The Plasma Language

Brian Kahne
bkahne@freescale.com

Peter Wilson
Peter.Wilson@freescale.com

Plasma, (**P**arallel **LA**nguage for **S**ystem **M**odelling and **A**nalysis) is a parallel language for system modeling and multithreaded application development implemented as a superset of C++. The language extensions are based upon those found in Occam, which is based upon CSP (Communicating Sequential Processes) by C. A. R. Hoare. The basic idea is that threads are explicitly created by the user and they communicate using typed channels. Plasma adds several additional concepts such as shared data structures protected by mutual-exclusion code and the ability to easily wait on the result of a function launched as a separate thread. Although C++ is a difficult language from which to extract formal properties, we believe that the combination of CSP (which does have well-understood semantics) and an identified subset of Plasma will make this possible.
A PDF version of this document may be found at:
http://www.somerset.sps.mot.com/projects/tools/plasma/index.pdf
The latest distribution tarball of Plasma may be found at:
http://www.somerset.sps.mot.com/projects/tools/plasma/plasma-0.8.13.tar.gz

0.1 Purpose of This Work

The goal of the Plasma project is to investigate whether a language with the appropriate constructs might be used to ease the task of system modeling by providing a simple means of representing systems models, increase productivity through clearer representation, and increase productivity and quality through increased compile-time checking of the more difficult-to-get-right aspects of systems models (the concurrency). In addition to all of this, a language with built-in parallelism would mean that it could be used for actual application development and not just modeling.
The ability to enhance single-thread performance, such as by increasing clock frequency, is rapidly diminishing: power is becoming a dominating factor and limiting scalability. Adding additional cores is a scalable way to increase performance, but it requires that customers of our parts have a method for developing multi-threaded applications. A Plasma compiler would allow customers to develop such programs and do so in a safer manner by providing various compile-time checks for obvious problems.
A full-blown Plasma system would thus have wide applicability: It could be used by Freescale design teams to model proposed products and perform early architectural and microarchitectural design trade-offs. In addition, customers could use the product to model their own complete systems, performing their own analysis to determine what should run as software on microprocessors and what should be dedicated hardware. The portions of the Plasma code which would be implemented as hardware could then serve either as executable specifications or could be synthesized directly into hardware.
The use of Plasma will make it easier for us to develop products which meet our customers' performance needs, while also meeting their power requirements, by adding extra processor cores rather than developing brand-new cores. For

example, an engine controller consisting of just a single thread might need a new core, at twice the performance of the original, to handle an engine twice the size, thus running into severe power-budget constraints. On the other hand, the same program implemented in Plasma, where each cylinder is implemented as a separate thread, could simply be run on a system consisting of two cores.

0.2 Existing Work

There exist a plethora of languages attempting to solve the problem of how to model complex systems. They generally fall into two main categories: Those originally derived from a hardware description language and those derived from C or C++. The first approach, typified by System Verilog, does allow the user to raise the level of abstraction, but it still remains primarily a language for describing hardware at a relatively low level. For example, it has relatively limited abilities for creating new abstract data structures. It appears unlikely that we could convince customers to develop their software using this language. The other approach, deriving a language from C or C++, is typified by SpecC, Handel C, and SystemC. The first two are languages with true parallelism: Their compiles are able to perform safety checks to guard against common concurrency problems. However, they are primarily aimed at describing hardware, though in a manner familiar to many programmers. This means that they tend to implement a subset of C or provide constructs which are very specific to certain types of hardware modeling. In addition, the fact that they are based upon C means that they are limited in their extensibility, versus the abstraction mechanisms offered by C++.

SystemC is not a true language in and of itself, but is a class library built on top of C++. Since it implements parallelism, which is not provided by C++, it can in some ways be thought of as its own language, in so far as a traditional library is usually thought of as building upon existing facilities of a language. Since SystemC builds upon C++, it does offer powerful abstraction mechanisms and the ability to use and create sophisticated data structures.

Its main problem, however, is that the compiler is unaware of the concurrency, and is thus unable to perform any relevant checks or optimizations. In addition, the way in which concurrency and channel communication is implemented in SystemC requires the use of C++'s object system. While this is not bad in principle, it would be nice to have the ability to write a concurrent program in an imperative, C-like fashion for those more comfortable with that style. In addition, communication between threads, via channels, requires the use of virtual functions. Taken together, these restrictions mean that SystemC is unlikely to be used for actual application development. However, it does show the utility of taking a powerful and extensible general purpose language and adding a modeling framework on top of it.

The Panama modeling environment, based upon SystemC and developed by FIL, has shown that SystemC can be used to do microarchitectural trade-off analysis. The view adopted by Panama, that a system can be modeled as a

collection of objects, each specialized to the system element to be modeled, and each given behavior through a script, is a promising beginning. The idea here is that a skilled modeling technology team creates the base objects, and folk in the product teams can personalize the behavior of the objects simply by writing scripts. This approach has the promise that the task of modeling can be distributed, and that a tools group can provide re-usable technology for use where needed. This approach has many advantages over the approach currently in effect in what was Somerset, that the modeling group provides both the technology and a modeling service. Such a service-based approach is efficient in throughput- modelers are always busy, but not in latency- no more modelers can be provided, and so the time that a project must wait before it can have a model constructed can be lengthy, unpredictable, and uncontrollable.
However, even Panama's approach can likely be improved upon: One area is its choice of specialized objects and a rather limited scripting language. Rather than having many different object types, each of which is scriptable, we could have exactly one object, call it a *Processor*, capable of executing scripts written in a more general-purpose scripting language. Couple this improvement with the understanding that a key new ability we need is to model software at multiple levels of abstraction, and the idea behind Plasma becomes clear.

0.3 An Introduction to Plasma

The novel aspect of this work is that Plasma is suitable for use as both an applications development language and as a modeling language. For application development, both in the embedded and desktop/server realm, it provides a convenient means for exploiting parallelism provided by a multi-core hardware platform. For modeling purposes, the parallelism in the language is used to model the parallelism inherent in the design being modeled. In either case, a production-ready compiler for Plasma will know about this parallelism and be able to create better optimized code than the traditional approach of using a threading library.
For instance, on an explicit thread switch, the compiler need only generate code to save the live registers of the current function, rather than the entire hardware context. In some cases, the matter is not just about optimization, but about actual correctness: An optimizer might think that a variable is dead, when in fact it is still being used by another thread. Useful safety checks might also be made. For example, the compiler can ensure that two different threads do not write to the same variable unless it is protected by mutual-exclusion code.
Another goal of Plasma is to add to C++ features that make it easier to create robust, correct programs. C++ already offers good abstraction mechanisms and strong type-checking. However, it lacks, for instance, garbage collection. The current Plasma implementation adds garbage collection via the Boehm Weiser Garbage Collector, a fully-conservative collector. The interface allows the user to work with explicitly-managed memory or with managed memory, thus allowing for interoperability with existing C++ code.

In a production-worthy version of Plasma we hope to further investigate what else could be added in order to make life easier for the programmer. For example, we might add default initialization of built-in types so that there is no chance of dangling pointers. We might also consider the recommendations found within Safe, Efficient Garbage Collection for C++.

0.3.1 Parallelism

Plasma inherits its parallelism constructs from Occam and extends them with a functional operator, similar to the *future* found in Multilisp. For example, to create two threads one uses the `par` block:

```
par {
  a();
  b();
}
c();
```

The functions `a()` and `b()` will run in parallel but will both finish before function `c()` is called. Each statement within a `par` block is launched as a separate thread and braces may be used to group multiple statements together.

A replicated form of `par` exists in the form of the `pfor` block:

```
pfor (int i = 0; i != Max; ++i) {
  foo(i);
}
```

In the above example, the body of the `pfor` block, the call to function `foo()`, is launched as a new thread `Max` times.

To work with a more functional approach, the `spawn` operator may be used:

```
Foo x;

Result<int> res = spawn(x.calculate(1,2,3));

cout << "The result is:  " << res.value() << "\n";
```

In the example above, the `spawn` operator runs `Foo::calculate` in a separate thread. Synchronization occurs when `Result::value()` is called, which returns the result of the thread.

0.3.2 Communication

Channels

Communication between threads is handled by typed channels. A channel can be any object which satisfies a certain API **??** and is written in a thread-safe manner. This means that a channel can be extremely efficient: It does not require the use of virtual functions and a production-ready version of the compiler may be able to use hardware-optimized intrinsics, if available. The standard library provides several varieties, including a single-item channel, a queued channel, where the queue size can be fixed or infinite, and an interface channel for using the `Result` object returned by a spawn function.

For example, the following code creates two threads, a producer and a consumer:

```
typedef Channel<int> IntChan;
IntChan c;
par {
  {
    // Thread 1.
    c.write(1);
    c.write(2);
    c.write(3);
    c.write(-1);
  }
  {
    // Thread 2.
    int x;
    do {
      x = c.get();
      mprintf ("Result:  %d\n",x);
    } while (x >= 0);
  }
}
```

The first thread writes numbers to a channel, while the second thread reads from this channel and prints them using `mprintf`, a mutex-protected version of `printf`. Flow-control is handled by the channel itself: If there is no data, then a read will block, while if there is already a data item in the channel, a write will block.

Calling `get()` works when only one channel needs to be queried, but the situation often arises when multiple channels exist. A thread needs to block only if no data exists on any of the channels. To support this, Plasma has the `alt` and `afor` constructs:

```
alt {
  c0.port (int v) {
    plasma::mprintf ("Got a value from port c0:  %d\n",v);
  }
  c1.port (int v) {
    plasma::mprintf ("Got a value from port c1:  %d\n",v);
  }
  c2.port (int v) {
    plasma::mprintf ("Got a value from port c2:  %d\n",v);
  }
  c3.port (int v) {
    plasma::mprintf ("Got a value from port c3:  %d\n",v);
  }
}
```

If none of the channels (`c0` through `c3`) have any data, then the thread will suspend until data is available on any one of the channels. At that point, the relevant block of code will be executed. If one or more of the channels already has data upon entry to the `alt`, then exactly one of the ready channels will be read and its code executed; the selection of the channel is non-deterministic.

Just as Plasma provides the `pfor` construct for replicating threads, it also provides `afor` for replicating channel reads:

```
afor (int i = 0; i != (int)channels.size(); ++i) {
  channels[i].port (int v) {
    plasma::mprintf ("From port %d: %d\n",i,v);
  }
}
```

In the above example, the `afor` block will suspend the thread if no channels in the `channels` container are ready.

`Alt` and `afor` blocks may be nested:

```
alt {
  afor (int i = 0; i != (int)ichannels.size(); ++i) {
    ichannels[i].port (int v) {
      plasma::mprintf ("From port %d:  %d\n",i+ibase,v);
    }
  }
  alt {
    xc.port (int v) {
      plasma::mprintf ("From port xc:  %d\n",v);
    }
  }
}
```

In the above example, the thread will suspend unless a channel in `ichannels` or the channel `xc` has data.

As mentioned above, the ordering in which channels will be read for `alt` and `afor` blocks is indeterminate. To guarantee a specific ordering, the `prialt` and `priafor` constructs may be used. These take the same form as `alt` and `afor` except that the channels will be queried in the order specified by the user.

Synchronous Communication

The Plasma standard library contains a special kind of channel, called a clocked channel, for modeling synchronous hardware. This is a standard channel in the sense that it is just a templated class that conforms to the channel API **??**. The difference is in its behavior: Writes may occur at any time but reads are restricted to clock boundaries, where the clock period is specified by a constructor parameter.

The benefit of this approach is that the user's code remains uncluttered by clocking logic. Timing changes may be made by simply changing parameters in the declaration of the various channels. This also retains the basic simplicity of CSP: Threads continue to communicate through channels, but the properties of the channels themselves regulate this data interchange to clock boundaries.

For example, consider the following program:

```
#include <iostream>
#include "plasma.h"

using namespace plasma;
using namespace std;

typedef ClockChan<int> IntChan;

void producer(int val,IntChan &chan)
{
  for (int i = 0; i != 10; ++i) {
    chan.write(val+i);
  }
  chan.write(-1);
  cout << "Producer done." << endl;
}

void consumer(IntChan &c)
{
  bool done = false;
  while (!done) {
    int d = c.get();
    if (d < 0) {
      done = true;
    } else {
      cout << "Got a value at time " << pTime()
           << ":  " << d << endl;
    }
  }
  cout << "Consumer done." << endl;
}

void pSetup(ConfigParms &cp) { cp._busyokay = true; }

int pMain(int argc,const char *argv[])
{
  IntChan chan(5);

  par {
    producer(10,chan);
    consumer(chan);
  }
  cout << "Done." << endl;
  return 0;
}
```

It consists of two threads: a producer and a consumer. The producer writes integers to a channel while the consumer reads them, stopping when a -1 is received. Notice that neither function deals with time explicitly; they simply communicate via a channel. Only the main function knows about time because it instantiates a clocked channel, `IntChan`, and specifies that the period of the clock is 5 time units. The output of this program is:

```
Got a value at time 5:  10
Got a value at time 10:  11
Got a value at time 15:  12
Got a value at time 20:  13
Got a value at time 25:  14
Got a value at time 30:  15
Got a value at time 35:  16
Got a value at time 40:  17
Got a value at time 45:  18
Got a value at time 50:  19
Producer done.
Consumer done.
Done.
```

The standard library currently consists of a single type of clocked channel, called `ClockChan`. It defaults to storing a single item but it can also act as a queue of fixed or arbitrary size via a constructor parameter. The behavior remains essentially the same: Writes never block unless the queue is full, which might never be the case if the queue size is not fixed. Reads, however, are always synchronous.

Shared Data Structures

Shared data structures may be easily implemented by using the `pMutex` class modifier:

```
pMutex class Foo {
  ...
public:
  pNoMutex bool foo() const;
};
```

A mutex class has all of its public member functions wrapped with serialization code (except for the constructors and destructors). This may be disabled by using the `pNoMutex` keyword, as shown for the method `foo`.

0.3.3 Simulation

Plasma contains a discrete-event time model for simulation purposes. This is not part of the language, per se, but is implemented as a set of functions which manipulate the back-end thread package. This means that a Plasma implementation designed for applications might not support the time model, while an implementation designed for modeling would.

A thread may delay itself by calling `pDelay` and may consume time by calling `pBusy`. A delay corresponds to a thread waiting for some event to occur; it is not consuming any resources and other threads may execute during this time. Being busy, on the other hand, means that the thread is working, or consuming the processing resources of what it is running on. These two functions are the only way in which time advances in Plasma: Everything else is considered to occur in zero time.

What this means is that a user may write real software for an embedded application in Plasma. To gain an understanding of the software's resource requirements, it may be annotated with `pDelay` and `pBusy` calls based upon the anticipated behavior of the underlying hardware. For instance, a multiply-accumulate loop might require two separate instructions, so a call to `pBusy` with a value of two would be made. However, this value could be changed in order to understand the effects of adding a specific multiply-accumulate instruction.

Threads may be distributed to multiple "processors" in order to model multiple hardware resources in a simulation. A thread on one processor might be busy, but this does not affect a thread on another processor:

```
Processor proc1,proc2;

par {
  on (proc1) {
    <code>
    pBusy(100); // Does not affect the other thread.
    <code>
  }
  on (proc2) {
    <code>
  }
}
```

Processors have their own private ready queues of threads, by default, but can be instantiated such that several processors share a single queue. This allows for SMP systems to be easily modeled.

Each thread may be given a priority. Higher priority threads (a lower number) execute until completion before lower priority threads. The lowest priority threads are time-sliced. A delayed high-priority thread may interrupt a lower-priority thread which is *busy*.

0.3.4 Power Modeling

Plasma supports power modeling by providing some simple hooks for recording and retrieving energy consumption on a per-processor basis. The model calls `pEnergy(energy_t)` to add to the current processor's consumed-energy value, where `energy_t` is currently defined as a `double`. Calling `pGetEnergy(Processor)` retrieves a processor's consumed energy and clears the value. Thus, to monitor power, a thread might regularly query processors for their energy and then divide this by the sample period.
Just as with evaluating compute requirements, a process of step-wise refinement may be applied: The user might start with a very course-grained analysis using average power values, then proceed to refine that by adding different energy consumption statements for different paths through the code or compute energy by looking at the hamming distance between consecutive data values read from a channel. Additionally, a model might wrap all calls to `pEnergy` so that a scaling value, based on clock-frequency, might be applied.
This concept can be extended to other resources besides just power and computation: The `ProcValue` **??** template allows a user to associate a data value with a processor. The class's methods allow for incrementing a value, retrieving and clearing it, or just retrieving it. The power API is a simple wrapper around this class, using a `double` as a data value for energy.

0.3.5 Modeling Limited Resources

The need to model an object with limited resources frequently arises. For example, a memory controller has a specific, fixed bandwidth. Plasma provides a class, called `Quantity` **??**, to help with this task. It lets the user specify how long an individual request will take, e.g. a read will take 10 time units. Threads then make requests of this object; their underlying processor is placed into a busy state until the request is satisfied. However, this busy state is time-sliced so that other, higher priority threads are allowed to potentially grab some processing time.

0.4 User's Guide

The Plasma user guide is available here **??**. Refer to it for more details about Plasma.

0.5 Examples

Plasma examples are available here **??**.

0.6 Implementation Details

Implementation details and scheduling information can be found here here **??**.

0.7 Additional Documentation

A Twiki page, containing additional background information, can be found here.

0.8 Future Work

A description of proposed future enhancements can be found here **??**.

Chapter 1

Plasma User's Guide

This document describes PLASMA (Parallel LAnguage for System Modeling and Analysis). It is a superset of C++ which adds concurrency constructs, a concept of simulation time for discrete, and various safety features.

1.1 Usage

Plasma is currently implemented as a front-end which converts a PLASMA program into C++, then invokes g++ to compile this into either an executable or into an object file. As such, its options are basically those of g++. General usage is:

```
plasma <options> <input file>
```

An input file has an extension of `.pa`.
For example, to create an executable from a single Plasma file:

```
plasma -g -Wall -O3 foo.pa -o foo
```

To create an object file, use the **-c** option:

```
plasma -g -Wall -O3 -c foo.pa -o foo.o
```

To link, use **ld** or **g++**. The script **plasma-config** exists to supply the required link line. For example:

```
plasma -c -o a.o a.pa
plasma -c -o b.o b.pa
g++ -o prog a.o b.o ‘plasma-config --libs‘
```

Within a *makefile*, use `$(shell plasma-config --libs)` to generate the link line.

1.2 Concurrency

This section discusses the new language features added in order to support concurrency. These fall into two main categories: Thread creation and thread communication.

1.2.1 Thread Creation

Par Block

The **par** block executes each statement in parallel, waiting until all child threads are finished before proceeding. For example:

```
int x,y,z;
par {
  x = sub1();
  y = sub2();
}
z = sub3();
```

In the above example, the statement `x = sub1()` and the statement `y = sub2()` will execute in parallel. Both will complete before the statement `z = sub3()` is executed. To create a more complex in-line operation, simply use a brace-delineated block, e.g.:

```
par {
  {
    <sequence of statements>
  }
  {
    <sequence of statements>
  }
}
```

PFor Loop

The **pfor** loop acts like a for-loop, except that for each loop iteration, its body is launched as a separate thread. All threads must then complete before execution continues past the loop. For example:

```
const int Max = 10;
int results[Max];
pfor (int i = 0; i != Max ++i)  {
  results[i] = sub(i);
}
```

In this example, ten threads will be launched. Each will return a result which is stored into an element of the *results* array.
An important feature of the **pfor** loop is that variables declared in the loop condition are passed by value to each thread, while all other variables are passed by reference, and thus may be modified. Thus, each thread contains a copy of the index variable **i** but may directly modify **results**.

Spawn Operator

The **spawn** operator creates a thread and returns a handle object which allows the result of the thread to be retrieved. **spawn** takes a single argument which must be some sort of function invocation. This might be an actual function, a method call, etc. The list of supported types is:

- Literal function call: `spawn(foo());`
- Function pointer call: `p = foo; spawn(p());`
- Method call w/reference: `spawn(a.b());`
- Method call w/pointer: `spawn(a->b());`
- Static method call: `spawn(A::b());`
- Method pointer w/reference: `p = &A::b; spawn(a.*p());`
- Method pointer w/pointer: `p = &A::b; spawn(a->*p());`

The function's arguments are evaluated immediately; a thread is then launched of the function with its arguments. The **spawn** operator returns an object of type `Result<T>`, where **T** is the return type of the invoked function. Calling the `value()` method returns the result of the thread; if the thread is not yet finished, it will block. Calling `wait()` will wait until the thread is finished and calling `kill()` will terminate the thread. In the latter case, the result of the thread will be the default constructor value of the return type.
Note that in order for a function to be used with `spawn`, its return type must have a default constructor.
A simple example is:

```
double foo(double a,double b)
{
  int xx = 0;
  for (int i = 0; i != 100000000; ++i) {
    xx += 1;
  }
  return a*a + b*b;
}

int pMain(int argc,const char *argv[])
{
  Result<double> r1 = spawn(foo(1.1,2.2));
  Result<double> r2 = spawn(foo(2.7,9.8));
  cout << "Result is:  " << r1.value()
       << ", " << r2.value() << endl;
  return 0;
}
```

If the spawned function has a void return type, then the `Result` type will be an integer and will always be 0. Since `int` is the default type for `Result`, you can declare an object in this manner:

```
void foo();

Result<> r = spawn(foo());
```

A common use of the **spawn** operator is to launch a series of threads, push their **Result** objects into a container, then iterate over that container, testing each value with a predicate. This act causes the parent (the thread that is doing the iteration) to wait until all threads are done, acting as a synchronization point.

Plasma provides templates for doing this. The most general case is called `ResultCheck<typename R,typename Pred>`, where `R` is the type returned by the spawned functions and `Pred` is a predicate function. The user can then call `ResultCheck::check()` to perform the iteration. If the predicate fails (returns false), `check()` stops and returns false.

Frequently, `ResultCheck` is used to check that all threads return an expected value, e.g. true. The `ValueCheck<T>` class will do this. It accepts the expected value as a constructor argument. A helper function `make_valuecheck()` simplifies the creation of the object.

For instance:

```
ValueCheckGen<Foo> results(make_valuecheck(foo));
```

where **Foo** is a type and **foo** is an instance of the type, e.g. `bool` and `true`. You then add threads like this:

```
results.push_back(spawn(myfunc(1,2,3)));
```

And check the results like this:

```
if (!results.check()) { return false; };
```

You can reuse it by calling clear:

```
results.clear();
```

Since a boolean is frequently used, a typedef is provided, called `BoolCheck`. Use this to capture and test the result of spawned functions which return booleans.

1.2.2 Thread Control

The following functions provide control over threads. These are declared in `plasma-interface.h` which is implicitly included in all plasma (.pa) files.

- `THandle pCurThread()`: Return a handle to the current thread.
- `pWait(THandle)`: Wait until the specified thread is finished.
- `pYield()`: Have the current thread swap to the next ready thread.
- `pTerminate()`: Kill the current thread.
- `pLock()`: Turn off time-slicing.
- `pUnlock()`: Turn on time-slicing.
- `pExit(int code)`: Terminate the program with the specified exit code.
- `pAbort(char *)`: Abort the program gracefully with error message and return exit code -1.
- `pPanic(char *)`: Abort program immediately with error message and return exit code -1.

1.2.3 The Time Model

Plasma implements a time model so that users may experiment with mapping an algorithm to a possible hardware configuration. The model works along the lines of a discrete event simulator: A thread may delay itself, in which case it stays idle until a specified amount of time has passed, or it may explicitly consume time. In other words, actual work done by the thread takes zero time but to model algorithmic complexity, explicit calls may be made to simulate a piece of hardware doing real work.
Time in plasma consists of discrete time ticks, but no unit is associated with the time. The main time type is a 64-bit integer.
There are three main functions for the time mode:

- `pDelay(x)`: Delay for **x** time units. The thread will be idle for this period of time.
- `pBusy(x)`: Consume **x** time units. This means that the processor is "busy" for this long. In order to use `pBusy()`, you must set `ConfigParms::_busyokay` to true. If not, a runtime error will occur when the function is called. If you are in the busy-mode then preemption is disabled; the only thread switches will be during alt, wait, delay, or busy commands.

  Lowest priority threads are time-sliced. The time slice value is set by setting `ConfigParms::_simtimeslice` in `pSetup()`. What this means is that a busy command will be divided up into these timeslices, allowing the same processor to squeeze in work from other threads.
- `pTime()`: Returns the current system time.

The user may declare multiple processors by declaring a **Processor** object, e.g.:

```
Processor a;
```

To create a vector of Processors, use the `Processors` data structure:

```
Processors procs(10);
```

New threads will be launched on their parent's processor unless one of the following mechanisms is used:

- With a **par** or **pfor** block, using an **on** block:

```
par {
   on (<proc name>) { ... }
   ...
}
```

- With the spawn command:

```
<proc name>.spawn(<command>);
```

Each **Processor** may be given a name by passing a `const char *` to the constructor or the **setName** method. This pointer is unmanaged by the **Processor** and is simply stored as-is. This means that you can compare it against another pointer for quick identification purposes.
By default, each processor has its own issue queue and threads must be explicitly launched on another processor. However, Plasma has support for allowing a group of processors to share a common issue queue. This allows for the modeling

of SMP systems. To do this, the user either creates a new processor using the `make_sharedproc()` function, or creates an array of processors by specifying *true* for the third argument of the `Processors` constructor:

```
Processor a = make_sharedproc(pCurProc());

Processors procs(10,"terminals",true);
```

In the above code, processor `a` shares its ready queue with the current processor and the ten processors declared by `procs` all share the same ready queue.

In the following example, even though all of the threads are launched on the same processor (element 0 of the `procs` array), they all finish in 50 time units. This is because the other nine processors share the same ready queue and are able to schedule these threads.

```
Processors procs(10,"processors",true);

pfor (int i = 0; i != 10; ++i) {
  on (procs[0]) {
    cout << "Block " << i << " start:  " << pTime() << endl;
    pBusy(50);
    cout << "Block " << i << " done:  " << pTime() << endl;
  }
}
```

The output from this program fragment is:

```
Block 0 start:  0
Block 1 start:  0
Block 2 start:  0
Block 3 start:  0
Block 4 start:  0
Block 5 start:  0
Block 6 start:  0
Block 7 start:  0
Block 8 start:  0
Block 9 start:  0
Block 0 done:  50
Block 8 done:  50
Block 1 done:  50
Block 9 done:  50
Block 7 done:  50
Block 5 done:  50
Block 2 done:  50
Block 6 done:  50
Block 4 done:  50
Block 3 done:  50
```

1.2.4 Priorities

Threads in Plasma have priorities. By default, a thread's priority is the same as its parent's priority, with `pMain()` starting at the lowest priority. Priorities are specified as an integer, where 0 is the highest. The number of priorities may be set by the configuration parameter `ConfigParms::_numpriorities`. The default value is 32.
Priorities may be specified using a functional API or as optional arguments to **spawn** or **on**:

- `pSetPriority(int)`: Set current thread's priority.
- `pGetPriorities()`: Return current thread's priority.
- `pLowestPriority()`: Return the lowest priority (timeslice queue).
- Optional second argument to **spawn** of a priority, e.g. `spawn(foo(),0);`
- Optional second argument to **on** block of a priority, e.g. `on(p1,0) { ... }`

For any given processor, threads execute in priority order, with timeslicing only for the lowest priority thread. Of course, if a higher priority waits on another

thread or enters an alt block, there is the possibility that a lower priority thread may execute. In other words, given the following code fragment:

```
par {
  on (pCurProc(),0) { /* thread 1 */ }
  on (pCurProc(),1) { /* thread 2 */ }
  { /* thread 3 */ }
  { /* thread 4 */ }
}
```

Thread 1 will execute first and complete before thread 2, which will also execute to completion before threads 3 and 4. Threads 3 and 4 will execute in a time-sliced fashion. Note the use of `pCurProc()`: The only way to set a priority with the **on** block is to use two arguments. If you want the thread to execute on the current processor, you must call `pCurProc()` to return the current processor.

1.2.5 Thread Communication

Channels

One method for threads to communicate among themselves is to use a channel. This is simply a data structure which allows one thread to write a value to it and another thread to read this value. It is up to the channel to make sure that these operations are safe and to ensure proper flow control. Any class may be a channel as long as it has a specific interface. This interface is required in order to use the **alt** and **afor** constructs.
The required interface for a channel of type **T** is:

- `T read()`: Returns a value read from the channel. Blocks if no value is present. Returns the last value read, until clear_ready() is called.
- `T get()`: Returns a value from the channel. Blocks if no value is present. Always fetches a new value. After a call to this, `read()` will return this same value.
- `void write(T)`: Writes a value to the channel. May block, depending upon the channel definition.
- `bool ready() const`: Returns true if the channel has a value.
- `bool full() const`: Returns true if the channel would block if written to.
- `void clear_ready()`: Clears the ready status, forcing the fetch of a new value.
- `set_notify(Thread *t,int handle)`: Stores the thread and handle. When the channel gets a value, it will wake this thread, giving it the handle.

- `clear_notify()`: Clears the stored thread so that no notification will take place if a value is written to the channel. It must be possible to call `clear_notify()` safely, e.g. this should not affect the behavior of a blocked writing thread.

- `value_type`: A typedef for the data type of the channel.

Note that `write()`, `read()`, `clear_ready()`, and `full()` are technically not required by **alt** and **afor**. Thus, it is possible to have a read-only channel. Currently, Plasma contains the following channels. These are declared in `plasma.h`.

- `Channel<class Data,class Base = SingleConsumerChannel>`: This is a typed channel which reads and writes an object of type *Data*. It contains only a single copy of this object; a second write will block if the first write's data has not been read. It may be used with multiple producers. By default, only a single consumer is allowed. To enable multiple consumers, specify `MultiConsumerChannel` for the second template argument *Base*.

- `BusyChan Data,class Base = SingleConsumerChannel>`: This channel is similar to `Channel`, except that if a read blocks, it places the thread's processor into a busy state. This can be used for when waiting on a resource holds up a task, e.g. a processor waiting on a load cannot do something else. The user can specify a timeslice value in the constructor, or specify 0 to mean no timeslicing.

- `QueueChan<class Data,class Base = SingleConsumerChannel>`: This is a typed queued channel: It allows for multiple producers and supports multiple consumers if the *Base* argument is set to `MultiConsumerChannel`. By default, the queue size is not fixed, but the user may set a maximum size by specifying it as the constructor argument.

- `Timeout`: Use this to break out of an **alt** block after a specified amount of simulation time. It uses `pDelay()` to block for a given amount of time. If nothing else has awakened the alt block thread before then, this will. It does not return a useful value, so it is generally used with an empty port statement, e.g.:

```
Timeout t(20);
alt {
  c0.port(...) { ... }
  c1.port(...) { ... }
  t.port() { cout << "Got a timeout!" << endl; }
}
```

  In the above example, a **Timeout** object is created which will awaken an alt block after 20 time units.

- `ClockChan<class Data,,class Base = SingleConsumerClockChannel,class Container = list<Data> >`: Clocked channels allow for the easy modeling of clocked hardware. The basic idea is that data may be written at any time, but the channel may only be read at certain intervals. This interval is specified by the constructor. For instance, a clocked channel with a period of 10 will block until the time is a multiple of 10. This class can be used in a queued or non-queued manner, also specified by a constructor argument. The third template parameter allows the user to specify an alternate container to hold the queued data.

  The basic idea of clocked channels is to let the channel worry about time, rather than the user's code. In other words, to model a pipeline, each pipeline stage acts as a consumer of data from the prior stage and a producer for the next stage. The communication occurs through clocked channels so that the timing is invisible to each stage, and is instead coordinated solely by the channels.

  Multiple producers are supported. Multiple consumers may be allowed by specifying `MultiConsumerClockChannel` for the second template argument.

  If the clock period is set to 0, the behavior is the same as an unclocked channel. This allows the user to switch between having a channel be clocked or unclocked by simply modifying a constructor argument, rather than changing the channel's type.

- `ResChan<class Data>`: The **spawn** operator may be interfaced to an **alt** construct by using this class. This is a read-only channel which will return the result value of the spawned thread. For example:

```
double foo(double a,double b)
{
  int xx = 0;
  for (int i = 0; i != 100000000; ++i) {
    xx += 1;
  }
  return a*a + b*b;
}

int bar(int a)
{
  return a * a * a;
}

void check(ResChan<double> &a,ResChan<int> &b)
{
  for (int i = 0; i != 2; ++i) {
    alt {
      a.port(double x) {
        cout << "x:  " << x << endl;
      }
      b.port (int y) {
        cout << "y:  " << y << endl;
      }
    }
  }
}

int pMain(int argc,const char *argv[])
{
  ResChan<double> r1 = spawn(foo(1.1,2.2));
  ResChan<int> r2 = spawn(bar(123));
  check(r1,r2);
  return 0;
}
```

Multiple Consumers

The basic Plasma channels all expect there to be a single consumer of data unless explicitly configured to support multiple consumers. In such a case, only one consumer will receive the data. This is useful in a situation, for example, where a group of consumers waits for the next available job. However, it does not allow for the broadcasting of information to a group of consumers all at

once. Instead, the `Broadcaster` class exists. It is an object which has an input channel and one or more output channels: Any data sent to the input is reflected to the output. It is templated on both its input and output channel types. Another class, `ClkBroadcaster`, specializes `Broadcaster` to work with clocked systems.

For example, you might use this in the following way:

```
typedef Channel<int> IntChan;
typedef Broadcaster<IntChan,IntChan> IntBroadcaster;

int pMain(int argc,const char *argv[])
{
  const int Num = 5;
  IntBroadcaster ib;

  par {
    producer(10,ib.get_source());
    pfor (int i = 0; i != Num; ++i) {
      consumer(i,ib.get_sink());
    }
  }
  plasma::mprintf ("Done.\n");
  return 0;
}
```

Calling `get_source()` returns the input channel, which is of type `Broadcaster::input_channel`. Calling `get_sink()` adds a new output channel to the broadcast object and returns a reference to it. This channel is of type `Broadcaster::output_channel`. Each time that the producer writes data to the input channel, it is broadcast to all of the consumers.

On the other hand, this only works if data may be replicated. If this is not valid, e.g. a transaction should not be copied but instead should be consumed by any of several listening consumers, then it is necessary to configure a channel to allow multiple consumers. This is done by specifying `MultiConsumerChannel` for the *Base* template parameter of non-clocked channels and `MultiConsumerClockChannel` for clocked channels. This does impose some slight overhead for bookkeeping, so it is generally recommended to keep a channel as single-consumer only unless otherwise necessary.

When a channel is configured as allowing multiple consumers, more than one thread may do a *get* from that channel, or use it in an *alt* statement. For example:

```
typedef ClockChan<int,MultiConsumerClockChannel> IntChan;

void producer(int id,int val,IntChan &chan)
{
  for (int i = 0; i != 10; ++i) {
    chan.write(val+i);
  }
  plasma::mprintf ("Done.
");
}

void consumer(int id,IntChan &c0,IntChan &c1)
{
  int foo = 10;

  while (true) {
    alt {
      c0.port(int d) {
        cout << id << ":  Port 0, Time " << pTime() << ":  Got " << d << end
      }
      c1.port(int d) {
        cout << id << ":  Port 1, Time " << pTime() << ":  Got " << d << end
      }
    }
    pDelay(20);
  }
}

int pMain(int argc,const char *argv[])
{
  IntChan chan0(5), chan1(5);

  par {
    producer(0,10,chan0);
    producer(1,100,chan1);
    consumer(0,chan0,chan1);
    consumer(1,chan0,chan1);
  }
  return 0;
}
```

In the above example, two producers feed to two consumers. Both consumers read from these channels. This means that if the first consumer reads from `c0`, then the second consumer will then read from`c1`.

Alt Blocks

An **alt** block allows for unordered selection of data from channels. Its syntax is:

```
alt {
  <channel expr> [ . | -> ] port (<value decl>) { <body> }
  [ alt { ... } ]
  [ afor { ... } ]
  [ { <default block> } ]
}
```

Each **port** statement specifies a channel to be read (the channel expression) and an optional declaration which will receive the channel value. The **port** body has access to this value. If no value declaration is specified, the channel's data is not accessible. This is useful for channels whose data is simply a boolean state, such as a time-out channel.

Upon entry to the **alt** block, all channels are checked for data. If a channel has data, the body of the corresponding **port** statement is executed. If no channels are ready, the thread will sleep until a channel has data. If more than one channel is ready, a single port statement is selected non-deterministically.

The **prialt** block should be used when the order of the channels is important. For example, use **prialt** when a *stop* channel should have a higher priority than the normal data channels. The syntax is exactly the same; the only difference is that the channels are guaranteed to be scanned sequentially, from the top down, when determining if any are ready.

If a default block is specified, the **alt** block will never cause the thread to sleep. Instead, if no channels have data, the default block will be executed.

alt and **afor** (explained below) blocks may be nested within **alt** blocks. This allows the user to block on multiple collections of channels, or a collection of channels plus one or more single channels, etc.

Afor Blocks

An **afor** block is similar to an **alt** block, except that it allows the user to loop over a data structure of channels. Its syntax is:

```
afor ( <s1> ; <s2> ; <s3> ) {
  <channel expr> [ . | -> ] port (<value decl>) { <body> }
  [ { <default block> } ]
}
```

Only a single **port** statement is allowed. The **afor** block is treated as a for-loop, looping over all channels specified by the channel expression. An iterator variable must be declared in *s1*; its value is accessible to the channel expression and the **port**'s body.

For example, the following code loops over an array of channels. As in the **alt** block, the thread will sleep if no channels are ready and there is not a default block.:

```
afor (int i = 0; i != (int)channels.size(); ++i) {
  channels[i].port (int v) {
    printf ("Got a value from port %d:  %d\n",i,v);
    if (v < 0) ++donecount;
  }
}
```

Plasma allows for non-integer index variables but this requires the creation of an auxiliary data structure, so performance might be a little slower, e.g. using an iterator rather than an integer as an index.
There are a few restrictions to follow for the **afor** block:

- You must declare the loop iterator in the first statement of the **afor** block.
- The loop will occur multiple times, so make sure that there are no side-effects.
- You only have access, within the **port** body, to the first loop iterator variable. Therefore, avoid fancy **afor** loops which declare multiple variables in the first statement or update multiple variables in the third statement.

As noted above, an **afor** block may be nested within an **alt** block. This allows you to block on one or more collections and/or to block on a collection plus one or more single channels. For example, the following code will block on a collection and an override channel:

```
alt {
  afor (int i = 0; i != (int)channels.size(); ++i) {
    channels[i].port (int v) {
      printf ("Got a value from port %d:  %d\n",i,v);
      if (v < 0) ++donecount;
    }
  }
  stopchan.port (bool b) {
    if (b) {
      printf ("Got a stop command!\n");
    }
  }
}
```

A prioritized version of **afor** exists and is called **priafor**. This guarantees that the channels will be scanned sequentially. In general, the **afor** and **priafor** will

behave identically. However, if the compiler can determine that the loop is constant, it may choose to unroll an **afor** and then re-order the channel querying.

Shared Data Structures

Threads may also communicate using shared data structures whose access methods are protected by special synchronization primitives. There are two means to do this. The easiest is to declare a class as being a mutex class:

```
pMutex class X { };
```

This will wrap all public methods of class **X**, except for constructors and its destructor, with serialization code. To prevent this on a per-method basis, use the modifier *pNoMutex*:

```
pMutex class Foo {
public:
  // Not protected.
  Foo();
  ~Foo();
  // Protected.
  int a();
  // Not protected.
  pNoMutex int b();
private:
  // Not protected.
  int c();
};
```

Be careful with using *pNoMutex*: Since it disables serialization, it is inherently dangerous. It is useful, though, when you have a constant method whose return value would not be affected by a thread preemption. For example, a method which returns a constant which is only initialized at construction time.
The other method for creating a shared data structure is to directly use the `pLock()` and `pUnlock()` primitives. This is more error prone than using *pMutex* but might be necessary in some cases, such as for protecting a plain function:

```
void msg(const char *fmt, ...) {
  pLock();
  va_list ap;
  va_start(ap,fmt);
  vprintf(fmt,ap);
  va_end(ap);
  pUnlock();
}
```

1.3 Power Modeling

Plasma has some simple hooks for supporting power modeling. The concept is similar to the way in which time is handled: A user annotates a model with explicit energy-consumption information, which is stored on a per-processor basis. To obtain power, a thread can be created which regularly queries a processor for its energy consumption and divides by the sample period.
The power API is:

- `energy_t`: The basic energy type. Currently defined as a `double`
- `void pEnergy(energy_t)`: Adds the argument to the current processor's energy consumption value.
- `energy_t pGetEnergy(Processor p)`: Returns the specified processor's energy consumption value. This clears the value.
- `energy_t pReadEnergy(Processor p)`: Returns the specified processor's energy consumption value. This does not clear the value.

A reporting thread might look something like:

```
void power(int period)
{
  while (1) {
    cout << pTime() << ":  current power:"
         << (pGetEnergy(pCurProc())/period) << "mW" << endl;
    pDelay(period);
  }
}
```

Interspersed throughout the code would be calls to record energy usage:

```
...
pEnergy(ops * ScaleValue);
...
```

A more sophisticated approach might construct a power-model by scaling energy values by the clock frequency. Thus, the model might not call `pEnergy` directly but would instead call a helper function, e.g. `myEnergy`, which would scale the energy argument and actually call `pEnergy`. Other possibilities might include constructing a special channel class that remembers a previous value and records energy usage based upon the hamming distance between the previous and current value, or have a very low-priority thread which adds energy when a processor is idle, attempting to model leakage current.

Of course, the power modeling API can be thought of as a specific instance of the more general concept of storing a value with a particular processor. In fact, this is exactly what is done: The power and energy API is based upon the `ProcValue` class, which lets the user hash a value against a processor. Its API is:

```
template <typename Data>
struct ProcValue : public hash_map<Processor,Data,HashProc> {
  // This adds to the value for the specified processor.
  Data add(Processor p,Data d);
  // This adds to the current processor.
  Data add(Data d);
  // This gets the processor's value and clears the value.
  // Returns Data() if the item is not there.
  Data get(Processor p);
  // Reads the value, returns Data() if item is not there.
  Data read(Processor p) const;
};
```

This is a template so that any kind of value can be stored. The `add` functions add their argument to the existing value associated with the specified processor and return this result. The `get` routine returns the current value and clears it in the hash, while the `read` routine returns the value without modifying the hash.

1.4 Resource Modeling

The need to model an object with limited resources frequently arises. For example, a memory controller has a specific, fixed bandwidth. Plasma provides a class, called `Quantity`, to help with this task. It works as follows: The user declares an instance of the class and specifies a timeslice as a constructor argument. The timeslice refers to the time it takes to do one request, e.g. for a memory controller, a single read may take 10 time units.
Once declared, threads may request an amount n, which will correspond to n requests. A thread will busy its processor until the request is satisfied. The `Quantity` class uses a round-robin scheme to satisfy all in-bound requests: For each thread, it decrements the request by one and delays by the specified timeslice. The busy processor is time-sliced so that higher-priority threads may gain some processing time.
The interface is:

- `Quantity::Quantity(int timeslice)`: Declare an instance and specify the timeslice amount.
- `Quantity::request(int amount)`: Request a specific amount. If there are no other requests, then this can be satisfied in *(timeslice * amount)* time units, but other requests will stretch this out.

1.5 Garbage Collection

Plasma is equipped with a garbage collector. This means that heap-allocated objects do not need to be explicitly freed; they will be collected when no more pointers to the object remain. This implicitly managed memory (referred to as simply *managed* from now on) may be used alongside explicitly managed memory, where a call to delete is required.
By default, allocations using **new** are explicitly managed. To allocate managed memory, you may derive an object from **gc** or **gc_cleanup** or allocate using the **GC** or **UseGC** placement attribute.
For example, the following class will be managed when allocated using **new**:

```
class A : public gc { };
```

However, its destructor will not be called. If you derive from **gc_cleanup**, the class's destructor will be called when the object is collected.
An example of using the **GC** placement attribute is:

```
int *x = new (GC) int[1000];
```

This will allocate a block of memory that is managed by the collector.
You may delete memory that is managed, but this is generally discouraged, since the collector will collect it when it is safe to do so.

1.6 Library Features

This section describes functions and classes provided by the Plasma standard library, `plasma.h`.

1.6.1 I/O Routines

A series of **printf**-style routines are provided which implement mutex-protected I/O routines. These are:

- `int mprintf(const char *format, ... );`
- `int mfprintf(FILE *,const char *format, ...);`
- `int mvprintf(const char *format, va_list ap);`
- `int mvfprintf(FILE *,const char *format,va_list ap);`

These behave just like the standard routines, except that they are protected from preemption while they are executing.

1.6.2 String Routines

Strings, such as **Processor** names, are garbage collected in Plasma. This may interfere with using the C++ string class, since it generally manages its own memory. Thus, when passing a string to a **Processor** via `setName()` or a constructor, if the string is not a constant, you should duplicate it using `gc_strdup()`.

- `char *gc_strdup(const char *);`
- `char *gc_strdup(const std::string &s);`

Both of the above routines duplicate their argument using GC-allocated memory.

1.6.3 Random Number Generation

The template `Random<Gen>` is provided for random number generation. It takes as a parameter a generator class, which does the actual work of generating pseudo-random numbers. The *Random* class wraps this generator and provides various functionality, such as supporting **N** streams of independent random number generation and the ability to read and write the state of these generators using C++ streams.
Several different generators are provided:

- **LcgRand**: Linear-congruential generator with period of 2^{32}. This is a reversible function. Its state is stored in one 32-bit word.
- **KissRand**: A combination of several generators, it has a period of 2^{127}. It is not reversible. Its state is stored in 5 32-bit words.
- **MtRand**: The Mersenne Twist random number generator has a period of 2^{19937}-1. It is reversible. Its state is stored in 625 32-bit words.

The default generator is **KissRand**.
The simplest way to use **Random** is to simply generate an unsigned integer by calling `genrand()` or generate a double in the range of [0,1] by calling `gendbl()`. For example:

```
Random<> Rand;                 // Instantiate w/default generator.

unsigned x = Rand.genrand();  // Generate uint in [0,0xffffffff].
double y = Rand.gendbl();     // Generate double in [0,1].
```

To create 3 independent generators with the Mersenne Twist algorithm:

```
Random<MtRand> Rand(3);

unsigned x = Rand.genrand(0); // Generate using stream 0.
unsigned y = Rand.genrand(1); // Generate using stream 1.
unsigned z = Rand.genrand(2); // Generate using stream 2.
```

The class can then be saved to a stream, for example to checkpoint the current state of a simulation:

```
ostringstream os;
Rand.save(os);                // Save the generators' state.

istringstream is(os.str());
Rand.load(is);                // Load the generators' state.
```

Random may also generate numbers using several distributions. Currently, the distribution functions only support unsigned integers. The various distributions are shown below. For each function, the first parameter specifies which stream to use for generation.

- `unsigned uniform(unsigned s,unsigned base,unsigned limit)`: Generates a uniform random number in [*base*, *limit*]. The *limit* must be greater than *base*.
- `unsigned triangle(unsigned s,unsigned l,unsigned mode,unsigned u)`: Generates using a triangle distribution in [l, u "), with the peak of the distribution specified by *mode*. Upper must be greater than lower.
- `unsigned exponential(unsigned s,unsigned scale,double lambda)`: Generates an exponential distribution in [0, *scale* ") with *lambda* describing the fall-off rate.
- `unsigned normal(unsigned s,unsigned mean,double std_dev)`: Generates a normal distribution with the curve centered around *mean*.

Chapter 2

Examples

This chapter demonstrates the use of Plasma in several areas of modeling and application development.

2.1 Clocked Hardware Modeling

In this section, we show how Plasma can be used to model clocked hardware. Specifically, we model a very simple RISC pipeline. This is done using Plasma's clocked channels **??**. The result is that no stage in the pipeline needs to know about time or the clock this is completely encapsulated within the channels. The timing of the whole system can thus be modified simply by changing the declarations of the clocked channels.

The source code for this example can be found here. It is organized such that each pipeline stage is a separate thread. Examining the main entry point function, `pMain`, you can see that all channels are declared here, as well as all pipeline stages:

```
int pMain(int argc,const char *argv[])
{
  MemClkChan fetch_icache(Clock,0,1), icache_decode(Clock,0,1),
    alu_dcache(Clock,0,1), regwrite1_dcache(Clock,0,1);

  PipeClkChan decode_regread(Clock/2,0,1), regread_alu(Clock,0,1),
    regread_muldiv(Clock,0,1), alu_regwrite1(Clock,0,1),
    muldiv_regwrite2(Clock,0,1), dcache_regwrite2(Clock,0,1);

  Chan decode_fetch, alu_fetch, icount_watchdog, fetch_watchdog;

  printf("\n\n\n\n"
         "the pipe: a simple risc pipeline simulator v0.06\n"
         "copyright  motorola 1997-2003\n");

  initCode(10000);          // create some instructions
  //initCode(10);
  initRegs();                    // set up registers
  printf("\nCode and registers initialised.");
  par {
    //===logging
    watchdog(20000, icount_watchdog, fetch_watchdog);
    {
      Fetch fetch(0, decode_fetch, alu_fetch, fetch_icache,
                  fetch_watchdog, 0);
      fetch();
    }
    icache(1, fetch_icache, icache_decode);
    {
      Decoder decoder(2, icache_decode, decode_fetch,
                      decode_regread, icount_watchdog);
      decoder();
    }
    regRead(3, decode_regread, regread_alu, regread_muldiv);
    alu(4, regread_alu, alu_regwrite1, alu_dcache, alu_fetch);
    muldiv(5, regread_muldiv, muldiv_regwrite2);
    regWrite1(6, alu_regwrite1, regwrite1_dcache);
    dcache(7, alu_dcache, regwrite1_dcache, dcache_regwrite2);
    regWrite2(8, muldiv_regwrite2, dcache_regwrite2);
  }
  printf("\n\n***All stopped.****\n");
  printRegs();
  return 0;
}
```

This is a frequently used programming idiom in Plasma models: A single function defines the channels used for communication among the components of the simulation and how they are wired together. This makes it easy to get a top-level view of how the model is assembled.

One other point of interest: Notice that the channel named `decode_regread` has a period of half the clock. As noted at the top of the source file, this is because register reads occur on a half-clock boundary. This shows that the timing of the particular channel, and thus of the model, can be altered by simply changing a declaration.

Another example of modeling hardware can be found here. In this example, the architecture of the processor is implemented in an auxiliary class called `DLX`. The Plasma code simply implements the microarchitecture. Communication is done by channels, as in the first example. One difference is that the `NoBlockChan` channel is used to represent simple registers: Writes never block; they simply overwrite existing data. Reads do not block either; they return the last data item until time advances, at which time they return the new data item. This channel is simply a template class and thus does not require any modification to Plasma itself.

2.2 Network System Modeling

In this section, we show how Plasma can be used to model a large-scale networked system. The model is of a database server, where a series of terminals send requests to a terminal server, which then dispatches queries to a database. The database either finds that it can answer the query by accessing memory or else it has to dispatch a request to a disk array. The decision factor, in this case, is a uniform random pick with a set probability.

The source code for this model can be found here.

In this example, different threads are explicitly run on different processors, as shown below:

```
// create the dynamically-sized interconnect
Chans
  terms_to_comms(actual_users),
  comms_to_terms(actual_users),
  comms_to_tp(tp_agents),
  tp_to_comms(tp_agents),
  free(tp_agents),
  tp_to_disks(tp_agents),
  disks_to_tp(tp_agents);

par {
  par {
    on (mainframe) {
      comms_in(mips, terms_to_comms, comms_to_tp, free);
    }
    on (mainframe) {
      comms_out(mips, tp_to_comms, comms_to_terms);
    }
    on (mainframe) {
      DiskManager disk_manager;
      disk_manager(mips,buff_size,disks,tp_to_disks,disks_to_tp);
    }
    on (mainframe) {
      Agents agents;
      agents(mips, comms_to_tp, tp_to_comms, free,
             tp_to_disks, disks_to_tp, tp_agents, hit_rate);
    }
    on (mainframe) {
      timestamp(ReportInterval, SimulationTime * sec,
                transactions, times);
    }
  }
  {
    pfor (int i = 0; i != actual_users; ++i) {
      on (terminals[i]) {
        user(comms_to_terms[i], terms_to_comms[i],
             i, SimulationTime * sec, transactions, times,
             sigma_transac, sigma_response);
      }
    }

    say("\nThe users have all finished..\nTotal of ");
    saynum(transactions);
    say(" in ");
    printfixed(pTime(), sec);
    say(" seconds for a transaction rate of ");
    //mprintfixed(transactions * sec, Now);
    printfixed(transactions, pTime()/sec);
    say (" tps\n");
    pExit(0);
  }
```

The *mainframe* processor runs the communications server, the database server, and the disk array, while each terminal runs on its own processor. Each of the threads running on the mainframe consume time by calling the function *think*, which simply calls *pBusy* **??**. This allows the user to judge how loaded the system is. If it cannot keep up with the number of inbound requests in the simulation, then tasks can be offloaded very easily, e.g. the disk array can be run on another processor. In other words, trade-off analysis can be performed in a simple fashion by simply adding or removing processors and moving threads accordingly.

As noted earlier, the decision of whether a request can be satisfied by a memory access or a disk access uses Plasma's *Random* **??** class. For example, the `mem_read` function in eav.pa picks a random number and sees if it is within the hit rate. If it is, then it simply delays, simulating a memory access. Otherwise, it dispatches a request to the disk array. Randomness is also used in other places within the model. For instance, in randomizing the delay between requests dispatched by the terminals and within the disk controller, simulating different seek times for the harddrive heads.

2.3 Embedded Applications.

In this section, we show how Plasma can be used for the development of real-time embedded control software by implementing a very simple model of an engine controller. The model is quite simplistic: The hardware is modeled as a flywheel and an engine. The flywheel has a concept of friction, so that it slows down if the engine is not accelerating it. The engine contains several cylinders; based upon the amount of fuel added, the model calculates the new speed of the engine.

The source code can be found here.

The engine model and the controller are started in ecu.pa. The idea is that a separate thread is launched for each cylinder in the engine. The *pBusy* **??** function is used to consume time in order to understand the load on the microcontroller. Thus, if the load were too great for a single core device, the threads could be distributed over multiple cores by simply allocating another processor and distributing the threads appropriately.

2.4 Desktop Applications

In this section, we demonstrate Plasma's use for general application development by implementing a simple C compiler. This just handles a subset of C but it does illustrate how Plasma can be used for general application development. It produces x86 assembly code which can be assembled by the GNU assembler and is based upon a compiler written in Python by Atul Varma.

The compiler works as follows:

- The lexer, implemented in bison, operates in its own thread and tokenizes each input file (unlike most compilers, this one can handle more than one C file). These tokens are sent over a channel to the parser. The input file is memory mapped using `mmap` so that identifiers can be referenced using a pointer and length tuple. The only time that a new string has to be generated is when the tokenizer encounters C's automatic string concatenation. In that case, a new string is allocated using the garbage collector.

- The parser, implemented using lemon, also operates in its own thread. It just creates the AST. When the file has been completely parsed, it then spawns off a thread to perform the rest of the compilation steps for that translation unit. The lexer and parser threads are started in the main program here.

- The rest of the compilation steps originate here. For each translation unit, a symbol table is first created. This is done serially due to the sequential nature of symbol lookup in C. The symbol table is implemented as a two-step look-up process:

  - A single hash table is used to map symbols to small-integer indices.
  - The hash value is then used to index into an array that stores the current environment. This array stores pointers to a linked-list of environment objects.

  When a new scope is entered, a new array is created, which is just a copy of the parent's environment. When a new variable is encountered, a new environment object is allocated and the relevant location in the array is set to point to this object. If the array location is already occupied, then the new object is inserted at the front of the list for that entry.

  After the symbol table has been created, flow-control analysis and type checking are done in parallel for each function. Virtual methods in the AST class dispatch to the appropriate derived-type.

  Finally, code generation occurs. This is also done in parallel for each function. The code generation routines are contained within the CodeGen class. Virtual functions in the AST class dispatch to the appropriate code generation routines. Register allocation is done using a simple stack machine: If a push occurs and a free register does not exist, the oldest register is spilled into memory. A pop will cause the value to be loaded back into a register.

  Generated code is stored as text strings within a container. A container is allocated for each function and then attached to it. The final step in the compilation process is to iterate over the top-level elements of the translation unit, writing them to the output file.

Except for the lexer and parser threads, the spawn operator **??** is used to create new threads. The result is a boolean indicating success or failure. The result objects are stored into a container, then iterated over, causing the iterating thread to wait until the child threads are done. This idiom is a common one and so it has been included within the standard library as a template **??**.
Since Plasma offers garbage collection, the C compiler does not have to worry about freeing memory. This makes for a much cleaner implementation than would be possible with explicit memory management. For instance, the symbol table environments are pointed to be many different AST nodes. Figuring out who owns what would be difficult- an ad-hoc reference-counting scheme would probably have to be implemented, further complicating the code. The string class is even more complicated: Some of its strings will point to the mmap'd source file and some to the heap. However, the implementation does not need to care, since the heap strings will be collected by the garbage collector.

Chapter 3

Future Work

The current version of Plasma exists as a prototype. It is implemented as a front-end filter, using OpenC++, which converts Plasma code into C++ code and calls to a simple user-mode thread package. The threads are implemented using Quickthreads. If work on Plasma proceeds, it would be productized by a professional compiler company, such as Metrowerks.

3.1 Productization Improvements

This section lists various improvements which would be made for a production-ready version of Plasma:

- The final product will feature a thread-aware compiler which would produce more highly optimized code, e.g. on a thread-yield, the compiler would only save the state of live registers.

  A thread-aware compiler will be able to perform additional checking to ensure program correctness. For example, a non-mutex-protected variable will not be allowed to be written to by more than one thread unless the compiler can prove that no conflicts will arise.

  A mechanism will be added so that containers may be safely sliced, allowing multiple threads to work on non-overlapping portions. For example, two threads could work on two non-overlapping slices of an array by creating two array slices. This might be implemented in a manner similar to the `valarray` and `slice` classes of the standard library.

- The prototype version of Plasma uses only user-mode threads. In other words, all threads currently run in the same process. This will be enhanced to support kernel mode threads. The user will be able to designate that a `Processor` should be placed in another kernel thread. This will allow true multi-threading on a multi-way machine.

- The compiler will be aware of the garbage collector and will optimize accordingly. For instance, the default `new` operator for a class would allocate from the atomic heap if the class did not contain any internal pointers. The default behavior for allocation might change so that a class is allocated from the collected heap unless the user specifies otherwise. For example, the user must currently write:

```
class Foo : public gc
```

  to indicate that the class should be managed. Instead, we might change it so that the inverse is true:

```
class Foo : public NoGC
```

  to indicate that a class should not be managed. Otherwise, it will be.

  The underlying collector, however, would always have a conservative component so that unmodified C++ code could always be used with Plasma. Although we might change Plasma so that, by default, a class is allocated from the gc heap, the behavior of a C++ library that is linked in would not change.

  Other improvements might include scan functions for Plasma classes, compaction for Plasma classes, and incremental collection.

- Plasma is designed to be a garbage-collected language, so garbage collector aware containers are a necessity. The standard containers, e.g. `vector`, `list`, etc., would be traced by the conservative collector. Currently, one has to write:

```
vector<int,traceable_allocator<int> >
```

  This will become the default. Equivalent containers, e.g. `gc_vector`, `gc_list`, will exist which will allocate from the managed heap and will not invoke destructors when destroyed. An efficient, managed string class will also be added.

- To be useful for modeling hardware, Plasma will need a data-type for modeling arbitrary-width integers. This will be implemented using a templated numerical class such as that found in SystemC. In fact, we might just want to use the SystemC class, if licensing permits. It should be possible, though the use of template specialization, to create highly optimized versions for common cases. For instance, 32-bit or smaller bit

vectors would use standard integers, 64-bit or smaller integers would use long-long arithmetic, and 128-bit or smaller vectors might be optimized to use Altivec.

- For easily modifying configurations, a scripting interface is desirable. We would like to make the Plasma interface language independent and have the actual implementation be hidden behind an interface. For example, the interface might provide methods to run a script and query for name/value pairs of data.

3.2 Fault Tolerance Extensions

Fault tolerance will be added to Plasma in three main ways:

- All system signals will be mapped to C++ exceptions. So, for example, a segmentation fault will generate an exception which can be caught and handled using `catch` blocks, rather than having to muck around with POSIX signal handlers. This will be true for any special exceptions provided by a system for specific fault tolerance exceptions, e.g. memory errors, interface timeouts, etc.

- An exception which propagates to the top of a thread will be caught and held until another thread tries to wait on that thread. The exception will then be propagated to all waiting threads. This means that an exception in a `par` block will propagate to the scope outside of the `par` block. An exception which occurs in a function which is launched via the `spawn` operator will propagate to any thread which calls `value()` on the relevant `Result` object. In this way, exceptions will propagate across thread boundaries, allowing them to be dealt with via the appropriate `catch` block.

- A feature for specifying alternative implementations will be added to Plasma. This follows the basic concept of *recovery blocks*: If a particular piece of code fails, the system will recover and try an alternative method for doing the same thing.

  If a method has an attribute of `pRecover`, it will be replaced by a recovery block mechanism and the original body will be the first implementation tried. Other implementations may be specified by using the attribute `pOr`. For example:

```
class Foo {
public:
  pRecover int func(int,int);
  pOr      int func_alt1(int,int);
  pOr      int func_alt2(int,int);
private:
  ...
};
```

Each alternative must have the same signature as the original `pRecover` method, but the name may be different. The list of alternatives terminates at the first method not marked by a `pOr` attribute.

The above example would then be expanded into something like the following:

```
int Foo::foo(int x,int y)
{
  checkpoint();
  try {
    int rc = func_orig(x,y); // Original func().
    complete();
    return rc;
  }
  catch (...) {
    recover();
    try {
      int rc = func_alt1(x,y);     // Alternative 1.
      complete():
      return rc;
    }
    catch (...) {
      recover();
      try {
        int rc = return func_alt2(x,y);   // Alternative 2.
        complete();
        return rc;
      }
      catch (...) {
        recover();
        throw;                    // No remaining alternatives.
      }
    }
  }
}
```

The first thing that happens is that the `checkpoint()` method is called. If the class does not have this method, it is skipped. This method should perform whatever actions are required to save the state of the class. In some cases, such as in a purely functional algorithm, there is no need to do any check-pointing, while in other cases work may be required.

Each implementation is then tried in order. If the function succeeds, the `complete()` function is called, if it exists. This allows the class to clean up, such as by deleting a checkpoint, or committing a transaction. If the method does not exist, the implementation's result will be returned directly.

If an exception does occur, it is caught and the `recover()` method is called if it exists. The next alternative is then called. If no alternatives remain, the exception is propagated up the stack.

3.3 Real-time Programming Extensions

We propose adding various extensions to Plasma to make it easier to develop real-time embedded applications. For example, the `deadline` class will be used to state that a certain sequence of code (from the point of declaration to the end of the scope) must execute within the specified amount of time:

```
{
  deadline d(20);
  ... <code> ...
  // Execution must reach this point within 20 time units.
}
```

Otherwise, an exception will be thrown. The exception can then be caught by a recovery block and an alternative action taken.
It should be possible to statically analyze a Plasma program which is free of unbounded-loops. The `deadline` declarations can then be used by the static analysis tool as assertions and these can be checked by analyzing the relevant code sequences.

3.4 Language Enhancements

- Default initialization for all built-in variables will be added. This inconsistency in the C++ language is frequently the source of difficult to find bugs and is completely unnecessary: A good optimizing compiler can remove unneeded initializations. For example:

```
int i; // Would be initialized to 0.
```

- Null-pointer checking will be performed. This can be implemented without adding extra runtime overhead: A null-pointer access will cause a protection fault, which can then generate a C++ exception.

- A `let` statement which would allow for the declaration of variables without explicitly stating their type will be added. The type would be implied from the type of the initializing expression. For example:

```
let i = ValueCheck<int,std::bind2nd(std::equal_to<int>(),10)
```

- A `final` block will be added to the language. This may only follow a `try` or `catch` block and will be executed whether an exception occurs or not.
- A proper module system will be added. C++'s use of header files for conveying declaration information make for extremely inefficient compilation. This is compounded by the fact that templates need to be in header files. A proper module system could dramatically speed up development time.
- Nested functions and lambda functions with true closures will be added. A closure would only be allocated if control-flow analysis indicated that a function might escape its parent's scope. In that case, the relevant local variables would be placed in a closure. Thus, no extra overhead would exist for the usual case of un-nested functions or even for the case where a function did not escape.

  The main advantage of adding this feature is to be able to better utilize generic, highly-optimized functions such as those present in the STL. For instance, the `foreach` function becomes far more useful when a lambda function can be used directly as the functor argument. Since `foreach` can be specialized for different data structures, it can potentially yield a faster solution for iterating over a container than a for-loop.
- A meta-object protocol/advanced macro processor will be added. As mentioned above, the current Plasma implementation uses OpenC++ to convert Plasma into C++. By making such a feature a part of Plasma itself, new language features could be added without changing the actual language definition.

  This means that constructs such as `alt` and `par` could still be implemented by the macro facility; the parallelism in the language would be implemented as low-level primitives which would not be directly used by the user.
- Other possible language enhancements might include:
  - Pattern matching, such as that found in the Gont language.
  - An alternative system for declarations, such as that described in *The Design and Evolution of C++*, page 46 and implemented in the preprocessor Cmm.
  - An alternative template syntax. The goal is to make it easier to write compile-time polymorphic code. Therefore, we would like to experiment with how to make it much easier to write generic functions, possibly borrowing ideas from the ML family of languages.

3.5 Behavioral Synthesis

We would like to be able to synthesize Plasma to hardware. This has already been demonstrated for SystemC and Handel C. The latter technology is very

similar to Plasma in that its parallelism is also based on that found in Occam. Therefore, it should be possible to use an existing synthesis tool, with relatively minor modifications to the front-end to understand Plasma's features. We will designate a synthesizable subset of the language, such as requiring bounded loops, no dynamic memory allocation, etc.

The result would be a very powerful system: A user could use the same language for modeling a complex design, then directly synthesize a portion of it to an FPGA or an ASIC, while other parts of the program are compiled to run on a FreeScale PowerPC part.

Chapter 4

Plasma Development Schedule

This document describes PLASMA's development schedule. It consists of a series of entries, each representing a feature to be added. A feature is considered complete when its functionality, documentation, and corresponding regression tests have been committed to CVS. The list is in chronological order but this order may change based upon external factors.

4.1 Par Block

4.1.1 Status

Completed 4/23/2004

4.1.2 Description

A par block launches each of its constituent expressions as separate threads. It proceeds only when all threads are finished.

4.1.3 Implementation

- The block will be converted such that each expression becomes its own subroutine. The parameter passed in will be a structure whose members are pointers to any variables used by the expression.
- Each thread will be launched via a call to pCreate. Then a call to pWait will exist for each thread.

4.1.4 Dependencies

- Thread library

- Parser is able to parse "plain" block structures.

4.1.5 Regressions

- basic/par1.pa
- basic/qsort.pa

4.2 Pfor block

4.2.1 Status

Completed 4/29/2004

4.2.2 Description

For-loop syntax, where body of the loop is launched as a thread. Construct blocks until all threads are finished.

4.2.3 Implementation

Same as for par block, except that the argument structure is allocated on the thread's stack structure. Each variable declared in loop's guard is passed by value, while everything else is passed by reference.

4.2.4 Dependencies

- Thread library
- Parser is able to parse "plain" block structures.

4.2.5 Regressions

- basic/par2.pa
- basic/par3.pa

4.3 Channels (Basic Alt block)

4.3.1 Status

Completed for 5/12/2004

4.3.2 Description

Define channel interface and implement basic alt block. The alt block is like a case statement, except that each condition is a channel variable and a variable to map the channel's return value to. The block blocks until one of the channels has data. It then reads that data, maps it to the variable, and executes the code associated with that guard.Basic syntax is::

```
alt {
  <channel expr> [ . | -> ] port(<value decl>): ....
  [ { default block } ]
}
```

4.3.3 Implementation

A channel will be any type that has the required interface. This is compile-time polymorphism, similar to how templates work. The required interface is as follows. For a channel of type T:

- `T read()`: Returns a value read from the channel. Blocks if no value is present. Returns the last value read, until `clear_ready()` is called.
- `T get()`: Returns a value from the channel. Blocks if no value is present. Always fetches a new value. After a call to this, `read()` will return this same value.
- `void write(T)`: Writes a value to the channel. May block, depending upon the channel definition.
- `bool ready() const`: Returns true if the channel has a value.
- `void clear_ready()`: Clears the ready status, forcing the fetch of a new value.
- `set_notify(Thread *t,int handle)`: Stores the thread and handle. When the channel gets a value, it will wake this thread, giving it the handle.
- `clear_notify()`: Clears the stored thread so that no notification will take place if a value is written to the channel.

Some details about channel implementation:

- Call pSleep() to block. You must have stored a handle to the current thread somewhere else before this call, e.g. storing it in a channel member variable.
- Call pWake() to awaken a thread. The general protocol is that the waker clears the thread member variable of the channel and it does this *before* the call to pWake.

- Call pAddReady() to add a thread to the ready queue, but not make it active. No switching occurs (assuming processor is locked to avoid preemption).

- A call to read() or get() should clear any notification. Thus, with an alt block, only the channels that had set_notify() called need to have clear_notify() called if a ready channel is found. The actual ready channel should not have clear_notify() called, since there could be a blocked writer waiting to go.

Code conversion for the alt block will be:

- Shutdown preemption.

- Loop through all channels- if anything is ready, save handle and exit loop. Else, call set_notify with current thread and handle (integer index of loop).

- If nothing ready, sleep.

- Case statement on return value of sleep, or index value from loop in (2). Execute relevant code.

- Call clear_notify on all threads. Do this within a catch(...) block, too.

- Alt blocks consume values, i.e. they call get().

4.3.4 Dependencies

- Need channel definition.

- Add `int pSleep()`: Puts the thread to sleep. Returns integer when thread wakes.

- Add `void pWake(Thread *t,int h)`: Wakes thread, giving it h.

4.3.5 Regressions

chan1 - chan9.

4.4 Looping Alt Block

4.4.1 Status

Completed 5/12/2004

4.4.2 Description

Same as alt block, but allows the user to loop over a data structure. Syntax is:

```
afor ( <s1> ; <s2> ; <s3> ) {
<channel expr> [ . | -> ] port (<value decl>) { <body> }
[ { <default block> } ]
}
```

Only one port statement is allowed. An iterator variable must be declared in <s1>.

4.4.3 Implementation

Same as for alt, except that we replicate the loop condition as a for-loop each time we deal with channnels. If the iterator is not an integer, we create an auxiliary vector and store the values there. We then store the corresponding index of the entry as the handle in each channel.

4.4.4 Dependencies

Completion of alt.

4.4.5 Regressions

- basic/chan4.pa
- basic/chan5.pa
- basic/chan6.pa
- basic/chan7.pa

4.5 Spawn Operator

4.5.1 Status

Completed 5/18/2004

4.5.2 Description

Thread creation w/o synchronization, e.g.:

```
spawn { foo(1,2,3); };
```

Evaluates the argument (must resolve to a function or an object's member invocation). The argument is launched as a thread. The return value is an object which meets the specifications of a channel. It will also have additional operators for thread control:

- wait(): Wait for thread to finish.
- kill(): Kill thread.

The object will be a special type of channel, so you can use it in an alt block and attempts to fetch the value before the thread is finished will result in a block. Unlike other channels, it will only ever have a single value, so calls to clear_ready() will be ignored.
Spawn should handle all ways to invoke a function:

- Literal function call: spawn(foo());
- Function pointer call: p = foo; spawn(p());
- Method call w/reference: spawn(a.b());
- Method call w/pointer: spawn(a->b());
- Static method call: spawn(A::b());
- Method pointer w/reference: p = &A::b; spawn(a.*p());
- Method pointer w/pointer: p = &A::b; spawn(a->*p());

4.5.3 Implementation

- Registered as a function call of a special dummy class.
- Void functions not handled- everything returns a value.

4.5.4 Regressions

- spawn1
- spawn2
- spawn3
- spawn4

4.6 Shared Data Structures

4.6.1 Status

Completed 5/20/2004

4.6.2 Description

Shared data structures will allow serialized access to data, i.e. mutexes will wrap the actual data access, ensuring safe use between threads. The most likely syntax will be a class attribute, e.g. pMutex class ... The public methods will then be wrapped with mutex access code. A per-method modifier will allow this to be disabled (will implement only if easy to do with OpenC++).

4.6.3 Implementation

Straightforward use of OpenC++'s example "WrapperClass".
Variadic function support is not perfect but can be made to work. You can't write a true variadic function, e.g. `foo(const char *fmt,..)`, because you can't pass the variable argument list. Instead, you must write a va_list function directly, e.g. `foo(const char *fmt,va_list ap)`. Plasma will then create a variadic version and a v_list version for you that are wrapped with locking code.

4.6.4 Regressions

mutex1

4.7 Thread Priorities

4.7.1 Status

Completed 6/4/2004

4.7.2 Decription

A thread will be able to change its priority using a function (pSetPriority(int)). The lowest level of priority will be timesliced. Otherwise, all threads of the highest priority (0) will run to completion before any others.
API:

- `pSetPriority(int)`: Set current thread's priority. Spawned threads will run at their parents priority.
- `pGetPriorities()`: Return current thread's priority.
- `pLowestPriority()`: Lowest priority (timeslice queue).

- New config parameter, `_priority_count` in pSetup to set number of priorities. Default is 32.
- Optional second argument to spawn of a priority, e.g. `spawn(foo(),0);`
- Optional second argument to on block of a priority, e.g. `on(p1,0) { ... }`

4.7.3 Implementation

Array of thread queues. Scheduler will run high priority threads first. Timeslicing will only be turned on when running the lowest-priority threads.
To the user, 0 is the highest priority, but internally 0 represents the lowest value and thus what we timeslice on.
The scheduler calls `get_ready()`, which returns the next thread to run, respecting priorities.
The `preempt()` function calls `Processor::ts_okay()`, which returns false if we're in the kernel or we're in a non-timesliceable thread.

4.7.4 Regressions

pri1 - pri4.

4.8 Support For Multiple Processors

4.8.1 Status

Completed 6/4/2004

4.8.2 Description

Users will be able to instantiate a **Processor** object. A spawn pseudo-method will allow them to launch a thread on that processor. Using an on-block, e.g.:

```
par {
  on(<processor> [,<priority>]) { ... }
}
```

will allow for a similar feature using **par** blocks. Support for **pfor** will also be included.

4.8.3 Implementation

- Rename **Processor** to **Cluster**.
- A **Processor** object will be a handle around **Cluster**.

- A global variable will contain a pointer to the current **Cluster**. Most of the interface functions will use that value, except for some that take a cluster. A new interface function will return a **Cluster** object pointing to the current cluster.

- The **System** object will have a queue of clusters. Each cluster will make one pass through its threads, then pass to the next cluster.

- Add spawn pseudo method and add support for optional second parameter setting priority.

4.9 Regressions

proc1 - 3

4.10 Time Model

4.10.1 Status

Completed 6/15/2004

4.10.2 Description

For more information, refer to the twiki page. In short, users may call `pDelay(<n>)` to delay for **n** time units or call **pBusy(<n>)** to consume **n** time cycles. When a processor is busy, it does no other work, whereas a delay means that a process is just waiting.

4.10.3 Implementation

Refer to twiki page for the basic flow. In short, time is maintained within System. Two priority queues (stl priority queues) exist: One for delayed objects and one for busy objects. If an object called pDelay, it's added to the delay queue and if an object called pBusy, it's added to the busy queue. Note that to use pBusy, you must set `ConfigParms::_busyOkay` or else pBusy will not be allowed. This disables preemption- the only task switching will be done when calls to pDelay or pBusy are made.
Time model functions:

- pBusy(): Consumes time.

- pDelay(): Delays a thread.

- pTime(): Returns current time.

The delay queue stores Thread objects, ordered by decreasing time (smallest time is at the front). The time is the sum of the starting time and the delay size (both stored in the Thread).
The busy queue stores processors, also ordered by decreasing time. The time is the busy thread's start time + busy time. The busy thread is identified by finding the highest priority non-empty queue, then looking at the back. This is the case b/c the busy thread is added back to its respective priority queue by the pBusy routine.
At a given point in time, we cycle through all processors. For each processor, we execute all available jobs. When no more processors exist with jobs to run, we call `System::update_time()`. It looks at both queues and chooses a new time that is the smallest of the next items on the two queues. This becomes the new time. We then transfer all delayed threads which have the same time as current back to their owning processors and add those processors back to the ready queue. Duplication is handled by having `Cluster::add_proc()` only add a processor if its state is not "Running". We then add back all busy processors whose time has expired. Then we continue.
If a delayed thread is ready to run, but its processor is busy, we interrupt the busy if the thread has higher priority than the busying thread. We record how much busy time has been consumed and re-enqueue the processor. For the lowest priority threads, they are considered to be timesliced. A configuration parameter, `ConfigParms::_simtimeslice`, determines the timeslice amount. A thread of the lowest priority that is busy will actually add itself to the busy queue using the timeslice amount. The busy routine itself tracks the total amount of busy time required and loops, re-busying the thread until all time has expired. Thus, for timesliced threads and for interrupted threads, the routine sees that more time is required and loops as necessary.

4.10.4 Regressions

time1 - 4

4.11 Garbage Collection

4.11.1 Status

Completed 6/17/2004

4.11.2 Description

Plasma is going to have a lot of producer/consumer type code, where the ownership of a particular piece of memory will be hard to track. Garbage collection will make the code much easier to understand and less error-prone.

4.11.3 Implementation

Boehm garbage collector. The main issue is that the collector needs to know about all roots in the system, i.e. thread stacks. This is accomplished as follows:

- A list exists (`System::_active_list`) that records all active threads. When a thread is realized, it is added to this list. Each Thread object has a **nt** and **pt** pointer for storing this information. When a thread is destroyed, it is removed from the list.
- In addition to the bottom of the stack, each thread records the top of the stack. This is set whenever a thread is swapped out by calling `Thread::setStackEnd()`.
- When the collector is called, it called the function pointer GC_push_other_roots. This is set to the function `System::push_other_roots()`, which iterates over the active list, pushing information about the top and bottom of the stack.
- Other routines used are GC_lock(), which does nothing since we do not use kernel threads at this point, and GC_stop_world() and GC_start_world(), which turn preemtion off and on.

4.11.4 Dependencies

The main issue is getting it to handle user-threads. It handles kernel threads and should be able to handle user-threads, but I don't know how to do it yet.

4.11.5 Regressions

No explicit tests- the rest of the regressions should test its usage.

4.12 Timeouts

4.12.1 Status

Completed 6/18/2004

4.12.2 Description

Create a channel that has a backing thread which wakes up and writes to the channel after a specified amount of time. Use in alt blocks.

4.12.3 Implementation

Created the Timeout class- it's in Interface.h. It's written entirely in C++ so that I could hide the implementation and not have to worry about linking Plasma code in with the rest of the thread package.

4.12.4 Regressions

chan12

4.13 Clocked Channels

4.13.1 Status

Completed 8/1/2004

4.13.2 Description

Clocked channels will be channels with knowledge of time. Writes will be able to occur at anytime, but reads will only be allowed on clock boundaries, where the clock period is defined by a parameter to the constructor.

4.13.3 Implementation

Implemented as a normal channel, except that a read will create a waking thread if we're not on a clock edge, then go to sleep.

4.13.4 Regressions

clock1- clock3