Plasma: *P*arallel *LA*nguage for *S*ystem *M*odeling and *A*nalysis

Brian Kahne, Peter Wilson
Freescale Semiconductor
Aseem Gupta, Nikil Dutt
University of California, Irvine

The Problem

How do we quickly and efficiently model a complex SoC so that we can perform a trade-off analysis early in the design cycle?

- **The language must be easy to use, with a minimum of extraneous syntax.**
- **Must be able to easily express concurrency and help the user avoid common pitfalls.**
- **It must be possible to map different parts of the program to different aspects of the system, e.g. *"this thread runs on this processor"*.**

Future chips are likely to have many programmable cores:

- **How will users develop software for them?**
- **How will users map their complex application to an existing product?**

Proposal

These two problems are closely related:

- Both require the development of explicitly parallel programs.
- In both cases, users want a clean way to express this parallelism.

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 and parallel application development:

- Increase productivity through clearer representation.
- Increase productivity and quality through increased compile-time checking of the more difficult-to-get-right aspects of systems models (the concurrency)

If successful, the language will have wide applicability:

- Useful throughout the life cycle of a design, from initial product definition to software development for the design.

Existing Work

SystemC **attempts to handle modeling but does not truly understand parallelism: No help with software development. It is also very hardware-centric.**

SpecC **is a language with true parallelism, but is still very hardware centric.**

HandelC **makes it easy to create hardware using a C-like language, but restricts the language, thus making it not relevant for general software development.**

OpenMP **adds parallel extensions to C++ but is entirely software oriented: It only handles shared memory systems.**

The Plasma Language

Plasma is a set of extensions to C++.

The basis for Plasma is Occam:

- Based on CSP (Communicating Sequential Processes).
- Threads are explicitly created by the user and communicate via typed channels.
- Plasma adds support for shared variables protected by mutex code and garbage collection (Boehm collector).

Plasma adds a discrete-event simulation API useful for modeling hardware.

The language can be used in two ways:

- For applications development, it provides a convenient means for exploiting the parallelism in a multi-core, multi-threaded system.
- For modeling, the parallelism features are used to model the parallelism inherent in the design being modeled.

Thread Creation

To launch a thread:

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

Each is a thread

This executes when the above threads complete.

Or:

```
// This launches foo as a thread.
Result<int> res = spawn(foo(1,2,3));
// This causes us to wait until foo finishes.
cout << "The result is:  " << res.value() << endl;
```

A replicating form of *par* also exists: The *pfor* block.

Channels

The primary means for communicating between threads is through the use of typed channels.

Channels are simply C++ templates, so new types of channels can be added without changing the language. Plasma currently defines the following channels:

- Single item channel: A read blocks if the channel is empty and a write blocks if the channel has data.
- Queued channel: The queue may be fixed in size or infinite.
- Clocked channel: Writes may occur at any time, but reads are limited to clock boundaries.
- Time-out channel.
- Spawn-result adapter channel.

By default multiple threads may write to a channel, but only one thread may read from the channel.

This behaviour can be changed by the user.

Channels

Channels are simply C++ templates. To declare a channel for transmitting integers:

```
Channel<int> chan;
```

To send data on a channel:

```
chan.write(10);
```

To read data from a single channel:

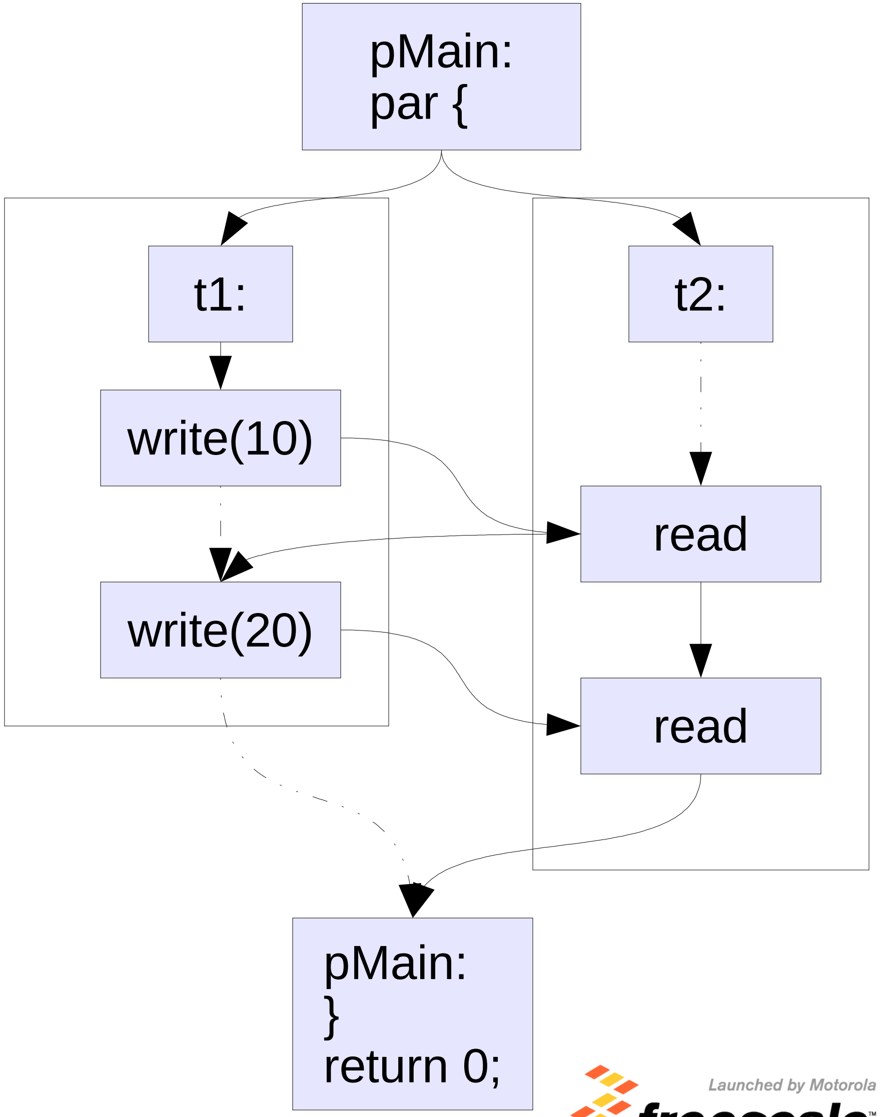
```
int x = chan.get();
```

Channel semantics:

- **Writing to a full channel causes the thread performing the write to block.**
- **Reading from an empty channel causes the thread performing the read to block.**
- **The result is implicit flow-control: Threads do not need to check the state of the channels, they simply perform their operations and will block if needed.**

Channels

```
int pMain(int argc,
          const char *argv[])
{
  Channel<int> c;
  par {
    t1(c);
    t2(c);
  }
  return 0;
}
void t2(Channel<int> &c) {
  c.write(10);
  c.write(20);
}
void t1(Channel<int> &c) {
  cout << c.read() << '\n';
  cout << c.read() << '\n';
}
```



Channels and the *Alt* Block

Plasma uses the *alt* block to allow a thread to read from more one than channel. Upon entry to an *alt* block:

- If only one channel is ready, that channel does a read.
- If more than one channel is ready, a non-deterministic selection is made.
- If no channels are ready, the thread blocks until one of the channels has data.

Prialt and *priafor* are similar to *alt*, except that they ensure the ordering specified by the user.

All of these blocks may be nested.

Channels and the *Alt* Block

To read from multiple channels:

```
alt {
      c1.port(int x) {
         cout << "Read " << x << " from c1.\n";
      }
      c2.port(int x) {
         cout << "Read " << x << " from c2.\n";
      }
   }
```

Data mapped to variable on a read.

Channels

Code executed when the read occurs.

A looping form for *alt* also exists:

```
afor (int i = 0; i != channels.size(); ++i) {
  channels[i].port(int x) { ... }
}
```

Processors

A *Processor* object exists to group together threads.

They allow the user to partition a design based upon hardware resources.

Threads can be directed to run on specific processors:

- **With par:**

```
par {
    on (processor1) { ... }
    on (processor2) { ... }
}
```

- **With spawn:**

```
processor1.spawn(foo(1,2,3));
```

Processors can be made to share their ready queue, enabling SMP modeling:

```
Processor p2 = make_sharedproc(processor1);
```

Priorities

Threads have priorities.

- The number of priorities can be set by the user.
- Higher priority threads execute before lower priority threads.
- Lowest-priority threads are time-sliced.

Changing priorities:

- A thread inherits its priority from its parent.
- The priority can be changed by an API call:

```
pSetPriority(0);
```

- The priority can be set at thread creation time:

```
par {
   on (pCurProc(),4) { ... }
   on (p2,5) { ... }
}
```

or

```
p1.spawn(foo(1,2,3),10);
```

Specifies the priority.

Shared Data Structures

To create a shared data structure:

```
pMutex class Foo { ... };
```

All public member functions, excluding constructors and destructors, will be wrapped with mutual-exclusion code.

Channels are implemented using *pMutex*.

Simulation Time

Plasma includes a discrete-event time model, implemented as an API.

Its presence is optional; support might not exist for a version optimized for software development.

Two important functions:

- **`pDelay(ptime_t)`: Thread is idle for specified amount of time.**
- **`pBusy(ptime_t)`: Processor is busy computing for the specified amount of time.**

Time only progresses by making calls to these functions.

The idea is that software written in Plasma can be annotated in order to understand performance.

Power Modeling

Power is modeled in a similar manner to time: Energy is stored on a per-processor basis.

- `pEnergy(energy_t)`: Adds energy to the processor.
- `pGetEnergy()`: Returns the processor's energy and clear the value.

To calculate power:

- Sample a processor's energy on a periodic basis.
- Divide the energy by the sample period.

Roadmap

Currently, we are conducting an experiment to see if this concept makes sense.

- **We have a front-end which parses Plasma and generates C++.**
- **It targets a simple user-mode threading library implemented with Quickthreads (all threads run in a single UNIX process).**

We are in the process of releasing Plasma as open-source.

Examples

Several examples have been developed in order to demonstrate Plasma's use:

- **An engine controller to demonstrate Plasma's use for embedded applications.**
- **A transaction processing example: Clients send database requests to a mainframe which might have to send requests to a disk array.**
- **A simple RISC pipeline. This demonstrates the use of clocked channels.**
- **A parallel C compiler: Code generation for functions happens in parallel.**
- **The 2-D Discrete Wavelet Transform (DWT) block used in the JPEG2000 and MPEG4 compression standards.**

Results

SystemC vs. Plasma:

- **The DWT block was originally modeled in SystemC, then converted to Plasma.**
- **The biggest advantages that we saw in using Plasma were:**
  - The ease with which threads can be created:
    - > No need to declare classes, mark methods as SC_THREAD, etc.
    - > Instead, threads can be launched as functions.
    - > SystemC 2.1 adds some of this capability with *sc_spawn*, but it still requires extra syntax for declaring a class and passing parameters.
  - Communication:
    - > The Plasma code was shorter due to the simpler flow-control mechanisms provided by Plasma's channels.
    - > Plasma's simple channels, since they are templated, were used for a wide variety of communication tasks.
- **Of course, this is all qualitative: Plasma is in too early of a stage of development to do a fair comparison on performance.**

Conclusion

Models of complex systems and multi-threaded software both share a common need: An easy way to express explicit parallelism.

Our proposal is the *Plasma* language:

- **Superset of C++.**
- **Parallelism based upon CSP.**
- **Inherent parallelism provides for the potential to optimize context switches, catch common parallel-programming mistakes, etc.**

The language may be used for both application development and for modeling:

- **The language provides a minimal set of parallel concepts.**
- **A library provides various types of channels for communication and a discrete-event simulation API.**
- **This allows a problem to first be modeled as a multi-threaded software application, then be decomposed into a hardware/software systems model.**