Tutorial: Conversion from a Stand-alone Program to a PIM Program

Jonathan Misurda (jmisurda@cs.pitt.edu)
http://www.cs.pitt.edu/~jmisurda/research/cogent/
Version 1.0 (8/3/2005)

Stand-alone Program

The following code listing is a stand-alone program to do insert and search on a Binary Search Tree. This program may look familiar as it was used to determine Virtual Machine timings earlier.

It is a standard program with a node element to represent the tree, and two primary algorithms, BST insert and BST find. The program begins in the main() method by constructing a BST and the searching it.

```
import java.util.Random;

public class BinSearch
{
	public static int MAX_NODES = 10000;
	public Random r;

	private class node
	{
		int data;
		node left;
		node right;
	}

	public node bst_find(node root, int val)
	{
		if(root == null)
			return null;
		else
		{
			node current = root;
			node prev = null;
			while((current != null) && (current.data != val))
			{
				if(val < current.data) {
					prev = current;
					current = current.left;
				}
				else
				{
					prev = current;
					current = current.right;
```

```
                        }
                }
                return current;
        }
}

public node bst_insert(node root, int val)
{
        node newNode = new node();

        newNode.data = val;
        newNode.left = null;
        newNode.right = null;

        if(root == null)
                root = newNode;
        else
        {
                node current = root;
                node prev = null;
                while(current != null)
                {
                        if(val < current.data)
                        {
                                prev = current;
                                current = current.left;
                        }
                        else
                        {
                                prev = current;
                                current = current.right;
                        }
                }
                if(val < prev.data)
                        prev.left = newNode;
                else
                        prev.right = newNode;
        }
        return root;
}

public node build_tree(int nodes)
{
        node root = null;
        int i;

        for(i=0; i < nodes; i++)
                root = bst_insert(root, r.nextInt()%nodes);

        return root;
}

public static void main(String args[])
{
        BinSearch bs = new BinSearch();

        node root = null;
```

```
            int i;

            bs.r = new Random();

            root = bs.build_tree(MAX_NODES);
            for(i = 0; i < MAX_NODES; i++)
            {
                  bs.bst_find(root, i);
            }
      }
}
```

Conversion into a PIM program

The first step in the conversion to a PIM program is determining what can be done as a PIM instruction and what needs to be made as executable code for the PIM. In this example, we are assuming our instruction library contains a BST insert and a BST search algorithm. We then can convert the `main()` method into a corresponding `SimRunnableMain()` as follows:

```
public class SimBSTExample implements SimRunnable {


      public int SimRunnableMain(SimPIM argPIM)
      {
            //Your program goes here
            int max_nodes = 10000;
            Random r = new Random();
            Node root = null;

            for(int i = 0; i < max_nodes; i++)
            {
                  root = PIM_instrs.bst_insert(
                              root,
                              r.nextInt() % max_nodes
                        );
            }

            for(int i = 0; i < max_nodes; i++)
            {
                  PIM_instrs.bst_find(root, i);
            }



            //Will be hijacked by Kaffe to return the time
            //this program took
            return 0;
      }
}
```

Here, the Node object is assumed to be part of the graph package. This class then goes into the com.cogent package.

Creation of PIM instructions

The next step is to implement our BST insert and find algorithms as PIM instructions. To do this, we are required to add them as static methods to the PIM_instrs class. There are no naming or method signature requirements, so the conversion is trivial. The one addition is that we will assume that these PIM instructions have variable timings based upon the number of comparisons it takes to do the operations. We do the counting and before the method returns, we call the magical method setPIMInstrLatency() which, when this program is run on top of the modified version of Kaffe, will update the instruction timing for these methods to be the specified parameters.

The conversion of the find method:

```
public static Node bst_find(Node root, int val)
{
	int i = 0;
	if(root == null)
	{
		setPIMInstrLatency(5);
		return null;
	}
	else
	{
		Node current = root;
		Node prev = null;
		while((current != null) && (current.data != val))
		{
			if(val < current.data) {
				prev = current;
				current = current.left;
			}
			else
			{
				prev = current;
				current = current.right;
			}
			i++;
		}
		setPIMInstrLatency(5*i);
		return current;
	}
}
```

We do the same for the insert method:

```
public static Node bst_insert(Node root, int val)
{
	int i = 0;
	Node newNode = new Node();
	newNode.data = val;
	newNode.left = null;
	newNode.right = null;
	if(root == null)
	{
		setPIMInstrLatency(5);
		root = newNode;
	}
	else
	{
		Node current = root;
		Node prev = null;
		while(current != null)
		{
			if(val < current.data)
			{
				prev = current;
				current = current.left;
			}
			else
			{
				prev = current;
				current = current.right;
			}
			i++;
		}
		if(val < prev.data)
			prev.left = newNode;
		else
			prev.right = newNode;
	}
	setPIMInstrLatency(5*i);
	return root;
}
```

Putting it all together and running

To get our program to run we now must send the appropriate message to a PIM as part of the boot-up process, and so we write a new Simulation program to run our BST example:

```
class Simulator
{
      public static void main(String[] args)
      {
            SimDispatchElement lclDispatcher =
                        SimDispatchElement.getInstance();
            SimPIM lclSimPIM = new SimPIM(0);

            Thread.yield();

            SimPacket lclOutputPacket = new SimPacket();
            lclOutputPacket.setInstruction(
                  SimPacket.ATTACHED_INSTRUCTION
            );

            lclOutputPacket.setCodePayload(new SimBSTExample());

            lclDispatcher.postEvent(
                        new SimPacketArriveEvent(
                              lclOutputPacket,
                              0,
                              lclSimPIM
                        ),
                        0
            );
            lclDispatcher.postEvent(new SimBootEvent(),0);
      }
}
```

We compile this, and then run it.