAPULTS scheduler for the CoW web server, part 2 of 2

8 Summary of Current Work

Our experience developing a customized scheduler for CoW shows CATAPULTS' applicability to Internet servers running on uniprocessor machines. We expect most types of servers (web, mail, database, etc.) to have similar scheduling requirements and obtain a similar performance benefit from application-specific scheduling. Although the scheduling requirements for embedded applications are much less consistent, we believe that our embedded test of CATAPULTS to be representative of a typical embedded system, so our very positive results are encouraging. 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.

9 Future Work

CATAPULTS' success with traditional, centralized schedulers has inspired us to explore the extension of CATAPULTS to other scheduling domains, such as realtime applications and distributed applications. The modifications necessary to facilitate these extensions have also led us to explore the use of a metaprogramming model for generating CATAPULTS grammars specialized for a given threading package (GNU Pth, cmthread.lib, etc.) and application class (classic multithreaded, realtime, distributed, etc.). Section 9.1 discusses our plans for using CATAPULTS with distributed systems, Section 9.2 discusses the obstacles that will need to be overcome to make CATAPULTS an effective tool for realtime scheduler development, and Section 9.3 describes the requirements of a CATAPULTS metaprogramming model.

9.1 Distributed Scheduling Enhancements

CATAPULTS scheduling can be extended to distributed computing. Application-specific schedulers are most likely to be useful on distributed systems with heterogeneous nodes, for reasons explained below. Developing application-specific schedulers for distributed programs provides several advantages.

- For applications that do not explicitly manage the distribution of their threads, threads can be assigned to computing nodes that most closely match their needs. For example, threads that are expected to do lots of I/O can be placed on nodes with fast disk access, threads that need to perform large, memory-intensive calculations can be assigned to nodes with lots of RAM, etc. Likewise, dissimilar threads can be grouped together so that they can more efficiently overlap I/O with computation (so that, e.g., two I/O bound threads will not wind up competing for disk bandwidth on the same machine).

- Different nodes can easily utilize different local scheduling policies and can still use global scheduling information (i.e., information about the state of the system as a whole) to guide their local decisions.

- If responsibility for thread distribution is moved from the application to the scheduler, it becomes easier to migrate the entire application to a new set of hardware. In this case, only the scheduler needs to be modified rather than the actual application code.

There is a lot more variation in the design and style of distributed applications than there is in traditional threading libraries. Distributed applications vary widely in areas such as explicit versus

implicit distribution, communication models (shared memory versus message passing), etc. Because these differences are so significant and have such a large impact on how the application is written, we will refer to the libraries and software used for distribution as distributed threading "frameworks" rather than just "threading libraries."

9.1.1 Design

CATAPULTS, as described in Section 5, currently assumes a single scheduler on a single node. In CATAPULTS' view of a distributed system, a scheduler specification will consist of several cooperating local schedulers with some centralized logic to glue them together.

Node Families Since different nodes are expected to have different resources and the exact number of nodes may or may not be known before runtime, CATAPULTS will be extended with the concept of node "families." A node family will be specified much as a regular CATAPULTS scheduler is specified — its data section lists variables that will be used locally by the scheduler on the node, but that are not available to schedulers on other nodes and its set of event handlers will be executed on nodes of that family. A node family specification will require two additional sections not present in regular CATAPULTS schedulers. The first section lists all globally distributed scheduler variables that the node scheduler uses. These variables are visible to the schedulers on all nodes and to the distribution specification (described below) via a Distributed Shared Memory (DSM) model. The second new section is a list of all properties of a specific node that should be exposed to code in the distribution specification (e.g., amount of free RAM at a node).

Each node family will be defined in a separate file; not only does this organization break a large distributed scheduler into smaller, more manageable chunks, it also facilitates the reuse of certain node families between applications that have different overarching distributed schedulers.

Distribution Specification As described above, a separate file is used to define each node family of a distributed CATAPULTS scheduler. An additional file, the distribution specification, specifies behavior of the scheduler as a whole. This file will contain the following:

- Declarations of node collections (described below).
- Any globally-available variables that should be made available to all node schedulers.
- Default implementations of event or query handlers for operations that are implemented in the same way across several node families.

- A "node arrival" block that contains logic for organizing the sets of nodes currently present in the system. In some distributed applications, this code may only be called a single time at the beginning of the program (if nodes are static) and in other applications it may be called at various points during execution of the application (if nodes come and go dynamically).
- A "node departure" block that is executed when nodes leave the system during execution.
- A "dispatch" block that specifies on which node in the system a new thread should be created. Some frameworks for distributed applications manage this explicitly at the application level; this section will not be required in schedulers for such frameworks. On the other hand, some distributed applications will not explicitly manage their own distribution (e.g., Pthreads applications automatically distributed via the Rthreads framework, as described in Section 9.1.4), leaving this to be handled by the scheduler.

Node Collections Distributed applications may or may not have a static set of machines onto which they can distribute their threads. Even for applications that do assume a static set of machines, the optimal distribution of threads to those machines may depend on application-specific information that is only available at runtime (e.g., how many threads are already running on a machine, how much disk space is left at a node, etc.). This requires that the distribution specification be able to collect and organize collections of available nodes, similar to how a scheduler manages collections of threads available for scheduling. To facilitate this kind of organization, CATAPULTS will make some of the thread collection types (stacks, queues, pstacks, and pqueues) available for organizing processing nodes. A "node" type will be provided for maintaining references to individual nodes (similar to the way "threadref" works) and node objects can also be organized into arrays.

Further Extensions Further distribution-oriented extensions to CATAPULTS can be implemented as additional event handlers in the CATAPULTS scheduler. For example, load balancing is often a concern in distributed systems and may or may not be handled explicitly at the application level. One possible way to accomplish implicit load-balancing would be to introduce work stealing event handlers. A scheduler could maintain two separate ready queues – one of regular, fixed threads and one of "stealable" threads. A new "steal" event could be added to allow the scheduler to select a thread to migrate to another idle node and the "schedule" event handler could be extended to steal from other nodes if no useful work is available on the current node.

9.1.2 Example

This section provides an example of what a CATAPULTS specification for a distributed scheduler might look like. As mentioned in Section 9.1.1, a complete distributed scheduler consists of several node families, each of which is defined in a separate file. The general scheduler definition must start with a section that declares these families and indicates where their specification files are located, as shown in Figure 11. After node families are specified, node collections are declared as shown

```
scheduler distrib {
    /* Node families */
    nodefamilies {
        "../nodetypes/fastdisk.family" : fastdisk;
        "../nodetypes/lowram.family" : lowram;
        "../nodetypes/embedded.family" : slowcpu;
    }
```

Figure 11: Node family declarations

in Figure 12. These collections, which function the same way as collections of threads in a regular scheduler, provide a way for the distributed scheduler to organize and manage the processing nodes to which it can dispatch threads. Next, the scheduler must define any globally distributed scheduling

```
    /* Node collections */
    nodes {
        pqueue numcrunch sortable on cpuspeed;
        pqueue decompress sortable on freeram;
        stack busynodes;
        queue sensors;
        node database;      // only a single DB node
        node display[3];    // exactly 3 display nodes
    }
```

Figure 12: Node collections

variables that should be available to the schedulers on all nodes. This section, as illustrated in Figure 13, is similar to the `data` section in a traditional CATAPULTS scheduler. As described

```
    /* Distributed global data */
    global {
        int lowram_threads = 0;
        int total_ram = 0;
    }
```

Figure 13: Distributed global scheduling variables

earlier, a CATAPULTS scheduler may require a "dispatch" block, which describes how new threads are allocated among the available processing nodes. Some threading libraries may require that the

application specify this explicitly at thread creation time, but schedulers for libraries that do not will require logic to dispatch new threads. The dispatch block, shown in Figure 14, has access to all globally distributed variables, but cannot access the local scheduling variables on any node. After the dispatch block is defined, "node arrival" and "node departure" blocks may be defined to

```
/*
 * Dispatch is like an event, but doesn't operate in any specific node.
 * It can only access distributed globals, not any node-local data.
 */
dispatch {
    node tmp;

    if ( |decompress| > 0 )
        decompress => tmp;
    else if ( |numcrunch| > 0 )
        numcrunch => tmp;
    }
    select tmp;
    tmp => busynodes;
}
```

Figure 14: Thread dispatch block

specify how new processing nodes should be managed. As with the dispatch block, these blocks are unnecessary for schedulers targeted at frameworks that handle thread placement at the application level. Examples are shown in Figures 15 and 16. Finally, the main distributed scheduler specification

```
arrival(n) {
    if (n is fastdisk)
        n => database;
    else if (n is lowram && |lowram| < MAXDECOMPRESS)
        n => decompress;
    ...other cases...
    else
        n => sensors;
}
```

Figure 15: Thread arrival block

concludes with default implementations for event or query handlers.

9.1.3 Implementation Optimization

The usage of globally distributed scheduling variables is likely to vary significantly between applications. Some applications will use globally distributed variables heavily and relatively uniformly among nodes. Other applications will use certain variables primarily at a single node and not at all at other nodes. Due to these different usage patterns, no one method of distributing the values is ap-

```
arrival(n) {
    if (n is fastdisk)
        n => database;
    else if (n is lowram && |lowram| < MAXDECOMPRESS) {
        total_ram += n.freeram;
        n => decompress;
    }
    ...other cases...
    else
        n => sensors;
}
```

Figure 16: Thread departure block

propriate for all types of applications. To provide the best performance possible, the CATAPULTS compiler should be able to analyze where and how often global variables are used by a scheduler's node families and generate appropriate code for distributing the value. For example, if all nodes need to use a scheduling variable frequently, but only a single node updates the variable's value, a broadcast method may be the best way to distribute changes to the value. On the other hand, if the value is used primarily by a single node, the value can be stored at that node and other nodes will issue requests if they occasionally need the value.

9.1.4 Evaluation

In order to evaluate the benefit of using CATAPULTS to generate distributed schedulers, we will need to update existing distributed threading frameworks to make use of CATAPULTS-generated schedulers and measure the performance impact that application-specific schedulers have on programs running on these frameworks. "Distributed threading frameworks" is a very broad target, so multiple frameworks and applications will be examined.

Kaffe JVM Kaffe [24] is a clean room implementation of the Java virtual machine. Kaffe would be a good framework to update for use with CATAPULTS because by default it uses its own internal threading library (although native Pthreads are also supported under Linux) and it also includes an RMI implementation for developing distributed applications. Since Kaffe is highly portable and the threading routines are already modularized [25], extending it to use CATAPULTS should be straightforward (ports already exist for several embedded systems and RTOS's). A large number of distributed applications already exist that use Java RMI, so it should be easy to find some good applications to use as benchmarks.

DesCaRTeS Runtime DesCaRTeS [26] is a platform for developing distributed applications on a network of embedded processors. DesCaRTeS is currently written using the multithreading features of the Dynamic C programming language [16], but we know that it should be relatively straightforward to port this `cmthread.lib`, a cooperative multithreading library we developed for the Rabbit processors and that already has a traditional CATAPULTS backend (described in Section 6.3). Although no real-world applications exist for DesCaRTeS yet, we plan to develop a synthetic benchmark that will simulate the processing and resource requirements of a real embedded control system.

Rthreads Rthreads [27] is another distributed threading framework that could be extended for use with CATAPULTS. Rthreads provides a distributed threading model that is closely related to Pthreads, both syntactically and semantically. Global variables are transparently shared between processing nodes via a DSM approach. A preprocessor is available to transform traditional Pthreads programs into distributed Rthreads programs. Rthreads is an excellent target for CATAPULTS because, unlike Kaffe and DesCaRTeS, distribution does not have to be explicitly defined at the application layer and can be handled at the scheduler level. Since Rthreads can translate and run traditional Pthreads programs, it should be straightforward to find suitable benchmark applications.

9.2 Extending CATAPULTS With Realtime Capabilities

Although embedded systems are already a functioning target of CATAPULTS, we would like to extend CATAPULTS to provide better support for developing soft realtime schedulers. Hard realtime systems generally perform schedulability analysis off-line and preallocate resources to tasks, which makes them less suitable for the CATAPULTS, but soft realtime systems are capable of dynamic scheduling and adaptation, which makes them a more suitable match for CATAPULTS. Realtime embedded control systems are a primary target for CATAPULTS schedulers.

9.2.1 Requirements

A couple of straightforward modifications to the CATAPULTS design will be necessary to facilitate realtime scheduling.

Time Datatype Clearly the concept of time is extremely important to the development of a realtime scheduler. At the moment, CATAPULTS provides no portable way of specifying and manipulating time values (although some systems can use verbatim statements and verbatim expressions to accomplish this). The creation of a time datatype will be necessary for realtime scheduling.

Event Failure In the current CATAPULTS model, event handlers can never fail to execute. That is, operations like "create thread," "schedule," or "suspend" always succeed. While this assumption is generally valid for regular schedulers, it is unsuitable for realtime schedulers. Realtime schedulers need to perform *admission control*, i.e., determine whether or not the system is capable of supporting another thread while still maintaining realtime deadlines. Therefore, CATAPULTS must be modified to allow event handlers to return a failure status if an event cannot be fulfilled. This failure will be returned to the application level when events such as "create thread" are denied.

9.2.2 Obstacles

In addition to the straightforward changes outlined above, several more difficult obstacles will need to be overcome to allow the development of realtime schedulers with CATAPULTS. Overcoming these obstacles, while still allowing flexibility in the types of schedulers that can be created, is our goal with realtime CATAPULTS.

Scheduler Execution Time Successful realtime scheduling depends on the accurate prediction and bounding of task execution times. Since the scheduler is itself a task that must compete for processor time, it is crucial that the execution time of the scheduler itself be bounded and predictable. The CATAPULTS translator will be able to analyze scheduling code and determine the worst-cost execution time (WCET) of various events (especially the "schedule" event). Accomplishing this will likely disallow the use of some of CATAPULTS' more general programming features, such as unbounded loops or verbatim statements/expressions. This is an inconvenience, but a necessary tradeoff to obtain meaningful realtime guarantees.

Application Analysis In order to perform meaningful realtime guarantees, a CATAPULTS scheduler will need to have relatively accurate estimates of the WCET of the various tasks that are to be scheduled. Estimating execution times will require a means of analyzing the application source code (or possibly the generated object code) to determine the length of each task. Since task scheduling is so important to realtime applications, we feel that a tighter coupling of application and scheduler is acceptable in a realtime system than in a regular application, so an application code preprocessor or source code annotations may be suitable requirements for application development.

9.2.3 Evaluation

As we mentioned above, realtime embedded control systems are one of our primary targets with this research. Although it is difficult to accumulate the hardware requirements of a real embedded control system, it is easy to develop synthetic control systems that simulate the behavior of real control hardware. We intend to develop simulations of real-world realtime control systems and we will measure our success by attempting to develop more effective schedulers for these systems with CATAPULTS. In terms of realtime scheduling, we will consider a scheduler effective if it allows more tasks to be run than conservative, generic realtime schedulers, while still maintaining realtime guarantees.

9.3 Grammar/Parser Generation Through Metaprogramming

Both of the CATAPULTS extensions in the previous sections (distributed and realtime schedulers) will require modifications to the CATAPULTS grammar. Some of these modifications are additions to the grammar (e.g., the introduction of a time datatype, or the addition of thread dispatch blocks in a distributed scheduler), while others are removals (e.g., unbounded loops should not be used in a realtime scheduler). Although we have tried to keep the CATAPULTS language generic and portable, our design decisions have likely been influenced by the design and requirements of the specific threading frameworks that we are currently working with (GNU Pth, Dynamic C, etc.). It is likely that other frameworks and scheduling domains will introduce concepts or requirements that we had not foreseen when we first developed the CATAPULTS language. Rather than develop a separate version of CATAPULTS with a different grammar for each of these situations (e.g., CATAPULTS-generic, CATAPULTS-rt, CATAPULTS-distrib, etc.), we will explore the possibility of using metaprogramming to generate instances of the CATAPULTS grammar that are specialized for the target framework. When CATAPULTS is first ported to a new threading framework, the porter should be able to write a specification for the new framework (e.g., "yes, a time datatype is required; no, event failures are not required." etc.) and have the grammar and translator frontend automatically generated. Although this approach somewhat decreases the portability of CATAPULTS schedulers, it is unlikely that the same scheduler would make sense for applications running on threading frameworks with such different structures. For example, our existing schedulers for the Pth threading library would not make sense in a distributed framework since they include no concept of separate processing nodes or distribution. The frontend and backend of the translator would still be separate since many frameworks might be able to make use of the same frontend (as

Pth and Dynamic C already do).

10 Timeline

This timeline is only tentative as it is difficult to determine the amount of time necessary for each phase of the project. The numbers below are estimates of the amount of time required to perform background research and to implement, test, and benchmark each feature. They do not include the time required to write conference or journal papers or to maintain the current implementation.

Task	Start Month	End Month
Distributed Frontend	1	1
Distributed DesCaRTeS Backend	1	4
Distributed Kaffe Backend	5	9
Distributed Rthreads Backend	10	13
Realtime Scheduling	14	21
CATAPULTS Metaprogramming	22	25

References

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

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

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

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

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

[6] 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.

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

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

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

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

[11] 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)*, 2002.

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

[13] 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*, September 2004.

[14] 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.

[15] David Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, University of Washington Department of Computer Science and Engineering, May 1993.

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

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

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

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

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

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

[22] Matthew D. Roper. CATAPULTS scheduler code for CoW webserver. http://www.cs.ucdavis.edu/~roper/catapults/examples/cow.sched.

[23] 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`.

[24] Kaffe.org. `http://www.kaffe.org`.

[25] Michael Barr and Jason Steinhorn. Kaffe, anyone? implementing a Java virtual machine. *Embedded Systems Programming - Embedded.com*, February 1998.

[26] Justin T. Maris, Matthew D. Roper, and Ronald A. Olsson. DesCaRTes: a run-time system with SR-like functionality for programming a network of embedded systems. *Computer Languages, Systems and Structures*, 29(4):75–100, December 2003.

[27] B. Dreier, M. Zahn, and T. Ungerer. The Rthreads distributed shared memory system. In *Third International Conference on Massively Parallel Computing Systems MPCS'98*, 1998.