

Modeling, Design, and Implementation of a Priority Buffer for Embedded Systems

Valery Sklyarov, Iouliia Skliarova

Abstract—The paper describes a model, architecture, and functionality of a priority buffer, which receives an arbitrary sequence of instructions and outputs a new sequence ordered in accordance with the priorities of the instructions that have already been received. Any new incoming instruction changes the output sequence because it has to be accommodated in the buffer on the basis of its priority. It is shown that the desired functionality of the buffer can be described efficiently by the proposed parallel hierarchical algorithms involving recursion. The algorithms have been modeled in general-purpose software and implemented in hardware (in a commercially available FPGA). The results of experiments have shown that the buffer operates in strong conformity with the requirements and specification. The required memory is allocated and deallocated dynamically. The proposed buffer architecture is easily scalable, which enables a buffer of any size to be provided.

I. INTRODUCTION

LET us consider an embedded system whose functionality is controlled by a sequential flow of external instructions. The number of instructions is not known in advance and the input instruction transfer rate is not the same as the instruction processing speed in the system. Thus, it is necessary to use input buffering. For some practical applications the instructions have to be processed non-sequentially. Each instruction is provided with additional field(s) indicating the priority or some other parameters required for the proper selection of the instruction. A priority buffer (a priority queue) is a device that stores an incoming (sequential) flow of instructions (or other data) and allows outputs to be selectively extracted from the buffer for processing. The following list exemplifies some typical selection rules:

- Each instruction (data item) is provided with an extra field indicating its priority. The selection mechanism has to be able to extract the instruction with the highest priority;
- The embedded system has to be able to remove from the buffer all the instructions that are not longer required;
- The embedded system has to be able to check if a particular instruction is in the buffer.

V. Sklyarov is with University of Aveiro/IEETA, 3810-193 Portugal (phone: +351234401539; fax: +351234370545; e-mail: skl@ua.pt).

I. Skliarova, is with University of Aveiro/IEETA, 3810-193 Portugal (e-mail: iouliia@ua.pt).

Any incoming instruction occupies the buffer memory that is allocated for it. If an instruction is removed from the buffer, the memory previously allocated for it has to be freed. Buffers of such type are required for numerous practical applications [1-8]. For example, in [1] a priority buffer (PB) stores pulse height analyzer events. Real-time embedded systems [3] employ a priority preemptive scheduling in which each process is given a particular priority (a small integer) when the system is designed. At any time, the system executes the highest-priority process. One of the proposals of [6] was to create a smart agent scanning and selecting the data according to their priority. A similar technique is also required for advanced control systems considered in [7]. Note that although the accumulated data can be of different types and the application of the buffer can vary, the basic operations and general functionality of the buffer is very similar to that described above. The architectures and design methods used are also diverse and they are mainly targeted to implementation in software. Many of them are based on sort and shift algorithms [5].

The paper suggests a technique for hardware implementation of a PB with the following distinctive features: a) run-time data sorting in a single buffer memory; b) dynamic memory allocation and deallocation in hardware. This technique includes: a) description of the PB by the proposed parallel recursive algorithms; b) modeling the algorithms in software; and c) implementation of the algorithms in hardware with the aid of a parallel hierarchical finite state machine. The technique has been employed for the design of a PB on the basis of commercially available field-programmable gate arrays (FPGAs). The functionality of the PB has been verified and a number of experiments have been performed.

The remainder of the paper is organized in five sections. Section II introduces and describes the model, architecture and the desired functionality of a PB. Section III presents parallel hierarchical algorithms for a PB (also involving recursion) and reports the results of modeling the buffer in software. Section IV gives details of a hardware implementation that emphasizes parallelism, hierarchy, and dynamic memory allocation/deallocation. Section V demonstrates the basic software and hardware architectures for the experiments and shows the results. The conclusion is given in Section VI.

II. MODEL, ARCHITECTURE, AND FUNCTIONALITY OF PRIORITY BUFFER

The proposed model is based on the incremental construction and processing of a special graph, which is similar to the binary tree considered in [9]. Incoming vectors (IV), representing instructions (data) and additional field(s), are ordered somehow, for example, by priorities or by the value of an instruction (data) code. The nodes of the tree contain N ($N \geq 3$) fields that are: a value of an IV, a pointer to the left child node, a pointer to the right child node and possibly some additional fields, for example: a counter indicating the number of occurrences of the value associated with the respective node; an additional pointer to the left child (right child) if more than one tree has to be built, etc. The nodes are maintained so that at any node, the left sub-tree only contains values that are less than the value at the node, and the right sub-tree contains only values that are greater. In order to build this tree for a given set of IVs, we have to find the appropriate place for each incoming vector in the current tree. In order to extract a value, we can apply a special technique that depends on the selection rules, and can be based on forward and backward propagation steps that are exactly the same for each node.

In the proposed model each node of the tree is associated with an incoming vector in such a way that:

- The first IV is associated with the root node of the tree;
- The relationship with other IVs is provided through pointers, i.e. through addresses of IVs in the buffer memory. We assume that such memory is allocated dynamically for implementation in software and sequentially for implementation in hardware. The latter will be considered in detail in section IV;
- Any new IV (after the first) is accommodated on the tree in such a way that it satisfies the basic rules, namely at any node, the left sub-tree only contains values that are less than the value at the node, and the right sub-tree contains only values that are greater;
- More than one tree can be constructed from the root node. For example, the first tree could order the IVs by their priorities and the second tree by the values of the instruction codes. In this case the number of pointers to the left and right sub-trees is doubled. Note that the number of IVs stored is the same and the memory for each cell is increased only by the amount required for storing additional addresses.

Fig. 1 depicts a general architecture for a PB. It is composed of 3 primary blocks, which implement the buffer:

- 1) building the tree with the characteristics considered above;
- 2) extracting data by applying the rules mentioned above;
- 3) rebuilding the tree (removing the nodes that are no longer required).

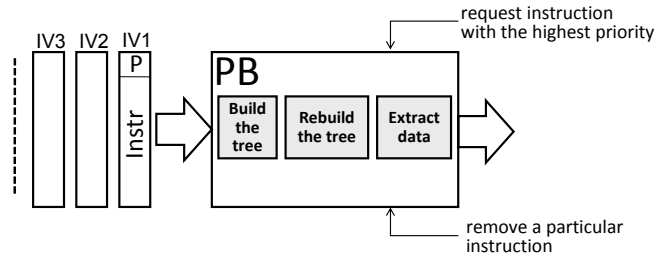


Fig 1. General architecture of a priority buffer.

Fig. 2 gives an example. Suppose it is necessary to build a tree for the sequence of IVs shown in Fig. 2, a. The vectors have to be ordered by their priorities (see the upper values written within rectangles representing IVs, such as 3,6,1,5,...) and by the instruction codes (see the lower values, such as 26,21,16,9,...).

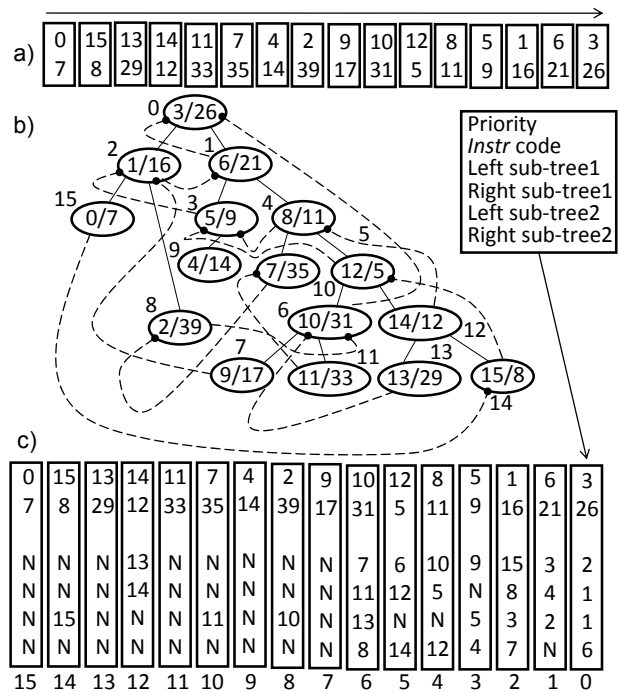


Fig 2. An example demonstrating how the tree for the given sequence (a) has been constructed (b) and how it has been stored in memory (c).

The first IV is 3/26 (priority = 3; instruction code = 26) and node 0 of the binary tree has been allocated (see Fig. 2, b). This node is the root for two trees, which are built differently. The first tree orders the IVs by their priorities and the second tree - by their instruction codes. The edges of the first tree are shown by solid lines and the edges of the second tree are shown by dotted lines (in the latter case, the connections of the lines with parent nodes are indicated by small filled circles for better visibility). Within each node, the priority and the instruction code are separated by a slash.

Fig. 2, c shows how the IVs are stored in the PB memory. Memory addresses are shown at the bottom and they are the same as the node numbers in the trees. Each cell with an address A ($A=0,1,\dots,15$) contains 4 additional fields that are: a pointer (address) to the left sub-tree of the first tree; a pointer to the right sub-tree of the first tree; a pointer to the left sub-tree of the second tree; and a pointer to the right sub-tree of the second tree. For example (see Fig. 2,b and 2,c), the node 4 (8/11) is stored at the address 4 and it has: a pointer 10 to the left sub-tree (7/35) of the first tree; a pointer 5 to the right sub-tree (12/5) of the first tree; a pointer N to the left sub-tree of the second tree where N is some predefined value indicating that the left sub-tree does not exist; and a pointer 12 to the right sub-tree (14/12) of the second tree.

The next section shows the method for building trees with the structure considered. Extracting pre-ordered (by the tree) vectors is rather simple. It is necessary to traverse the tree in such a way that just the right sub-trees are selected at each step and if the right sub-tree does not exist, it means that the rightmost node has been reached containing the highest value. This is either the IV with the highest priority (see the first tree in Fig. 2, b) or the IV with the largest value of the instruction code (see the second tree in Fig. 2, b). After getting a vector with the highest value, the traversal process backtracks to the nearest parent node and extracts its value. Then the left sub-tree is explored in the same manner.

In order to extract nodes with given values it is necessary to apply a special method, which will be presented in the next section.

III. ALGORITHMS AND MODELING IN SOFTWARE

To execute the operations required for the PB we can use a variety of techniques. We will apply recursive algorithms because of their clarity and effectiveness for operations over binary trees, which was shown in [10,11] on numerous examples. Although in software iterative algorithms over binary trees reveal slightly better performance, the implementation of recursive algorithms in hardware often gives the opposite result [11,12] and the relevant circuits consume less resources and exhibit better performance. It will be shown below that forward and backward propagation steps are exactly the same for each node. Thus, a recursive procedure can be applied directly.

Fig. 3 depicts a flow chart for the basic algorithm of the PB composed of 7 modules, which are: Z_0 – the top level algorithm; Z_1 – gets IVs and activates Z_4 for each of them (adding new IV to the tree); Z_2 – extracts an IV from the buffer (with the aid of module Z_6) using the selected priority rule; Z_3 – provides synchronization with other modules and removes unneeded tree nodes already extracted by Z_6 or on a request from the embedded system; Z_4 – provides synchronization with other modules and adds a new IV to

the tree; Z_5 – removes an unneeded tree node; Z_6 – finds the required IV using the selected priority rule and sends it to the embedded system. For simplicity Fig. 3 shows just the operations needed for functionality of the PB and the operations for interfacing with the embedded system are hidden.

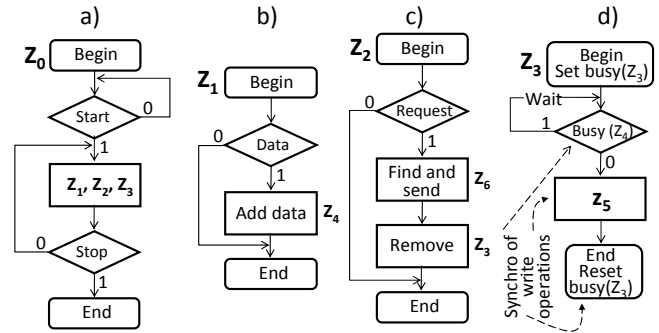


Fig 3. Basic algorithms of a PB.

Note that Z_1 , Z_2 and Z_3 might be executed in parallel in case if access to shared buffer memory is properly synchronized. We will apply pseudo parallelism in the software implementation and explain how to execute different modules in parallel in hardware circuits in the next section. Flow charts for the modules Z_4 , Z_5 and Z_6 are not shown in Fig. 3 because these modules are described in detail in C++ code fragments presented below.

The basic modules, that construct the binary tree and permit nodes to be removed from the tree, are Z_4 and Z_5 . The other modules execute supplementary operations and synchronize access to the shared memory. The following C++ code fragments describe the primary operations of the modules Z_4 , Z_5 and Z_6 (for simplicity, exception handling for such problems as errors in memory allocation is not shown).

```
// Z4 module
tree_node* build_tree(tree_node* node, int value)
{ if (node == 0)
  { node = new tree_node;
    node->value = value;
    node->c = 1; // setting counter to 1
    node->r = node->l = 0;
  }
  else if (value == node->value)
    node->c++; // incrementing counter
  else if (value < node->value)
    node->l = build_tree(node->l, value);
    // traversing the left sub-tree
  else
    node->r = build_tree(node->r, value);
    // traversing the right sub-tree
  return node;
}

// Z5 module
void extract_from_tree(tree_node*& node, int value)
{ tree_node *temp_node;
  if (node != 0) // verifying if node exists
    if (value > node->value)
```

```

    // traversing the right sub-tree
    extract_from_tree(node->r, value);
else if (value < node->value)
    // traversing the left sub-tree
    extract_from_tree(node->l, value);
else
{ if ( (node->l == 0) && (node->r == 0) )
// in this case the node has to be deleted
{ delete node;
  node = 0;
}
else if (node->r != 0)
{ // changing pointers for the right node
  temp_node = node->r;
  if ((node->l) != 0)
    build_subtree(temp_node, node->l,
                  node->l->value);
  node->r = temp_node->r;
  node->l = temp_node->l;
  node->value = temp_node->value;
  node->c = temp_node->c;
  delete temp_node;
}
else
{ // changing pointers for the left node
  temp_node = node->l;
  node->r = temp_node->r;
  node->l = temp_node->l;
  node->value = temp_node->value;
  node->c = temp_node->c;
  delete temp_node;
}
}
}

// Z6 module
void extract_most_priority(tree_node* node)
{ if (node != 0)
  { while (node->r != 0)
    { node = node->r;
      // send node->value to the embedded system
    }
  }
}

```

In this code the `tree_node` is considered to be the following structure:

```

struct tree_node
{ int value; // node value (instruction code)
  int c; // counter for repeated values
  struct tree_node* l; // pointer to the left
                    // sub-tree
  struct tree_node* r; // pointer to the right
                    // sub-tree
  // other fields if required
};

```

The `build_subtree` function is a simplified `build_tree` function with the following code:

```

tree_node* build_subtree(tree_node* node
                        tree_node* subnode, int value)
{ if (node == 0) node = subnode;
  else if (value < node->value)
    node->l =
      build_subtree(node->l, subnode, value);
  else node->r =
      build_subtree(node->r, subnode, value);
  return node;
}

```

Section V explains how the C++ functions considered above have been used for modeling the priority buffer in software.

IV. IMPLEMENTATION IN HARDWARE

In order to implement the same algorithms (see Fig. 3) in hardware, we must solve the following two problems:

- Provide for implementation of recursive calls, which are not directly supported by hardware description languages;
- Manage dynamic memory allocation/deallocation, which is significantly more difficult comparing to software.

We can solve the first problem with the aid of a hierarchical finite state machine (HFSM) [13] enabling the hardware circuits to implement hierarchical and recursive calls.

The following technique has been used to allocate and free memory dynamically. Storage for the PB (such as that shown in Fig. 2, c) has been implemented in a memory block with a fixed number of cells (such as those numbered by the indices 0,1,2,...,15 in Fig. 2,c). A special register is provided that contains the index of the memory cell holding the root node of the tree. Each cell is expanded with a one bit flag field – F, indicating whether the cell is occupied (F=1) or not (F=0). The tree is constructed sequentially in such a way that for any new incoming node, the first cell from the beginning for which F=0 is selected. As soon as a node is removed the relevant flag F is set to 0 indicating that the cell can be reused to store new data. Thus, the cells are occupied and emptied during run time and dynamic memory allocation and deallocation is enabled.

This process is illustrated in Fig. 4. Let us assume that the size of the fixed memory block is 8, the tree is constructed only on the basis of instruction priorities and that the incoming data are the same as in Fig. 2, a (see Fig. 4, a). Suppose initially four instructions (3/26, 6/21, 1/16 and 5/9) arrive. They will be stored in memory as shown in Fig. 4, b. Then the two instructions with the highest priority (6/21 and 5/9) are extracted. As a result, the memory will be changed (see Fig. 4, c). If three new instructions from the incoming flow in Fig. 4, a (8/11, 12/5 and 10/31) then arrive, they will be stored in memory as shown in Fig. 4, d. Fig. 4 e, f, g illustrate: extracting the highest priority instruction 12/5 (Fig. 4, e); receiving three new instructions (9/17, 2/39 and 4/14) from the incoming flow in Fig. 4, a (Fig. 4, f); and extracting the five highest priority instructions 10/31, 9/17, 8/11, 4/14, and 3/26 (Fig. 4, g). Now cell 2 represents the root of the tree and there are 2 nodes in the tree. The index 2 is kept in the special register mentioned above and this enables the traversal procedure to be started from the correct node.

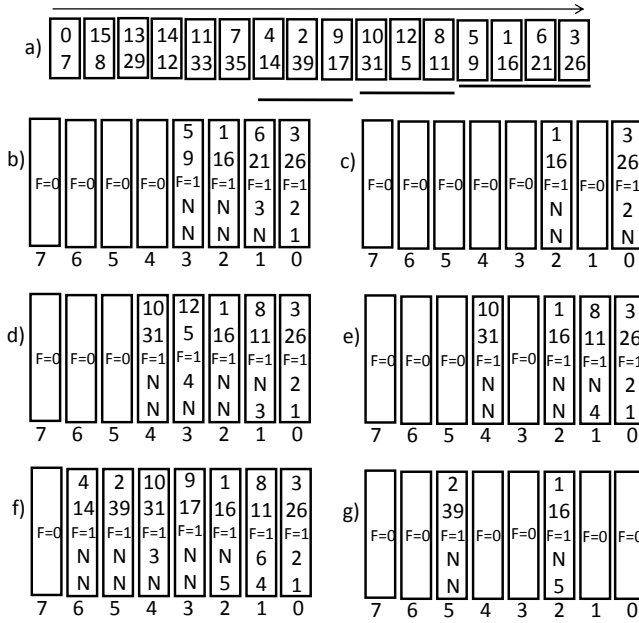


Fig 4. Dynamic memory allocation/deallocation in hardware.

The modules Z_0 - Z_6 have been considered as behavioral specifications of an HFSM whose register memory has been replaced with a stack. The basic architecture of such an HFSM is described by a known HFSM template [10], i.e. by a predefined customizable hardware description language (HDL) code.

Specifications of the modules Z_0 - Z_6 are used to properly customize the HFSM template and the resulting HDL code is synthesizable in commercially available CAD tools, such as Xilinx ISE [14] for FPGAs. The HFSM enables any module to activate itself [13]. Thus, the required recursive calls in the modules can be executed. Parallel calls, such as Z_1, Z_2, Z_3 in Fig. 3, a, have been implemented in the HFSM through the use of multiple stacks (one stack for each parallel module). Dynamic memory allocation and deallocation, such as *node = new tree_node;* in the module Z_4 and *delete temp_node;* in the module Z_5 , have been provided in accordance with the technique described above in this section. Additional modules Z_7 (instead of the C++ operator *new*) and Z_8 (instead of the C++ operator *delete*) were used for such purposes.

V. EXPERIMENTS

The description and modeling of the PB was done in C++ (Microsoft Visual Studio). The primary objective was to verify the algorithms (see section III) and the intended functionality allowing subsequent implementation in hardware. Fig. 5 gives the general architecture of the model.

The PB implements the algorithms shown in Fig. 3, permitting incoming data to be accumulated in memory that is dynamically allocated and deallocated according to the external requests from blocks A and B. Block A generates a

sequence of instructions with random priorities. Block B forms random requests for getting instructions with the highest priority and removing instructions that are no longer required. Input/output timing parameters have also been selected randomly and differently for the blocks A and B. Instead of parallel execution of the modules Z_1, Z_2, Z_3 , pseudo parallel mode was implemented, i.e. an individual time slot for each module was used. The time slots for modules Z_1, Z_2, Z_3 were equal and they were repeated cyclically: 1,2,3,1,2,3, etc. The experiments in software showed that the proposed algorithms and architecture of PB are correct and can be implemented in hardware.

Note that all the experiments were performed in software in pseudo parallel mode. Thus, the design and test of hardware is very important because only in hardware we can verify the required parallelism.

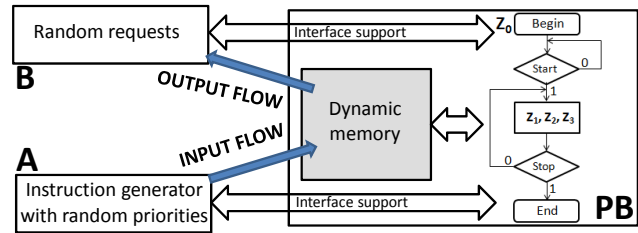


Fig 5. Modeling of PB in software.

Hardware implementation was done in an FPGA of Spartan-3 Xilinx family using DETIUA-S3 [15] and NEXYS2 [16] prototyping boards. The control circuit for the PB was implemented on the basis of an HFSM VHDL template that contains a reusable predefined code for HFSM stack memory and a customizable code for HFSM combinational circuits. Customization was done in accordance with the algorithms described in section III and modeled in software. Synthesis of the HFSM was realized using the methods [10,13]. Parallel modules (such as those shown in Fig. 3, a) were executed with the aid of individual stacks for each module running in parallel. Thus, real parallelism has been achieved. Fig. 6 gives the general architecture of the hardware implementation.

A processing unit, which is composed of a parallel HFSM with an attached datapath, implements the algorithms considered in section III. The datapath consists of hardware circuits controlled by the FSM that generate branching/waiting conditions for the FSM. In general, these circuits permit access to shared memory to be properly synchronized, provide temporary storage for executing operations, etc. The method described in section IV was used for dynamic memory allocation/deallocation. The parallel FSM also provides for an FPGA interface with supplementary circuits that execute the same functions as the blocks A and B in Fig. 5. The block A continuously

generates instructions. If the memory in Fig. 6 is full, the PB sets an indicator requesting the block A to suspend generation of instructions. As soon as some memory is free generation of instructions is resumed. The memory is organized as 512 of 36 bit words. Obviously, the size of memory can be increased easily.

All components of the processing unit in Fig. 6 were described in VHDL. Synthesis from the VHDL specification was done in Xilinx ISE [14]. The experiments with the PB implemented in hardware have demonstrated that all the requirements have been satisfied and the PB functions in close conformity with the specification.

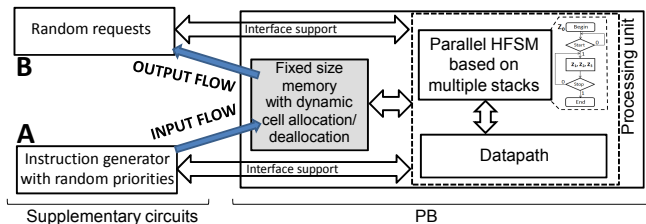


Fig. 6. Implementation of PB in hardware (in FPGA) and supplementary circuits for experiments.

The buffer has been used in an automatic system for garage control described in [17] for managing parking slots and priorities based on accumulation of data about free parking slots with their priorities for potential parking of (new) arriving cars. The accumulation is done in a PB, which takes input data items about released slots and outputs items with the highest priority. The PB is organized in such a way that the established priorities might be dynamically rearranged as well as some items can be removed on external requests (for example, when some parking slots are reserved for special purposes). The PB is organized on the basis of the proposed tree-based structure and it has been implemented, validated and tested in hardware.

VI. CONCLUSION

Priority buffers enable incoming data to be accumulated in memory in accordance with additional information indicating the order in which these data have to be processed in an embedded system. The proposed buffer is built on the basis of a fixed size memory that is filled and emptied dynamically. The number of occupied memory cells is equal to the number of incoming data that are ordered with the aid of a binary tree which is dynamically coded and kept in the buffer.

To construct and to process the tree, a set of parallel and hierarchical specifications have been suggested and modeled in software. It is shown that these specifications can be implemented in hardware with the aid of a parallel hierarchical finite state machine that also supports an

external interface and dynamic memory allocation and deallocation. The buffer was modeled in C++ and further described in VHDL, synthesized, and implemented in FPGAs of Xilinx Spartan-3 family.

REFERENCES

- [1] R.A. Mewaldt, C.M.S. Cohen, W.R. Cook, *et al.*, "The Low-Energy Telescope (LET) and SEP Central Electronics for the STEREO Mission", *Space Science Rev.*, 136, pp. 285–362, 2008.
- [2] *Proceedings of the Second UK Embedded Forum*, Newcastle, Leicester, Southampton, 2005.
- [3] S.A. Edwards, "Design Languages for Embedded Systems", Computer Science Technical Report CUCS-009-03, Columbia University, May, 2003.
- [4] H. Lonn and J. Axelsson, "A Comparison Of Fixed-Priority And Static Cyclic Scheduling For Distributed Automotive Control Application", *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, York, England, pp. 142-149, 1999.
- [5] P.A. Pietrzyk and A. Shaoutnew, "Message Based Priority Buffer Insertion Ring Protocol", *Electronics Letters*, vol. 27, no. 23, pp. 2106-2108, 1991.
- [6] H.T. Sun, "First Failure Data Capture in Embedded System", *Proceedings of IEEE IIT*, 2007, May 17-20, Chicago, USA, pp. 183-187, 2007.
- [7] V. Sklyarov, "Models, Methods and Tools for Synthesis and FPGA-based Implementation of Advanced Control Systems", *Proceedings of the 2nd International Conference on Mechatronics, ICOM'05*, Kuala Lumpur, Malaysia, pp. 1122-1129, 2005.
- [8] T. Lin, "Mobile Ad-hoc Network Routing Protocols: Methodologies and Applications", Ph.D. thesis, Blacksburg, Virginia, 2004.
- [9] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice Hall, 1988.
- [10] V. Sklyarov, "Reconfigurable models of finite state machines and their implementation in FPGAs", *Journal of Systems Architecture*, 47, pp. 1043-1064, 2002.
- [11] V. Sklyarov, I. Skliarova, and B. Pimentel, "FPGA-based Implementation and Comparison of Recursive and Iterative Algorithms", *Proceedings of the 15th International Conference on Field-Programmable Logic and Applications - FPL'2005*, Finland, pp. 235-240, 2005.
- [12] V. Sklyarov and I. Skliarova, "Recursive and Iterative Algorithms for N-ary Search Problems", International Federation for Information Processing, vol. 218, *2nd IFIP Symposium on Professional Practice in Artificial Intelligence - AISPP'2006*, ed. J. Debenham, *19th IFIP World Computer Congress - WCC'2006*, Santiago de Chile, Chile, pp. 81-90, 2006.
- [13] V. Sklyarov, "Hierarchical Finite-State Machines and Their Use for Digital Control", *IEEE Transactions on VLSI Systems*, vol. 7, no. 2, pp. 222-228, 1999.
- [14] Xilinx products, available at: www.xilinx.com.
- [15] M. Almeida, B. Pimentel, V. Sklyarov, I. Skliarova, "Design Tools for Rapid Prototyping of Embedded Controllers", *Proceedings of the 3rd International Conference on Autonomous Robots and Agents - ICARA'2006*, Palmerston North, New Zealand, pp. 683-688, 2006.
- [16] Digilent products, available at: <http://www.digilentinc.com/>.
- [17] V. Sklyarov, I. Skliarova, A. Neves, "Modeling and Implementation of Automatic System for Garage Control", *Proceedings of the ICROS-SICE International Conference 2009*, Fukuoka, Japan, August 2009.