Designing with SystemC: Multi-Paradigm Modeling and Simulation Performance Evaluation

L. Charest, E.M. Aboulhamid and A. Tsikhanovich
DIRO, Université de Montréal
2920 Ch. de la TourCP6128 Centre-Ville
Montréal, Qc, Canada H3C 3J7
Phone: +1(514)343-6822, FAX: +1(514)343-5834
{chareslu, aboulham, tsikhana}@iro.umontreal.ca

Abstract

The objective of this paper is to demonstrate the benefits of the multi-paradigm design methodology in hardware or hardware/software design, and to show that the advantages to use it outweigh the risks.

We will show that with this methodology we will be able to develop more resilient specifications and models, express different behaviors and views of the same "entity", facilitating design exploration, ease of specification and libraries development.

We focus on the multiple paradigms of SystemC to implement typical hardware concepts. This work may be considered as a design methodology for RTL and architectural models using SystemC. We conclude the paper by a performance evaluation of SystemC simulation engine on a multiprocessor case study.

1. Introduction

SystemC is a modeling platform consisting of C++ class libraries and a simulation kernel for design at the system-behavioral and register-transfer-levels. Designers create models using SystemC and standard ANSI C++[1]. Different models of computation and design methodologies may be used in conjunction with SystemC. The design libraries and models needed to support these specific design methodologies are considered to be separate from the SystemC core language standard. This work may be considered as a design methodology for RTL and architectural models. By its nature, SystemC inherits the capabilities of C++, like the support of multiple paradigms: classes, overloaded functions, templates, modules, ordinary procedural programming, and others. The freedom afforded by these capabilities may also bring new difficulties to the understanding of the model or the ability to synthesize it. The objective of this paper is to demonstrate the benefits of this multi-paradigm environment, and show that the advantages outweigh the risks. We will show that we will be able to develop more resilient specifications and models, express different behaviors and view of the same "entity", facilitating design exploration and ease specification and development of libraries.

We will illustrate this by comparing SystemC capabilities with those of the VHDL hardware description language. The lack of some useful paradigms in VHDL has already been described as early as 1991 as summarized in [2]. Early attempts tried to augment VHDL with other constructs, but since systems have more and more software programmable components, many designers are leaning toward a common language for hardware and software modeling. In order to make a fair comparison, we intentionally restrict ourselves to hardware modeling both at the functional and RTL level. We will mention only in passing the system design capabilities of SystemC 2.0 (like Channels and interfaces). System-level capabilities using C++ and/or SystemC are well described elsewhere [3, 4]. This may serve also as guideline for VHDL designers to understand and potentially adopt the SystemC methodology. In our comparison, we will use the concept of *commonality* and *variation* developed by Coplien [5, 6] to compare the two environments. Multi-paradigm design, defined in [5, 6] is a specific approach to domain engineering that builds on a collection of paradigms supported by some programming languages. This methodology is based on an application domain analysis – definition of the commonality and variation for the components of a model, and a solution domain analysis – a commonality and variation matching to the implementation technology structures. In digital design the commonalities and variation are expressed by means of the HDL constructs.

In the next sections we will consider how SystemC multi-paradigm design applied to hardware or hardware/software modeling can help hardware designers to increase design reuse and facilitate a development of hardware libraries and executable specifications.

The section 2 succinctly describes the support of commonality and variation in VHDL. Section 3 brings some solutions using C++ paradigms to build hardware libraries. Section 4 highlights the relations between the commonality and variation concept and some interesting SystemC/C++ constructs. Section 5 draws a sketch of ways of combining multiple paradigms together. Section 6 presents a concrete SystemC test bench built using some paradigms presented in this article. Simulation performance evaluation of SystemC is given in section 7 and finally, section 8 concludes this work.

2. Commonality and Variation in VHDL

VHDL allows the expression of commonality by what is called design units. A design unit is a VHDL construct that may be independently analyzed and inserted in a design library. These design units are:

- **Entity declaration**: describes the interface view of a component (like a Data Book description). It is implementation independent.
- **Architecture body**: describes an implementation of an entity (like a single schematic diagram).A single interface may have alternative architectures.
- **Package (declaration and body)**: contains information common to many design units. This information consists of functions, types, signals, and constants. It hides details, simplifies design, and may invoke other packages.
- **Configuration**: relates local entity and architecture references to actual units in libraries (like a parts reference list).

Commonality can also be expressed by generate statements and generic constants for regular structures, like a ripple carry adder or an interconnection network.

Variation is obtained by configuration, which allows the designer to choose architecture among many others during design space exploration. Variation can also be obtained by giving a specific value to a generic parameter, or by overloading functions and subprograms in coordination with packages. Overloading allows the reuse of models even if the basic data types are changed, like going from a bit type to a 9-valued standard logic. Commonality between processes is very limited, except if we choose very complex ways like concurrent procedure calls rendering models quite cryptic.

3. Design Reuse and Hardware Libraries using SystemC

As seen before, the main mechanism for design reuse in VHDL is libraries of design units. Structural hierarchy is the way to reuse components declared in a library. SystemC has the same capability as VHDL and also other mechanisms, based especially on inheritance, templates and overloading.

3.1. Module Inheritance in SystemC

In designing a library of gates, latches... we can specify all the properties related to gates in a module `Gate` from which, `And`, `Or`, `Xor`... will be derived later. The purpose of the `Gate` module is to act as an interface. All the properties and methods for Flip-Flops will be captured in a similar module (Setup and hold time, delays, etc.). This is an example of simple inheritance of a module:

```
class Gate : public sc_module
{
 public :
 Gate(const sc_module_name& name)
   :sc_module(name)
 { (...) }
};

class And : public Gate {
public :
  SC_HAS_PROCESS(And);

  And(const sc_module_name& name) : Gate(name)
  { (...) }
};
```

3.2. Using Inheritance to Insert Tags or Attributes

In VHDL, we can reuse a component with a variation of behavior or semantics based on attributes. For example, we can use attributes to specify that a process is at the RTL or behavioral level. However, it is impossible to derive processes from previous ones, or specialize their methods. In modern software-oriented methodology, recommendations are made supporting a common root for all objects in the software design; `sc_object` constitutes such a root in the core of the SystemC library. For the purpose of modeling a library of components and IPs, `sc_module` can be the building block for the models. From this, we may derive other specialized modules, as described in the Figure 1.

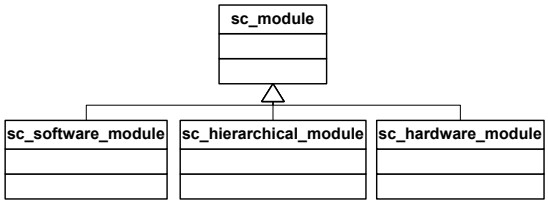

Figure 1: Module specialization

Then these basic building blocks can be used to construct our libraries. With this kind of construct, we can specify how the module should be instantiated. Because all three modules inherit from the same `sc_module`, they have the same behavior (unless we choose otherwise). A designer could decide that a

particular software module should be implemented in hardware, by simply changing the ancestor of his design. The kind of construction could be recognized by synthesis analyzers without having to alter the SystemC model. However, as it is part of the static configuration, the drawback of this approach is the necessity of recompiling the code, when a configuration modification is required.

3.3. Hierarchical Module Construction for SystemC Hardware Libraries

Class hierarchies can be a great help when designing hardware libraries. Here are some proven advantages:

- **Modularity of structure**: construction of hardware properties are reused in derived classes.
- **Locality of code**: modifications may be limited to a class or its ancestor(s) without spreading all over the model.
- **Reusability of code**: software implementation in base classes may be reused in derived class.

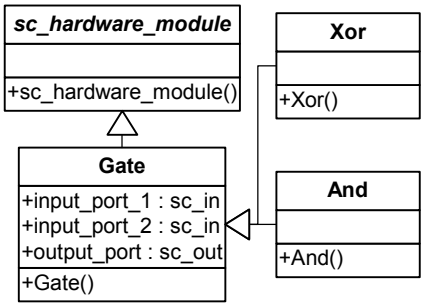

Figure 2: Behavioral hierarchy

There is no need in the "child" class to declare again the port information (this is not possible in VHDL). All we need to do is to implement a sensitive method to these ports and have these methods be set sensitive in each of the constructors.

```
And::And(...) : public Gate(...)
{
  SC_METHOD(main_process);
  sensitive << input_port_1 << input_port_2;
}
```

This duplication of definition of the sensitivity list, illustrated also in **Figure 3**, can be avoided; if the virtual init() method defined in the Processor class is sensitive to some signals, then the derived init methods in classes such as `DLX_Processor` will inherit this sensitivity. Therefore, adding and refining the hardware components could result in the construction of a library from ground up.

4. Communality, Variation and Configurations

4.1. Multiple Architectures

Using VHDL, we can have multiple architectures related to the same entity. We can consider the entity as the abstract class and different architecture as the derived classes from the basic abstract class. Note that in VHDL, this derivation process is limited to two levels of abstractions. In SystemC, this refinement process can continue indefinitely as illustrated in **Figure 3**.

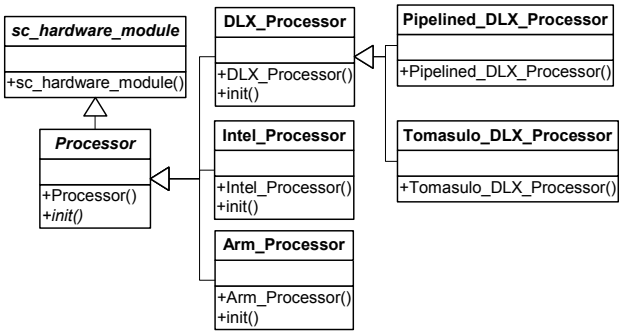

Figure 3: Behavioral refinement

4.2. Generation and Configuration of Regular Structures

In VHDL, generate statements are used to model regular structures composed of processes or components. In SystemC, a simple declaration of an array of components is sufficient, as illustrated below. Configuration is obtained in a very natural way by instantiation.

```
Processor *processor_array[10];
Processor_array[0]= new Arm_Processor();
processor_array[1]= new DLX_Processor();
processor_array[2]= new Intel_Processor();
```

The configuration could be put in a file and then read at execution time during elaboration. A switch statement could then be used to instantiate the different processors. This has no correspondence in VHDL:

```
int i = 0;
FILE input = fopen("cpu.cnf", "rt");
while (!eof(input)) {
char buffer[500];
buffer = fgets(input);
switch(atoi(buffer))  {
  case ARM :
   Processor_array[i++]=new Arm_Processor();
  break;
  case INTEL :
   Processor_array[i++]=new Intel_Processor();
  break;
  (…)
  }
}
```

4.3. C++ Polymorphism in SystemC Library

If we reexamine the UML behavioral refinement of **Figure 3**, we note that the abstract class Processor has a pure virtual method `init()`. This method is defined in the next level of abstraction in the derived classes `DLX_Processor`, `Intel_processor` and `Arm_processor`. If general enough, these methods are reused in the third level of refinement without having to redefine them. Modularity of code is very important to achieve reusability. A module, which uses these processors, can be written using only the abstract class processor. Depending on the true instance of the processor the right init method will be invoked by polymorphism:

```
for (int i = 0; i < 10; i++)
  //initialize all processors
  processor[i]->init();
```

If the same effect is desired in VHDL, we would need a special signal `Reset_Processor`, which should feed only processor components, is required. This is less elegant and less readable. It is also more time consuming during simulation because events on signals put a very heavy burden on the VHDL simulator.

4.4. Using Overloading Mechanism to Change the Behavior

In VHDL, overloading is limited to functions and procedures. In SystemC, not only methods but also constructors of classes can be overloaded, allowing more dynamic configuration of threads and modules. To determine the correct method to call, the compiler only looks at the type of the parameter(s) when the method call is issued. Here is an example of overloaded constructor for a `Processor` object:

```
Processor::Processor(const sc_module_name &name,
                     int bus_format)
  : sc_hardware_module(name)
{
  bus = new sc_in<int>();
  (...)
}
Processor::Processor(const sc_module_name &name,
                     char bus_format)
  : sc_hardware_module(name)
{
  bus = new sc_in<char>();
  (...)
}
```

4.5. Using Overloading to Speedup Simulation or Specify a Particular Behaviour

In VHDL, specification of a generic entity with *n* ports can be achieved by defining a generic and then declaring an array of *n* ports. However, if for a specific number of ports we have a particular behavior or any optimization, the only way is to test the parameter inside the architecture related to the generic entity.

In SystemC, by using overloading we can perform this in a more efficient way as illustrated below:

```
void And::compute(void); {
  output_port = input_port_1 &&
                input_port_2;
}
void And::compute(int n_ports); {
  if (n_ports != 0)
    for (int i = 0; i < n_ports; i++)
      (...)
}
```

4.6. Using Templates to Describe Regular Behavior and its Elaboration at Compilation Time

Interconnection networks or other behaviors can be easily described recursively. Using templates, we can have very abstract descriptions translated in an iterative behavior at compile time, hiding these abstractions from the simulator, and potentially the synthesis tool.

```
template<int n_ports>
class Gate : public sc_module
{
  public :
  sc_in<bool> input_ports[n_ports];
  sc_out<bool> output_port;
  (...)
};
//this is the recursive generic part
template<int N>
class Compute
{
  public :
  static inline void compute(bool &result,
                  sc_in<bool> *input_ports)
  {
  Compute<N-1>::compute(result, input_ports);
  result = result & input_ports[N-1];
  }
};
  //this is to end the recursive construction,
  //"overload" the generic part
class Compute<2> {
  public:
  static inline void compute(bool &result,
                  sc_in<bool> *input_ports)
  {
    result = input_ports[0] & input_ports[1];
  }
};
template<int n_ports>
class And : public Gate<n_ports> {
  public :
  SC_HAS_PROCESS(And);
  void compute_process(void)  {
    bool result;
    while(1) {
    //this is the call that will be
    //statically resolved by the compiler
      Compute<n_ports>::compute(result,
                                input_ports);
      output_port = result;
      wait();
    }
  }
```

```
  And(const sc_module_name& name)
    : Gate<n_ports>(name)
  {
    SC_THREAD(compute_process);
    for (int i = 0; i != nb_ports; i ++)
      sensitive << input_ports[i];
  }
};
```

Templates are resolved at compilation time. The instantiation of a specific And gate can be done either by:

```
#define FOO 10
new And<FOO>("simple_And");
```

or by:

```
const int foo = 10;
```

new And<foo>("simple_And");

The compilation of the following code:

```
while(1)
{
 Compute<n_ports>::compute(result, input_ports);
 output_port = result;
 wait();
}
```

would result in the following iterative hardware structure:

```
while(1)
{
  result = input_ports[0] & input_ports[1];
  result = result & input_ports[2];
  result = result & input_ports[3];
  (...)
  result = result & input_ports[9];
  output_port = result;
  wait();
}
```

This is because the compiler unrolls the template inlined method calls. This allows the compiler to perform very useful optimizations. As mentioned in section 2.2.2, due to the use of a static configuration (like in VHDL), the drawback of this approach is the necessity of recompiling the code, when a configuration modification is required.

5. Combining Mechanisms

The highest configuration flexibility is achieved by combining many paradigm mechanisms together. In **Figure 4**, we used the multiplexor as a "container" of other modules (gates). In the following we describe two modeling alternatives.

A first way would be to have base class to describe a `hardware_module`. Child classes are then all `hardware_modules` by definition. The problem using this approach is that the multiplexor ability to contain other module is not well highlighted and isolated from a standard basic hardware. One might like to create a new type of module and to be able, with the help of this new class, to distinguish a hardware module from a software module and from a hierarchical module. **Figure 1** illustrates partly this solution.

Using overloading and polymorphism, the behavior of the component can be changed. In the hierarchy represented by **Figure 4**, the `Gate` may be derived from hardware_module but have the multiplexor derived from `hierarchical_module`. The process in the `Gate` base class is the `process()` method which is abstracted but redefined by the children (similar to the `init()` method of **Figure 3**). Each gate is then responsible of their implementation of the process.

If we want to push further this methodology, we can have a `behavioral_process()`in the base class that would suggest that the derived module could have such a behavioral process that would simulate the module from a behavioral perspective, while `structural_process()` would be the one that should represent the module by composition from other modules. Because the initialization phase might be different, `init_behavioral()` and `init_structural()` methods would be provided, and the general `init()` method would be responsible for calling the appropriate initialization method. The constructor of the `hierarchical_module` is the one that builds the module and chooses whether the behavioral or the structural definition of the derived class would be used.

When the hierarchical module is composed of only one module (might even be decided or changed at run time during initialization phase), we could supply a method `has_subcomponent()` to determine the real nature of the module. This might hide the Boolean value that would be adjusted by the constructor at compile time. The list of sub-modules would be adjusted to allow an easy traversal of the hierarchy.

Another approach which would be appropriate uses a design patterns approach [7].

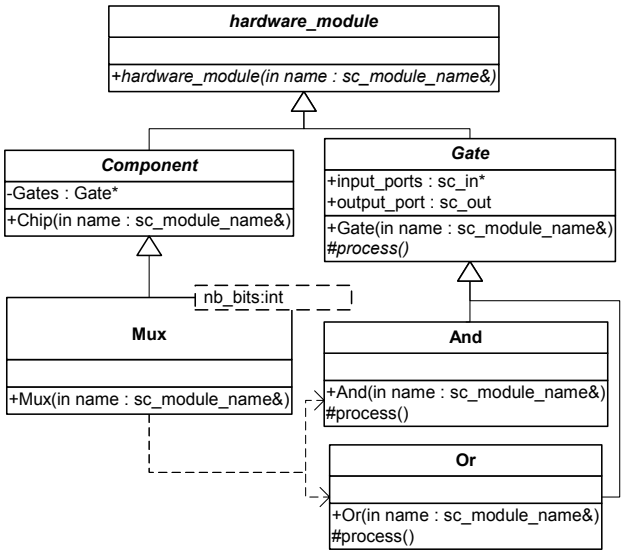

Figure 4: A multi-paradigm module description

6. Applying the SystemC multi-paradigm modeling methodology

6.1. Pipelined DLX multiprocessor

In the following, the use of the SystemC multi-paradigm modeling methodology is shown in terms of an example of pipelined DLX multiprocessor model. In this model we have used only some of the SystemC paradigms described above.

The pipelined DLX processors are connected through a unidirectional ring, where every pair of adjacent nodes can send and receive messages concurrently. As it can see in Figure 5, the `sc_module` class serves as a building block for the model from which the others modules have been derived. In the pipelined DLX multiprocessor model we have used the following C++ families of abstractions:

- Data, group related values
- Inheritance, groups classes with similar behavior
- Preprocessor constructs, such as `#ifdef`, used for fine-grain variations in code and data

All the modules in our example are the `structs` or `classes` to unify a common family. This paradigm is directly supported by SystemC core standard. The next paradigm, used in the pipelined DLX multiprocessor model is inheritance, that groups modules with the same basic behavior. We have constructed two class hierarchies with the `Addressable_Device` as a base class : a hierarchy defined different types of memory and one defined communication devices. This paradigm helps sufficiently decrease the development time of specialized modules with similar behaviors. With the help of this base module, we defined a way, for modules, to communicate together.

Preprocessor directives as it mentioned in [4] are most useful to express exceptions to the rule. We use this paradigm to ease the debugging process with our model. We can, at will, enable or disable debugging of modules independently.

This multiprocessor is used as a general benchmark to obtain some figures of merit concerning the performance of SystemC in modeling and simulating a multiprocessor at a cycle accurate level.

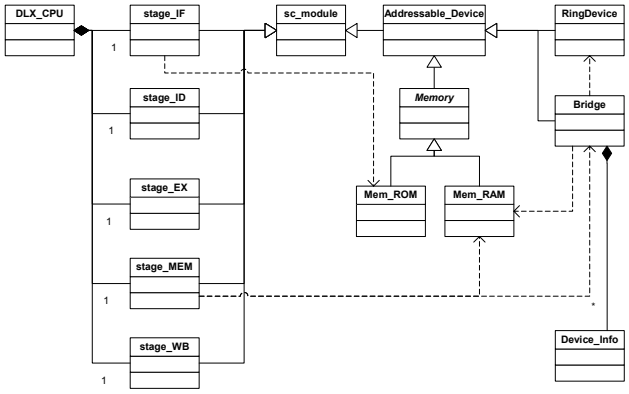

Figure 5: Class diagram of the multiprocessor model

7. Simulation Performance Evaluation

A detailed experimentation of SystemC environment is described in [8]. This section summarizes some of the findings. We run different models for the DLX multiprocessor on two SystemC versions. The pipeline stages were modeled using `SC_METHODs` and `SC_THREADs`: We first built the DLX using `SC_METHODSs`, which ends up being quite faster than `SC_THREADs`, then for the purpose of simplicity, we just changed the pipeline for `SC_THREADs` while keeping the rest of the model as being `SC_METHODs`. The usage of `SC_THREADs` results in a 30% increase in the execution time for the DLX pipeline only. We think that converting other methods to threads will result in more dramatic loss of performance. A positive point for `SC_THREADs` is that it could bring other benefits we could have used (such as the possibility to use wait statements) which would have helped us lower the total number of methods.

Usage of SystemC 2.0 instead of SystemC 1.2.1 resulted in up to 60% deterioration of performance. Again for the purpose of simplicity, we converted directly version 1.2.1 code to version 2.0 code, by inserting `dont_initialize()` clauses in the original code. This quick conversion is recommended by the specification manual, and we do not think that the `dont_initialize()` is responsible for the slow down. The metric used in our evaluation is as follows: In a DLX multiprocessor the figure of merit is computed as the number of cycles divided by the execution time, the result being multiplied by the number of processors.

The model can run as fast as 76 KHz (76,000 simulated DLX cycles per second) for a monoprocessor model running on a 450-MHz Linux machine, and as slow as 14KHz for a 128-processor model where nodes exchange messages between them.

8. Conclusion

This paper gives an overview of the multi-paradigm design methodology and demonstrates its application in hardware modeling in order to minimize the impact of the increasing complexity and the shrinking of the time to market of consumer products.

Multiple paradigms of C++ were explored to show the possible solutions to the typical modeling problems, some of them have been used with success in the modeling of a DLX multiprocessor.

SystemC offers a very interesting simulation performance and allows also the adoption of other Software Engineering Methodologies, such as design patterns, which were not addressed specifically in this work.

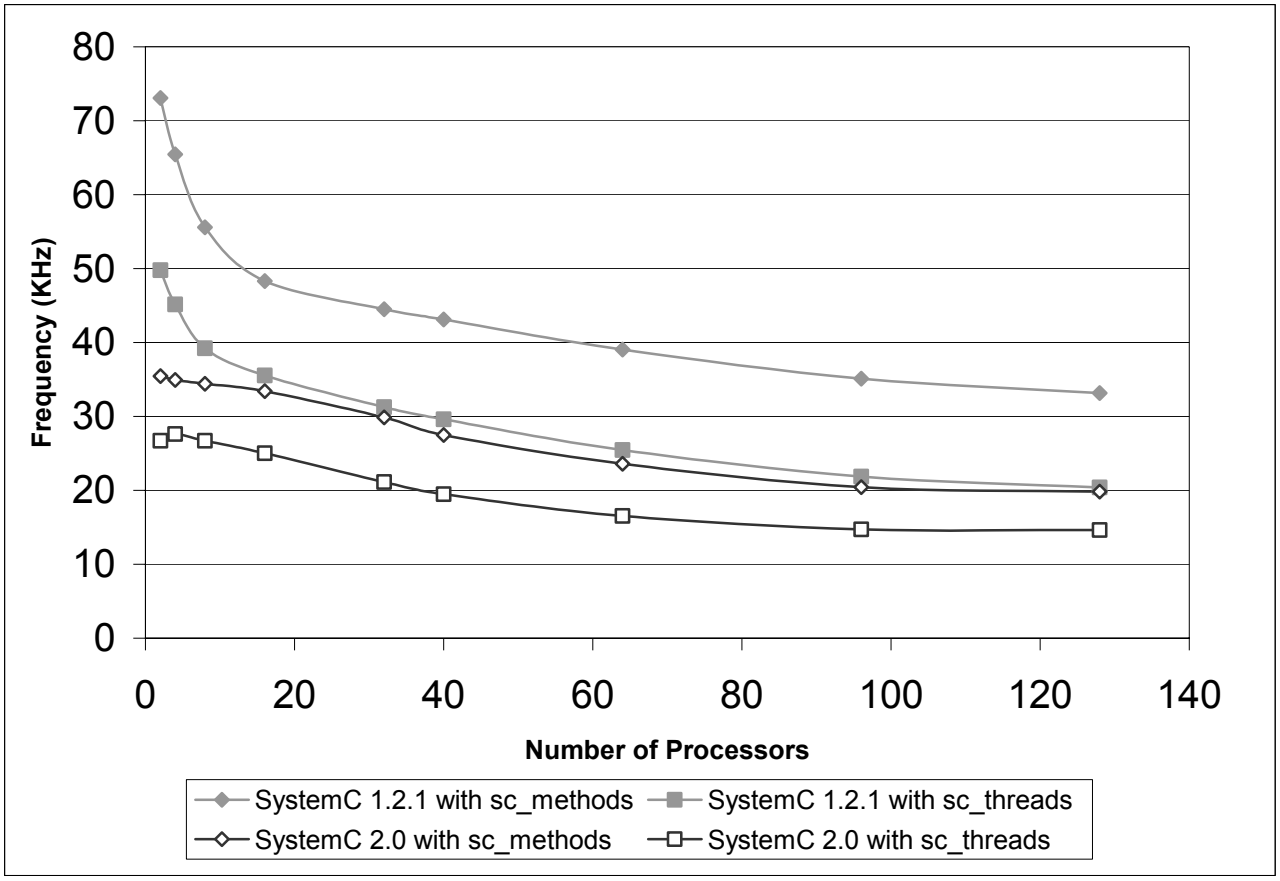

Figure 6: Multiprocessor DLX performance on a 450 MHz Linux intel pentium machine

References

[1] Open SystemC Initiative (OSCI), Functional Specification for SystemC 2.0, http://www.systemc.org, 2001.

[2] S. Swamy, A. Molin, and B. Covnot, "OO-VHDL: Object-Oriented Extensions to VHDL," *Computer*, vol. 28, pp. 18-26, October 1995.

[3] D. Verkest, J. Kunkel, and F. Schirrmeister, "System Level Design using C++," Proceedings of Design, Automation and Test in Europe, Paris, France, 27 - 30 March, 2000.

[4] Open SystemC Initiative (OSCI), SystemC, http://www.systemc.org, 1999-2001.

[5] J. Coplien, D. Hoffman, and D. Weiss, "Commonality and Variability in Software Engineering," vol. 15, pp. 37-45, November/December 1998.

[6] J. Coplien, *Multi-Paradigm Design for C++*. Reading, MA: Addison-Wesley, 1999.

[7] L. Charest, E.-M. Aboulhamid, and G. Bois, "Applying patterns and multi-paradigm approaches to hardware/software design and reuse," in *Patterns And Skeletons For Parallel And Distributed Computing*, F. Rabhi and S. Gorlatch, Eds. London: Springer-Verlag, 2002, pp. in preparation.

[8] L. Charest, E. M. Aboulhamid, C. Pilkington, and P. Paulin, "SystemC Performance Evaluation Using A Pipelined DLX Multiprocessor," Proceedings of Design and Test in Europe Designers' Forum, Paris, March, 2002.