Developing Embedded Multi-threaded Applications with CATAPULTS, a Domain-specific Language for Generating Thread Schedulers *

Matthew D. Roper
Department of Computer Science
University of California, Davis
Davis, CA 95616-8562 USA
roper@cs.ucdavis.edu

Ronald A. Olsson
Department of Computer Science
University of California, Davis
Davis, CA 95616-8562 USA
olsson@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 thread schedulers provides three advantages. First, it modularizes the thread scheduler, making it easy to plug in and experiment with different schedulers. Second, using a domain-specific language for scheduling code helps prevent several of the common programming mistakes that are easy to make when programming in low-level C or assembly. Finally, the CATAPULTS translator has multiple backends that generate code for different languages and libraries. This makes it easy to prototype an embedded application on a regular PC, and then develop the final version on the embedded hardware; the CATAPULTS translator will take care of generating the appropriate code for both the PC prototype and the final embedded version of the program. Using our implementation of CATAPULTS for Z-World's embedded Rabbit processors, we obtained a performance gain of about 12.6% at the expense of about 12.7% increase in code size for a fairly typical embedded application.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.11 [**Software Engineering**]: Software Architectures; D.4.1 [**Operating Systems**]: Process Management

General Terms

Languages, Performance, Design, Experimentation

Keywords

Thread scheduling, domain-specific languages, user-level threads, application-specific schedulers

*This work is partially supported by Z-World, Inc. and the University of California under the MICRO program. The National Science Foundation partially supported our equipment through grant EIA-0224469.

CASES'05, September 24–27, 2005, San Francisco, California, USA.

1. INTRODUCTION

Embedded control systems are generally responsible for handling several concurrent tasks (e.g., driving different pieces of hardware) and thus lend themselves to a multi-threaded design. 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 regular computer, which has much more processing power.

This paper introduces CATAPULTS, a system for developing application-specific schedulers, and shows how to use CATAPULTS for embedded systems applications. Generating specialized schedulers for a specific application improves 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 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," "quantum expired," "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 low-level C

or assembly language (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; this is especially useful when simulating the system on a regular PC before developing the actual embedded version.

Our primary targets for CATAPULTS are small, resource-constrained embedded controllers with low processing power. Such systems generally run the control software directly on the hardware, without the support of a real time operating system. Although we are currently working on extensions to CATAPULTS to aid in the development and verification of soft realtime schedulers, these are not a primary focus of our current implementation.

The rest of this paper is organized as follows. Section 2 provides an introductory example of a CATAPULTS scheduler for a representative embedded application. Section 3 gives an overview of related work. Section 4 describes the purpose and design of the CATAPULTS system. Section 5 provides details on the organization of the CATAPULTS domain-specific language. Section 6 describes how the components of the system were implemented. Section 7 discusses our experience, including performance results, of using CATAPULTS on representative embedded applications. Finally, Section 8 provides some discussion, describes possible avenues for future exploration, and concludes the paper.

2. AN INTRODUCTORY EXAMPLE

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 [12], 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 6.2). 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 won't 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.).

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

Figure 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 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

```
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

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: 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 6.2). 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).

```
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

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 tar-

```
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

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

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 [2], 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;

```
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

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 [7] 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 applications of domain-specific languages for embedded systems include Hume [5]. 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.

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 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.

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

Although threading libraries on embedded systems are generally far simpler than their PC counterparts, modifying or replacing the scheduling algorithm used by a thread library is still a non-trivial task. Although it may be easy to locate the function that picks the next thread, most threading libraries have other scheduling code spread throughout the library (e.g., code that manipulates queues of threads when various events occur). Tracking down every such reference to the scheduler's data structures is tedious and error-prone. For example, although the GNU Pth threading library [4] 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. 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.

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 ap-

plication. Thus, it is very easy to try out different scheduling strategies.

4.2 Error Prevention

Threading libraries for embedded systems are often written in C or low-level assembly since these are the languages that are most often used for application development. Although C and assembly are very powerful languages, 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 [12]) and GNU Pth [4] (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 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.

Although both of the backends that we have developed so far have used a cooperative approach to task switching, CATAPULTS is also applicable in preemptive environments. Schedulers for a preemptive threading library would take the same form as those for cooperative libraries except that they would 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.). Porting an existing cooperative scheduler to a preemptive library would simply require the addition of these few additional handlers.

5. LANGUAGE DETAILS

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.

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 2). Any changes that the application makes to the variable will immediately be visible through the linked scheduler variable and vice versa.

As seen in Section 2, 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.

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.

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.

5.3 Verbatim Statements and Expressions

CATAPULTS provides verbatim statements and 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. For example, a scheduler that uses a random number generator to make some of its scheduling decisions will use verbatim statements to generate random numbers because CATAPULTS has no instructions to do so. 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 expressions will need to be rewritten for each target language/library to which the scheduler is compiled).

6. IMPLEMENTATION

This section describes two key parts of the CATAPULTS implementation: its frontend and one of its 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 Dynamic C backend (for embedded systems) and a GNU Pth backend (for PC's). Below, we focus on the Dynamic C backend; Reference [11] discusses the Pth backend.

6.1 The Frontend

The CATAPULTS translator is written in Python using PLY (Python Lex/Yacc) [3]. 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 Dynamic C Backend

To apply CATAPULTS in an embedded environment, we developed 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 [10], 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. Our Dynamic C backend generates a custom version of the cmthread.lib library that contains the custom generated scheduling code inline. 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.

(In contrast, our Pth implementation dynamically loads schedulers at runtime [11], which is neither possible nor advantageous in the Dynamic C environment; it therefore uses indirect function calls via function pointers, which incurs some additional overhead.)

7. EXPERIENCE

Below, we describe some of our experiences using CATAPULTS for embedded systems applications. We have also used CATAPULTS with the Pth backend for regular PC applications such as the CoW web server and numerous small schedulers [11].

7.1 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 2. 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 Section 2 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 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).

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

7.2 CoW Web Server

We also adapted CoW [6], 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. Although CoW, provides satisfactory performance with this generic scheduler, we realized that performance could be improved by using an application-specific scheduler.

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:

- 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.

Table 1: 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

Table 2: Comparison of CoW web server throughput (with 4 handler threads) under generic and CATAPULTS schedulers.

# of clients	generic scheduler (bytes/sec)	CATAPULTS scheduler (bytes/sec)	Increase in Throughput (ratio)
1	4245.23	5958.96	0.40
2	7359.28	12028.24	0.63
4	16181.50	18571.56	0.15
6	20567.03	21488.52	0.04
8	23407.21	24906.37	0.06
10	7451.62	26542.34	2.56
12	6477.88	16513.81	1.55

Table 3: Comparison of CoW web server throughput (with 8 handler threads) under generic and CATAPULTS schedulers.

# of clients	generic scheduler (bytes/sec)	CATAPULTS scheduler (bytes/sec)	Increase in Throughput (ratio)
1	4460.36	5390.08	0.21
2	8738.18	9621.48	0.10
4	15653.21	17587.51	0.12
6	21623.12	21712.17	0.00
8	26289.43	23062.81	-0.12
10	33543.06	26981.74	-0.20
12	38085.71	30544.68	-0.20

- 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 Reference [9].

We benchmarked CoW with both the generic scheduler and the specialized CATAPULTS scheduler using a varying number of web clients. Tables 2 and 3 presents 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.

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.

8. CONCLUSION

Although real world embedded systems are likely to have a very diverse set of scheduling requirements, we have applied CATAPULTS to what we believe to be a typical embedded system and have achieved very positive results. Performance gains are likely to vary widely 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 sched-

ulers with CATAPULTS is relatively straightforward and can provide significant performance gains. Moreover, using CATAPULTS provides more safety and portability.

We have a number of plans for future exploration. We realize that many schedulers share common code and data structure organization, so we would like to make it possible to derive new schedulers from existing ones and then override specific parts in an object-oriented manner. Hierarchical thread scheduling (e.g., as discussed in Reference [1]) is another area we would like to explore; CATAPULTS could be extended to develop different scheduling algorithms for different subgroups of threads. Such hierarchical schemes would allow different people to develop different schedulers for subgroups of threads in an application and then schedule those subgroups according to a higher-level scheduler. It would also be interesting to explore the possibility of creating schedulers graphically with a GUI design tool; this would be especially attractive to engineers of a mechanical system who do not have a lot of experience with software development. Finally, CATAPULTS could be extended to work in a distributed environment with systems like DesCaRTes [8] that distribute threads over a network of embedded, uniprocessor controllers.

Acknowledgments

Jason Cheung, especially, and Glen Sanford helped with the development and testing of CATAPULTS. The referees provided very helpful comments.

9. REFERENCES

[1] D. Auslander, J. Ridgely, and J. Ringgenberg. *Control Software for Mechanical Systems: Object-Oriented Design in a Real-Time World*. Prentice Hall PTR, 2002.

[2] 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.

[3] D. Beazley. PLY (Python Lex-Yacc). `http://systems.cs.uchicago.edu/ply/`.

[4] R. S. Engelschall. GNU Pth - the GNU Portable Threads. `http://www.gnu.org/software/pth/`.

[5] K. Hammond and G. Michaelson. Hume: a domain-specific language for real-time embedded systems. In *Proceedings of the Second International Conference on Generative Programming and Component Engineering*, 2003.

[6] T. Ishihara and M. D. Roper. CoW: A cooperative multithreading web server. `http://www.cs.ucdavis.edu/~roper/cow/`.

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

[8] J. T. Maris, M. D. Roper, and R. 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, Dec. 2003.

[9] M. D. Roper. CATAPULTS scheduler code for embedded CoW webserver. `http://www.cs.ucdavis.edu/~roper/catapults/examples/cmthreadcow.sched`.

[10] M. D. Roper. Dynamic threading and scheduling with Dynamic C. `http://www.cs.ucdavis.edu/~roper/dcdynthread/`.

[11] M. D. Roper and R. A. Olsson. CATAPULTS: Creating and testing application-specific user level thread schedulers. in preparation.

[12] Z-World Inc. Dynamic C user's manual. `http://www.zworld.com/documentation/docs/manuals/DC/DCUserManual/index.htm`.