An Introduction To The Plasma Language

Brian Kahne †, Aseem Gupta ‡, Peter Wilson †, Nikil Dutt ‡

† Freescale Semiconductor Inc
Austin, TX 78729, USA
{bkahne, Peter.Wilson}@freescale.com

‡ Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697, USA
{aseemg, dutt}@uci.edu

Abstract

The ability to enhance single-thread performance, such as by increasing clock frequency, is reaching a point of diminishing returns: power is becoming a dominating factor and limiting scalability. Adding additional cores is a scalable way to increase performance, but it requires that system designers have a method for developing multi-threaded applications.

Plasma, (**P**arallel **LA**nguage for **S**ystem **M**odeling and **A**nalysis) is a parallel language for system modeling and multi-threaded 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 goal of the Plasma project is to investigate whether a language with the appropriate constructs might be used to ease the task of developing highly multi-threaded software. In addition, through the inclusion of a discrete-event simulation API, we seek to simplify the task of system modeling and increase productivity through clearer representation and increased compile-time checking of the more difficult-to-get-right aspects of systems models (the concurrency).

The result is a single language which allows users to develop a parallel application and then to model it within the context of a system, allowing for hardware-software partitioning and various other early tradeoff analyses. We believe that this language offers a simpler and more concise syntax than other offerings and can be targeted at a large range of potential architectures,including heterogeneous systems and those without shared memory.

1 Introduction

We initiated the Plasma experiment within Freescale after recognising the confluence of two separate trends: the rise of multi-core systems

(denoting a need for languages and tools to program them efficiently) and the increasing complexity and importance of software in SoC solutions (making the ability to model the software at least as important as the ability to model the hardware).

Plasma, (**P**arallel **L**anguage for **A**rchitecture, **S**ystems, **M**odeling and **A**nalysis) is a parallel language for system modeling and multi-threaded application development implemented as a superset of C++. The language extensions are based upon those found in Occam[1], which itself was based upon CSP[2] (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.

Just as there are higher-level languages in the sequential or functional domain than C/C++, we are happy to believe in the future existence of languages attacking the problems of concurrency at a higher level than Plasma. However, we believe that such languages must be built upon some appropriate abstractions, and that a language making those abstractions explicit, as Plasma does, is highly useful.

2 Motivation

The goal of the Plasma project is to investigate two major areas:

- 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). A key constraint for this usage is to present a language "feel" which is natural for processor micro architects and SoC architects; in general, while they are comfortable with C (and Verilog) they do not take kindly to the complexities of the C++ language and the necessary associated class libraries.
- To provide a basis for real systems implementation, including facilities important for embedded work including fault tolerance and real-time deadline specification. 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 have a standardized method for developing multi-threaded applications. A Plasma compiler would allow designers to develop such programs and gain the benefit of a single clear simple model of concurrency, sharing and communication (as opposed to choosing between a plethora of APIs and libraries), and also to gain the

benefits of a compiler's ability to detect misuse of the language (such as unprotected sharing of data).

A full-blown Plasma system would thus have wide applicability: It could be used throughout the life cycle of a design: in the initial stages, it can model proposed products and perform early architectural and microarchitectural design trade-offs; to create refinable models of software; to allow co-design to identify which portions of the behavior should be implemented as software and which as application-specific acceleration hardware; to partition and balance the software across the right number of execution units; and finally to build an implementation with the software being compiled and mapped appropriately to processors and the acceleration behaviors being directly synthesized into hardware.

The current implementation is limited in scope, being intended to allow the language to be evaluated for its intended use rather than aiming at real system work. The intent is to release the implementation under an appropriate open-source license in the near future.

3 Existing Work

There exists 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 SystemVerilog[3], does allow the user to raise the level of abstraction, but it still remains a language primarily 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[4], HandelC[5], and SystemC[6]. The first two are languages with true parallelism: Their compilers 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 semantic[1] 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

[1] There are clarity problems, too, which are dealt with later.

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.

4 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[7][8], a fully-conservative collector. The interface allows the user to work with explicitly-managed memory or with garbage-collector managed memory, thus allowing for interoperability with existing C++ code.

In a production 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 [9].

4.1 Parallelism

Plasma inherits its parallelism constructs from Occam and extends them with a functional operator, similar to the *future* found in Multilisp[10]. 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, *Max* threads are launched concurrently, each executing the body of the *pfor* block, the function *foo()*.

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.

4.2 Communications

4.2.1 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.
  writer(1);
  writer(2);
  writer(3);
  writer(-1);
 }
 {
  // Thread 2.
  int x;
  do {
   x = c.get();
   imprint ("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 *imprint*, 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 a 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) {
  mprintf ("From port c0: %d\n",v);
 }
 c1.port (int v) {
  mprintf ("From port c1: %d\n",v);
```

```
 }
}
```

If none of the channels (*c0* or *c1*) 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 != c.size(); ++i) {
 channels[i].port (int v) {
  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. This allows the user, for example, to wait on multiple containers of channels or individual channels combined with containers of channels, etc.

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 "textual" order in the same way as *else if* clauses are handled in a normal C conditional.

4.2.2 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 its implementation 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.

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. Optionally, the size may be set to zero, which means that a write will block until a read has removed the data. This is useful for modeling a fully interlocked pipeline.

4.2.3 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*.

4.3 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 simulation 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.

Decorating software in this way is both tedious and error-prone; we therefore envisage production-quality simulation tools being able to examine and annotate the internal representation of a plasma program intended for simulation using target architecture and microarchitecture information to provide an appropriate density of automatically-generated annotations.

Threads may be distributed to multiple "processors" in order to model multiple hardware resources in a simulation. As in the real world, in the Plasma simulation universe a thread on one processor might be busy (that is, using up processor resources), but this does not affect a thread on another processor:

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

The *on* construct provided by Plasma is a novelty of the language, as it specifically allows simulation and application development for multiprocessor systems. Other languages do not provide explicit support for multiprocessor systems and special code needs to be written [11].

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 both distributed and SMP systems (and mixes) to be easily modeled.

Each thread may be given a priority. Priority-based preemptive scheduling is provided: higher

priority threads (a lower priority number) execute until completion before ready lower priority threads are run. The lowest priority threads are time-sliced. A delayed high-priority thread may interrupt a lower-priority thread which is busy.

4.4 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 coarse-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. As with the modeling of processor (or resource) utilization mentioned above, we envisage the toolchain making use of some architecture-specific power information to decorate the code automatically.

5 Implementation

Our pilot implementation of Plasma's front-end parser makes use of OpenC++[12] an open-source C++ grammar and meta-object protocol. The Plasma front-end processing code is linked in with the OpenC++ framework to form the Plasma executable.

From the user's point of view, using the Plasma program is identical to using a normal C++ compiler: You can produce an object file, an executable, or link together multiple object files to produce an executable. The transformation of the Plasma primitives is generally fairly straightforward. Unfortunately, C++'s lack of orthogonality sometimes complicates matters. For example, the *par* block's contents are moved into individual functions. These are then launched as independent threads using the *pSpawn* primitive. However, since C++ does not allow nested threads, each function must be placed at the global level and any variables used by the code must be passed as references.

The Plasma language is (optionally) garbage collected. This is accomplished by using the Boehm Weiser Garbage Collector[7][8], a conservative garbage collector. In order for it to work with the thread library, a hook was added so that the stacks of all threads could be searched during a collection.

The Plasma threading library is a simple user-mode threading library based upon QuickThread[13]. All threads in this implementation of Plasma run within a single OS process. The QuickThreads library only implements thread-switching primitives. The scheduler, various queues, etc., are implemented within the Plasma library. The thread library generally operates in either of two modes, defined by whether calls to *pBusy* are allowed. If they are not allowed, then preemption is enabled: An alarm is set using a UNIX *signal*. The signal handler calls the scheduler which schedules a new thread to be run.

If calls to *pBusy* are allowed, then preemption is disabled and all thread switching is done cooperatively, either explicitly via calls to functions such as *pBusy*, or implicitly, such as by reading from an empty channel. In addition, calls to *pDelay* mean that the current thread is added to the delay list; it will only execute again once time has advanced sufficiently. A call to *pBusy* means that the entire *Proc* object is removed from the list of available processors and added to the busy queue; it will only be moved back to the queue once time has advanced sufficiently.

6 System Modeling

In order to evaluate Plasma's effectiveness for system modeling, we have modelled a 2-D Discrete Wavelet Transform (DWT) block, used in the JPEG2000 and MPEG4 compression standards. The multi-resolution representation derived from DWT time-frequency decomposition demonstrates extraordinary advantages in signal analysis and compression. The full frame nature of DWT decorrelates the image over a large scale and eliminates blocking artifacts at high compression ratios which is a drawback in block-based transformation standards. We compared the implementation of DWT in Plasma with another implementation of DWT in SystemC.

SystemC in its current version has two kind of processes: SC_METHOD and SC_THREAD. Each process must have ports in order to interface with any other process. An SC_METHOD process has a sensitivity list associated with it and whenever an event occurs on a signal or a port in the sensitivity list, the process executes. Such a process completes execution in the same time step (with no delays or waits) and returns control back to the simulation kernel. Thus, an SC_METHOD process cannot be suspended or contain an infinite loop. A process of type SC_THREAD can be suspended and then made to resume execution based on a time delay or on occurrence of a certain event; the events may be described in a sensitivity list and the *wait()* method is used to await their occurrence. By default all SystemC threads are executed in parallel, which in some sense is ironic, given that the C++ compiler does not understand parallelism well and does no optimizations. SystemC does not explicitly support sequential execution. The execution sequences of threads are determined by a signal-trigger mechanism, which means that in order to write a sequential SystemC model the designer must add trigger signals and *wait()* statements for each thread. For example, in the following piece of code, in order to make *b()* execute after *a()*, designers must declare an event *a_done* using sc_event, which is triggered in the last statement of the thread *a()* by using *notify()* method and add a statement in the thread *b()* which makes the execution of the thread wait an *a_done:*

```
sc_event a_done;
a(){
  ...
  a_done.notify();
}
b(){
  wait(a_done);
  ...
}
```

In Plasma, however, threads execute sequentially by default and can be easily made to run concurrently using the *par* statement. In the following code, *a()* finishes execution, followed by parallel execution of *b()* and *c(),* followed by sequential execution of *d():*

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

The clumsiness of composing a sequence of activities in System C we attribute to its goal of describing just hardware systems, in which all elements always execute concurrently; this, however, sharply reduces the usability of the language for describing software. SystemC version 2.1 improves on the situation slightly; it provides *sc_spawn*, which is used to create a dynamic process instance, but this is also rather verbose and still requires threads to be methods of classes.

Another feature of Plasma is the powerful handling of communication requirements between threads by channels. In our experience, communication was less error-prone in Plasma than in SystemC. This observation is partly due to the fact that the code requirements for the communication primitives were a lot more verbose in SystemC. SystemC needs three constructs to describe communication between modules: Ports, Interfaces and Channels. A module has ports through which it communicates with other modules. There are three kinds of port defined by *sc_port:* input, output and inout [14]. An interface declares a set of methods for accessing a channel and is implemented by the channel, hence the definition of the methods declared in the interface are described in the channel. An instance of a port connects a module to a channel, enabling reads from and writes to the channel. A port can be associated with an arbitrary number of access methods; the set of access methods is defined by the interface.

Figure 1 shows module *mod_a* has a port *port_b,* which can only access methods *meth_d()* and *meth_e()*, which are specified in the interface *intf_c*. The channel *chan_f* contains the declarations of the methods *meth_d()* and *meth_e().* The module *mod_a* is able to communicate with other modules using channel *chan_f* through an interface *intf_c,* which has the methods *meth_d()* and *meth_e()*, for enabling communication over the channel. In general, a port is declared using *sc_port* of a specific interface:

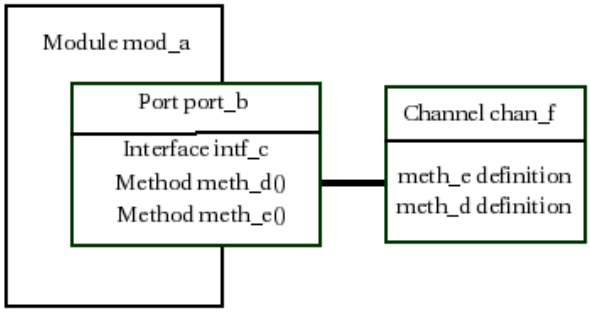

Figure 1: Ports, Channels, and Interfaces in SystemC

```
sc_port<interface_name> port_name;
```

port_name is a port, which exhibits the interface behavior as specified by the *interface_name.* Thus, the port can access the methods specified by the interface and consequently can access the channel that is associated with that interface.

An interface is an abstract base class of C++ that inherits from the class *sc_interface.* Interface, along with the set of access methods is specified as follows:

```
class interface_name:
  virtual public sc_interface
```

```
{
  public:
  virtual return_type
   method_name(parameter_list) = 0;
  ...
};
```

A channel is an object that serves as a container for communication and synchronization. A channel can implement one or more interfaces and has the definitions of the methods. A channel in SystemC is defined as:

```
class channel_name:
  public sc_module,
  public interface_names
{
 public:
   virtual return_type
   method_name(..) {
     ...
   }
   ...
};
```

The above described module will be declared in SystemC as:

```
class intf_c:
  virtual public sc_interface
{
  public:
    virtual int meth_d(char) = 0;
    virtual int meth_e() = 0;
};
class chan_f:
  public sc_module,
  public intf_c
{
  public:
    virtual int meth_d(char)
    {
      ...
    }
    virtual int meth_e()
    {
      ...
    }
  :
};
SC_MODULE (mod_a){
  sc_port <intf_c<int> > port_b;
  ...
}
```

The verbosity and complexity of SystemC as seen here, in order to describe the communication between a channel and a module makes this task hard on the designer. Note that, the above exercise was just to set up the communication primitive and declarations, and does not involve statements about any communication transaction. On the other hand, Plasma has taken care of this problem by providing much simpler constructs. As seen in Section 4.2.1, channels can be instantiated in one statement and can simply be used by any thread. Channels can be conveniently set to transfer any custom data structure, without affecting the channel's interface and provide a simple user interface of *write(), read(), get(), ready(), full().* It is up to the channel to make sure that these operations are safe and to ensure proper flow control. For example, let us suppose two threads have to communicate using a complex data structure called *packet,* defined using *struct.* Then a channel, *ChanPack,* which can be used for transferring data of type *packet,* is declared in a single statement:

```
typedef Channel<packet> ChanPack;
```

The threads need to instantiate channels of type *ChanPack* and use the interface. For example:

```
void Producer_thread (ChanPack &CP)
{
  packet p1;
  CP.write(p1);
}
```

Plasma already comes with channels with predefined behaviors such as Busy-Channel, Queue-Channel, A Timeout Channel, Clocked-Channel. Busy-Channel is similar to a regular channel but 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 zero to mean no timeslicing. Queued channels allow for multiple producers and supports multiple consumers also. The queue size is not fixed by default, but the user may set a maximum size by specifying it in the constructor. Timeout channel is used to break out of an *alt* block after a specified amount of simulation time. In Plasma, the reads block on empty channels and writes block on full channels and it is up to the channel to make sure that these operations are safe and to ensure proper flow control. Unlike SystemC, the designer is free from the burden of implementing any flow control mechanisms for channels.

Plasma's conciseness reduces the number of errors compared to SystemC. Designers can swiftly explore the optimal distribution of an application among multiple threads without having to spend too much effort on getting the communication primitives right.

Plasma has also been used in several diverse areas of modeling and application development. It has been used to model clocked hardware, more specifically a very simple RISC pipeline. Clocked channels were used and no stage in the pipeline needs to know about time or the clock, which is now completely encapsulated in the clocked channels. The timing of the whole system can thus be modified simply by changing the declarations of the clocked channels. Plasma was also used to model a large scale networked system of a database server, where a series of terminals send requests to a 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. Plasma functions such as *pBusy* and *Random* were used. Plasma was also tested for modeling real-time embedded systems by implementing a simple engine controller. 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 has several cylinders and based upon the amount of fuel added, the model calculates the new speed of the engine. A separate thread is launched for each cylinder and the *pBusy* function is used to consume time in order to understand the load on the micro-controller. Thus, if the load were too great for a single core device, the threads could be distributed over multiple cores by the using the *on* construct.

7 Conclusion

Growing system complexity and unmanageable power densities have made multi-core processors a popular choice for System on Chips. However, the absence of convenient languages for design and development of multi-threaded application has been a limiting factor. We introduce

a new language, Plasma **(P**arallel **La**nguage for **S**ystem **M**odeling and **A**nalysis**),** in an attempt to provide a parallel language for system modeling and multi-threaded application development. In this work, we have motivated the need for another modeling language based on the unsuitability of other contemporary languages. Plasma is a superset of C++ with parallel constructs based upon CSP. The fact that the language is inherently parallel provides for the potential to optimize context switches and perform safety checks to guard against common concurrency problems at compile time. We also discussed the various communication primitives provided by the language and compared it against SystemC for clarity. Plasma offers explicit support fort multi-processor simulation by featuring special constructs, which allow allocation of threads to specific processors. In this paper, we also briefly presented the power modeling capabilities of Plasma at the system level. Plasma will soon be publicly released as an open source language.

References

[1] C. Hoare, *Occam 2 Reference Manual*. Prentice Hall, 1988.

[2] C. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.

[3] Accellera, "SystemVerilog Language Reference Manual 3.1a."

[4] D. Gajski, J. Zhu, R. Doemer, and A. Gerstlauer, *SpecC: Specification Language and Methodology*. Springer, 2000.

[5] M. Bowen, *Handel-C Language Reference Manual*. Embedded Solutions, Ltd., 1999.

[6] T. Grotker, S. Liao, G. Martin, and S. Swan, *System Design With SystemC*. Kluwer Academic Publishers, 2002.

[7] H. Boehm, "Space Efficient Conservative Garbage Collection," *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pp. 197–206, 1993.

[8] H. Boehm and M. Weiser, "Garbage Collection In An Uncooperative Environment," in *Software Practice and Experience*, vol. 18, pp. 807–820, 1988.

[9] B. Werther and D. Conway, "A Modest Proposal: C++ Resyntaxed," *ACM SIGPLAN*, vol. 31, no. 11, pp. 74–82, 1996.

[10] R. Halstead, " MULTILISP: a language for concurrent symbolic computation," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 4, 1985.

[11] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino, "SystemC Cosimuation and Emulation of MultiProcessor SoC Designs," *IEEE Computer*, 2003.

[12] S. Chiba, "A Metaobject Protocol for C++," *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 285–299, 1995.

[13] D. Keppel, *Tools And Techniques For Building Fast Portable Threads Packages*, 1993.

[14] J. Bhasker, *A SystemC Primer*. Galaxy Publishing, 2002.