An Introduction to the MISD Technology

Aleksey Popov
Bauman Moscow State Technical University,
Moscow, Russia
alexpopov@bmstu.ru

Abstract

The growth of data volume, velocity, and variety will be the global IT challenges in the next decade. To overcome performance limits, the most effective innovations such as cognitive computing, GPU, FPGA acceleration, and heterogeneous computing have to be integrated with the traditional microprocessor technology. As the fundamental part of most computational challenges, the discrete mathematics should be supported both by the computer hardware and software. But for now, the optimization methods on graphs and big data sets are generally based on software technology, while hardware support is promising to give a better result.

In this paper, the new computing technology with direct hardware support of discrete mathematic functions is presented. The new non-Von Neumann microprocessor named Structure Processing Unit (SPU) to perform operations over large data sets, data structures, and graphs was developed and verified in Bauman Moscow State Technical University. The basic principles of SPU implementation in the computer system with multiple instruction and single data stream (MISD) are presented. We then introduce the programming techniques for such a system with CPU and SPU included. The experimental results and performance tests for the universal MISD computer are shown.

1. Introduction

The Internet of Things, social media, and mobile devices generate Exabyte's of unstructured data, which are extremely hard to be stored and analyzed with the commonly used computing technology [1]. Despite the fact that the performance of multicore CPUs achieved a great milestone, the conceptual and technological problems became manifest for modern computers as current challenges. The turning point came when single core performance started going backwards and showed limitations in further development [2]. Such initiatives as heterogeneous architectures, OpenPOWER [3] or HSA Foundation [4], allows one to create a high performance system with different types of accelerators. However, the set of acceleration cores is still limited from the GPUs and cryptography accelerators to digital signal processors and graphic accelerators. Let us focus on the important issues of discrete optimization, which is widely used in the most computational challenges, but is not fully supported by the acceleration technology.

Discrete optimization applications arise in applied science (e.g., bioinformatics, chemistry, statistics, physics, and economics) and in industries (e.g., telecommunication, manufacturing, retail, and banking). Therefore, it is important to note the disadvantage of universal computing when discrete optimization could be only processed through multiple calls of primitive arithmetic operations. There is still no discrete mathematic units in current computers to operate over data sets directly. All this confirms the importance of effective hardware to process big data sets.

The purpose of this paper is to shortly describe the fundamentals and implementation results of a principally new computing system with multiple instruction streams and a single data stream (MISD under Flynn's Taxonomy), which were developed and implemented at Bauman Moscow State Technical University.

The rest of this paper is structured as follows. In the next section, we present some technical arguments to implement the special hardware accelerator for discrete optimization. Section 3 presents the fundamentals of MISD system operations. The micro-architecture and instruction set of the new special-purpose microprocessor (called the Structure Processing Unit or SPU) will be introduced in Section 4. Since the SPUs instruction set differs much from generic CPUs, it was required to modify algorithms in the parallel MISD form and to implement a special compiling technology. Section 5 presents the hierarchical nesting principle of computer systems architecture. The programming techniques for the new architecture is briefly introduced in Section 6, and the example of the useful MISD algorithm is shown in Section 7. Implementation features and experimental results are presented in Section 8. Finally, Section 9 provides the preferable implementation and conclusions, including future research.

URI: http://hdl.handle.net/10125/41272
ISBN: 978-0-9981331-0-2

2. Background

The fundamentals of discrete mathematics are implemented in computing technology as functions to store and operate over multiple discrete structures. A number of data structures such as arrays, hashes, lists, and trees have been developed to accelerate algorithms [6, 7]. B+trees, for example, are speeding up the data retrieval in databases by managing indices in a more effective manner [8].

It should be noted that computational problems for most data structures are caused by CPU pipelines and slow memory subsystem. Nowadays, microprocessors and memory devices are developed to effectively perform sequential memory access. But the random access for lists and trees raises much more cache misses and page faults than the sequential one for vectors or arrays. In addition, lists and trees use pointers to follow from one element to another. Only after passing the pointer through the CPU pipeline will the next address become known to the Memory Management Unit [9]. On the other hand, the CPU's speed is growing at a much faster rate (60% per year) than the memory (10% per year) [10]. This causes penalty of the pipeline to wait for required operands. So, the access to the sequential memory's addresses is about thirty times faster than to random addresses.

Some recent works addressed the development of load/store accelerators [11, 12], and Field-Programmable Gate Arrays (FPGA) are often used as a hardware platform for such an application. In contrast with the processor pipeline, FPGAs allow the implementation of a massively parallel infrastructure to reach an optimized computing speed [13, 14, 15]. But the functionality of ”key-value” accelerators, which have been developed for a particular algorithm, is always restricted. The extended instruction set of such a system is limited to a few basic operations over data structures, such as search, insert, or delete values by their keys. At the same time, discrete mathematic operations are required to support such operations as slicing, union, intersect, or complement. Other useful operations are the following: the traversal of data structure elements according to their sequence; maximum or minimum search; the cardinality operation to understand the number of elements in the set; search of a neighbor; and others. This set of operations is required in many real-life algorithms like Dijkstra's shortest path search or Kruskal's minimum spanning tree search. Next, we will show the main idea of the new architecture as a series of principles.

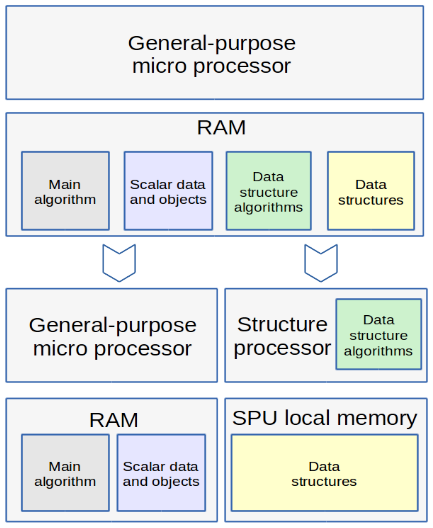


Figure 1. Computer system with hardware support of operations on large data sets

3. System description

From a formal point of view, the data structure combines two types of information: particular information about stored data (i) and information about the structure organization itself (ii). For example, the binary search tree [7] consists of the data set (as the information part) organized due to the structure relationships (binary tree as the structure part). When we have to find the minimum data element from the tree, we do not need to know the particular data value but should operate only on its relations. The duality of data structures is clear but has been never used by the hardware to share computing into two threads: first, to compute over the structure part, and second, to compute over the information part. In this work, we develop the special hardware processor to directly support the operations over structure relations in parallel with operations over scalar data.

In Figure 1, we show the basic concept of a new architecture. It is clear that the microprocessor in a generic system should execute both the main algorithm and the algorithm to operate data relations. Even if the multi-threaded microprocessor is used, the programmer has to make great efforts to create a parallel code in order to access the shared memory for multiple threads. To extend the ability of parallel code execution, we dedicate the relations computing thread to the SPU. This device should operate over structured data in the local memory and therefore is able to independently execute the special instruction set. As a result of those instructions, scalar data moves from SPU to CPU for further computing.

Principle 1. *The SPU is a special unit that processes the relational part of data structures while the CPU performs computations on the informational part of data structures.*

Let us consider an example to understand this principle and its advantages.

3.1 How this system works

The search for the minimum (or maximum) value from a big data set is the usual problem in optimization algorithms. For network routing, it is often necessary to define a path with minimum latency in order to allow data transmission as quickly as possible. Those operations are required for motion planning in robotic systems when it is necessary to choose a set of single movements in order to take a position with minimum energy consumption. Thus, it is important to determine a computing system architecture that is most effective for such an operation. If the introduced system is used for that purpose, this makes it possible to store data sets in the SPU's local memory and to request values in the desired sequence under the separate control flow (Figure 2). At the moment, when CPU will really request the minimum value, the data will be already provided by the SPU. Therefore, the acceleration effect can be achieved due to parallel operations over data sets in the SPU simultaneously with the generic workload execution in the CPU.

Even if it is impossible to understand in advance what kind of data will be requested by the main algorithm, the CPU just has to transmit the request to the SPU, and then, it will be able to make other computations at that time. Both processors use their dedicated local memory and should not conflict during memory access. Thus, the entire system works faster due to deep parallelism and efficiency.

3.2 CPU and SPU interactions

To execute both scalar and structure operations in parallel, the independent execution pipelines should be organized for the SPU and CPU. We propose to use the system architecture as shown in Figure 2. As usual, the CPU starts all initial procedures and reads data from the I/O subsystem (flash memory, disk storage, network, SSD, etc.). Following this, the CPU initializes its data segments in the local memory and inserts the data structure into the SPU through multiple "insert" commands. Data structures are organized in the form of key-value pairs inside the SPU's memory.

Principle 2. *The SPU has independent access to the local memory in order to store data structures and execute instructions. The results of the SPU's instructions are forwarded to the CPU for further use in the computational algorithm.*

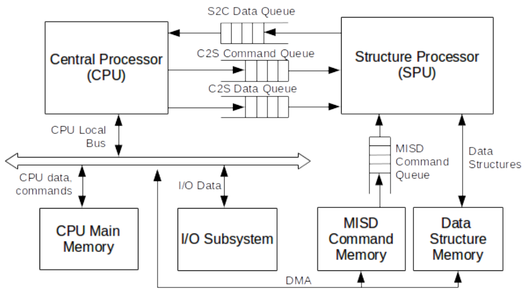


Figure 2. Multiple instruction and single data stream computer architecture

There are two possible ways to start the SPU's instruction: when CPU sends it through the C2S command queue (i) and when the SPU starts it from the local command memory (ii). If the first one is used, then this system is similar to the heterogeneous system with an accelerator. If the second way is used, this system can be classified as a parallel computer with multiple instructions and single data stream architecture.

3.3 About the Flynn's taxonomy

From the M. J. Flynn taxonomy, the four classes of computer systems are known, which are the following: SISD (single instruction stream and single data stream), SIMD (single instruction stream and multiple data stream), MISD (multiple instruction stream and single data stream), and MIMD (multiple instruction stream and multiple instruction stream). All those classes, excluding MISD, are used in generic computer systems, while experts do not place any working system in the MISD category [9]. To be sure, this type of computer architecture was previously implemented in 1970 for systolic arrays [5] and in space shuttle flight control computers. But systolic arrays were highly specialized for applications and therefore not widely implemented. The information about space shuttle computers is very limited and still unclear. In any case, those systems were not developed to perform discrete optimization by the hardware.

Argument 1. The data structure is the representation of data and its relations inside the single data stream. This is true because even the sequence or the absence of any data element can be used by the algorithm as useful information. If the algorithm operates over the data structure, there are two sequential stages of computing:

operations on the structure part to solve data relations (i); as the result of the first stage, it becomes possible to operate on scalar data (ii).

Argument 2. The SPU and CPU operates in parallel and independently. The ability to execute both threads in parallel is supported by the CPU and SPU and their local memories. When the system allows the execution of code independently to solve one problem, this means the computation of the multiple instruction streams. The execution, admittedly, may not be free of data dependencies. But in comparison with microcode dependencies in the processor pipeline, the SPU and CPU operate under independent control sequences and are not defined just as one operation stage. The example of such operations is the graph traversal algorithm when we need to compute every graph's vertex one by one. The SPU can do this with different traversal algorithms: Breadth-First Search (BFS) or Depth-First Search (DFS) [7]. Thus, the CPU uses results of graph traversal to solve the main problem with shared and independent instruction stream.

Principle 3. *The computing system with hardware support of data structures executes multiple instruction streams for a single data stream.*

The SPU execution thread uses two types of synchronization with the CPU thread. The first one is a special semaphored operand in the SPU instruction to fix the situation when the SPU should wait for that operand from the CPU. For that purpose, the instruction consists of the operation field, operands, and their tags. If the operands should be received from the CPU data bus, their tag has to be set to the semaphored state. Similar to generic microprocessors, the SPU has the conditional jump instruction to move the execution flow to a target instruction. This command also has two semaphored operands: target instruction address (i) and direction bit (ii). In addition, the SPU is able to execute commands from the CPU's bus or from the local command memory. This ensures the system's ability to operate in several modes.

Principle 4. *The SPU can operate in MISD mode, in accelerator mode, and in combined mode.*

4. SPU microarchitecture and instruction set

The SPU uses the B+tree model to perform all operations and to access data in the local memory. The first reason for this is known from the Database Management System (DBMS): this type of trees is extremely effective for block-oriented memory access. Due to the fact that we have to use a burst-oriented Random Access Memory (RAM), it allows the loading and storage of big data blocks faster. The second reason is that B+trees may have high fan-out in a node that reduces the number of memory access operations to find the element. In addition, the fan-out can be easily changed to implement the variety of data structures from tiny to huge.

Therefore, the SPU stores information as key-value pairs in the form of non-overlapping B+trees. Thus, every instruction includes from one up to three operands: one to three tree numbers, key, and value. The SPU has a pipelined microarchitecture to provide B+tree processing in parallel: it allocates memory for nodes and leaves, performs search in subtrees, provides insertion and deletion operations through the tree structure [16]. Most of the SPU commands require O(log n) memory access operations, but they can be executed much faster in comparison with generic CPU programs due to the effective SPU's hardware and the special caching technology. Such a technology allows us to speed up the computing process over keys and to store the search path inside the special internal cache (named Catalogue or CAT).

Principle 5. *The sequence of B+tree nodes on the route from the tree root to its leaf is called "trace". The trace is stored in the SPU's cache memory for immediate access.*

Let us consider the SPU's microarchitecture, which is shown in Figure 3. B+tree processing can be formally divided into its tracing stage and leaf operation stage. Thus, there are two units to perform them. CAT performs the processing of the tree forest from the root down to the last internal level. The second stage is performed by the Operational Buffer (OB), which is used to operate keys and values on the leaf level. The SPU's memory subsystem includes multilevel storage devices:

- First-level memory comprises of registers to store nodes and leaves for immediate access inside the CAT and OB. It is organized as an associative memory to perform useful operations such as searching, shifting, union, slicing, etc.
- The second-level is the Internal Data Structure Memory (IDSM), which is boundary addressable. All bounds are processed in CAT to define the physical addresses for IDSM access.
- The third-memory level is the external Data Structures Memory (DSM) that stores all data structures outside the SPU. This memory is a typical RAM and should have a large capacity to store all data structures.

CAT consists of multiple single Catalogue Processor Elements (CPE) to perform root and internal node processing (excluding bottom leaves). Every B+tree has many boundaries to define the sub-tree, where the search key can be stored. To make processing more parallel, the Control Unit (CU) sends the same microcode to all CPEs in order to perform the same operations for dif-

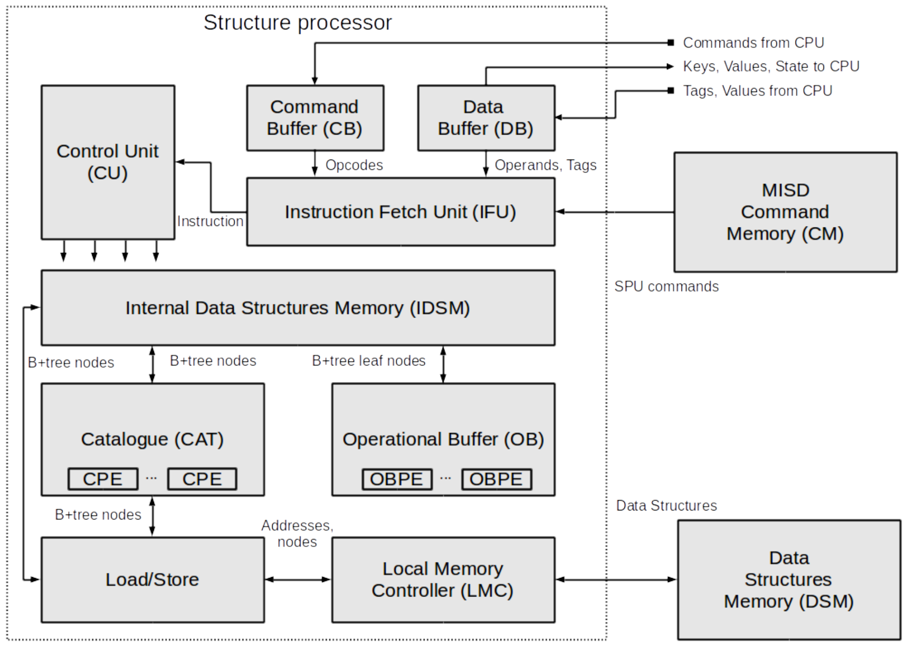


Figure 3. Structure processor microarchitecture

ferent nodes. This type of processing is known as an SIMD, while each CPE is defined as an SISD unit.

Any SPU command begins with a request to the CAT to define the right track. If the stored track is not valid for the new request, CPEs raise missed signals to the CU. After that, the CAT unit begins to route upwards to the tree root to find the right trace, level by level. When the correct sub-tree is found (up to root level in the worst case), CAT starts to load the new trace from DSM by going downwards to the leaves. In parallel with that, the trace loads into the IDSM and OB. Such a procedure significantly accelerates B+tree tracing by combining the caching due to fast memory access and the parallel processing inside CPEs. Experiments show the great acceleration of such hardware processing in comparison with software programming.

Principle 6. *The bottom node of the B+tree consists of a number of key-value pairs (called ”line”) and stores in the Internal Data Structure Memory (IDSM).*

After the full track becomes known, the OB loads the line from the IDSM. It processes the search, insert, and delete operations for the leaves. The most difficult to realize are operations over two or three data structures, such as union, intersect, complement, and slicing. To support these operations, the OB consists of three parts: Buffer A and Buffer B to store data for source structures A and B and Buffer R to store and process the result data structure (named R). The last one is much more functional in comparison with A and B Buffers and allows multiple types of shifting and searching operations to find the right place for the new key or to delete the key and its value. As it is fixed for CAT, all of OB’s buffers also include Operational Buffer Processor Elements (OBPE) to process keys and values in parallel. Due to the fact that all OBPEs execute one instruction over multiple data streams, the OB can be classified as an SIMD unit.

After the leaf is processed in the OB, the R Buffer uploads the result line back on IDSM. If any further processing is needed (for example, if Buffer R overflows or if the next block should be loaded), CAT defines the new trace and then raises the new line loading into the OB. When the whole operation is done, the OB puts result (if supported for that command) to the Data Buffer (DB) and forwards it to the CPU.

The next unique SPU block is Fetch Instruction Unit (FIU). The function of FIU is to fetch commands from the MISD Command Memory (CM) and from the CPU as well. The FIU defines priorities for both instruction streams and check operand readiness. As is shown above, each operand inside the SPU instruction can be tagged by the CPU. If the tag is not valid, it raises the FIU to go to the waiting state.

Principle 7. *The tag is a field in the SPU instruction to indicate the operand validity. Such an instruction can be executed only when all its tags are in the valid state.*

If the instruction is received from the CPU (accelerator mode), then such a command obviously has all operands in the valid state. As for instructions from DSM, all "non-valid" operands can be received from the CPU in any order, but the tag number should be attached. Such a case suggests a new addressing mode for operands, which is called "external addressing".

Principle 8. *External addressing is a mode of operand addressing used to indicate that the operand should be received later from the CPU.*

Theoretical research has shown strong dependencies between sequences of CPU and SPU operations for all the discovered algorithms. One way to have less dependencies is by reusing the previous results or operands. In most cases, those results can be reused as a part of the new compound key (i.e., similar to compound or composite keys in databases). Therefore, we designed the hardware unit to automatically generate new keys from special result registers. Such a technique, as research has shown, significantly reduces dependencies from 80-90% down to 30-50% of the SPU's instructions. We are also introducing the compound register addressing mode to use this hardware function.

Principle 9. *The compound register addressing is the addressing mode used when the new operand should be constructed from the parts of previously defined results.*

The SPU command set is constantly growing due to algorithm requirements. Now, it consists of the 20 high-level machine instructions listed below.

Search (SRCH): The SPU expects the structure number and the key as operands and performs the exact search of this key.

Insert (INS): The SPU expects the structure number and key-value pair and then inserts them into the structure. If the key is already stored, the SPU renews its volume.

Delete (DEL): The SPU performs the exact search for the specified key and removes it from the data structure.

Smaller and Greater Neighbors (NSM, NGR): These instructions search the key that is a neighbor of and smaller (or greater) to the given operand and then return its key and value. The neighbor operation is useful for fuzzy or heuristic computing for many algorithms, such as clustering, aggregation, and interpolation.

Maximum and Minimum (MAX,MIN)): These instructions find the first or last key in the data structure.

Cardinality (CNT): The SPU receives the structure number to find the cardinality.

AND, OR, NOT: These instructions perform union, intersection, and complement operations on two data structures and then put the result into the third one.

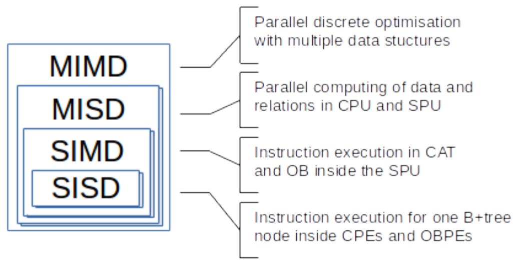

MIMD
MISD
SIMD
SISD
Parallel discrete optimisation with multiple data stuctures
Parallel computing of data and relations in CPU and SPU
Instruction execution in CAT and OB inside the SPU
Instruction execution for one B+tree node inside CPEs and OBPEs

Figure 4. The hierarchical nesting of computer systems architecture on the example of the SPU

Slices (LS, GR, LSEQ, GREQ): These are the instructions to extract the subset of one data structure into another.

Search next and previous (NEXT, PREV): These instructions find the neighbor (next or previous) keys in the data structure from the stored key. In contrast with NSM or NGR instructions, the key should be mandatorily stored in the data structure.

Delete all structure (DELS): This instruction clears all resources used by the given structure.

Squeeze (SQ): This instruction compresses the memory blocks used by the data structure. It can be timely used to low DSM memory fragmentation.

Jump (JT): This instruction branches the SPU code in order to give the CPU control and is available only in the MISD mode. This command includes one address operand and two tags. The first tag indicates the address validity, and the second tag indicates the jump direction. For example:

- The JT(?, ?) command is waiting for both direction tag and jump address with its tag
- The JT(?, @1) command is waiting for the direction tag and jumps to the next command if tag=0 or to @1 address when tag=1
- The JT(1, @2) is the unconditional jump to the @2 address

5. Hierarchical nesting of computer systems architecture

Let us arrive at one important conclusion in this section. As we fixed above, the SPU includes CAT and the OB, which can be classified as SIMD units due to the fact that each of them consists of a number of single SISD processor elements (Catalogue Processor Elements and Operational Buffer Processor Elements). Therefore, we have found at least one example of computer architecture, where the MISD computer includes some SIMD units and every SIMD unit includes a number of SISD units. Also, we can construct the parallel MIMD system with some MISD units. This fact allows us to make a statement about the hierarchical nesting of computer systems architecture (as shown in Figure 4).

Principle 10. *The principle of the hierarchical nesting of computer systems architecture suggests that one MIMD system includes multiple MISD systems, one MISD system includes multiple SIMD systems, and one SIMD system includes multiple SISD systems.*

6. Programming features in MISD system

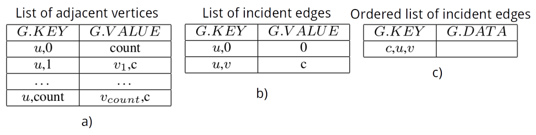
The new computing principles cannot be implemented without changes in the programming technology and algorithms. Like other heterogeneous systems, the MISD computer requires the modification of generic sequential algorithms to the asymmetric multi-threaded version. This could be the significant disadvantage in case of high programming complexity and weak technology readiness. Therefore, it is necessary to develop the simple and convenient technology to take into account the new architecture features. For the proposed architecture, this problem could be solved since most of the optimization algorithms are formed with operations from the SPU's instructions set. Thus, there is no need to choose specific data structures to develop MISD algorithms for a particular problem, because data structures' physical layer is already implemented by the hardware.

The main stages of the methodology to represent the generic algorithm in the MISD version are noticed below.

Stage 1. Data modelling: The algorithm's performance depends on the data models and the complexity of the implemented methods. Thus, it is quite important to effectively represent both of them into the data structure paradigm in order to design the productive program. This process can usually be implemented similar to the implementation of DBMS and consists of three levels: conceptual, logical, and physical. On the conceptual and logical levels, all the significant information for the problem area should be recognized and described in the form of known data formats (data structures, objects, scalar values, etc.). As an example, we will show the graph representation technique on the logical level when the conceptual one is already clarified and the physical level is covered by the SPU hardware. We will show three possible ways to represent the graph.

CASE 1. THE GRAPH $G(V,E)$ REPRESENTATION BY THE ADJACENCY LIST.

Let the algorithm require traversing the graph *G*, for example, through the DFS algorithm [7]. Therefore, the main graph operation is to search those vertices $v \in Adj[u]$, which are specified as incident to *u*, and the subsequent transition to the processing of all associated vertices. Since the vertex degrees are different, it is required to store the number of edges in the data structure.


List of adjacent vertices

$G.KEY$	$G.VALUE$
u,0	count
u,1	v_1,c
$\dots$	$\dots$
u,count	v_{count},c

a)

List of incident edges

$G.KEY$	$G.VALUE$
u,0	0
u,v	c

b)

Ordered list of incident edges

$G.KEY$	$G.DATA$
c,u,v	

c)

Figure 5. Graph $G(V,E)$ representation examples

We will use the "dot" notation to show $STRUCTURE.KEY$ as a key field, $STRUCTURE.VALUE$ as a value, and (..,..,..) as concatenation function. Let the G structure store vertices by their numbers. Then, the $G.KEY$ field stores information about one edge of *u* vertex (*u*, *i*), where *i* is the internal index for the *uv* edge. Therefore, $G.VALUE$ stores the vertex *v* number and its weight *c*. One way to understand a particular vertex degree is to store additional record with a zeroed index. When the edge index is equal to zero (*u*, 0), this record stores the vertex degree *count* $= |Adj(u)|$ (as shown in Figure 5.a).

CASE 2. THE GRAPH $G(V,E)$ REPRESENTATION BY A LIST OF INCIDENT EDGES.

If the algorithm has to find the edge between *u* and *v* vertexes, the data structure can be represented in a different way. The $G.KEY$ field can be compounded by *u* and *v* numbers, while $G.VALUE$ should store the weight *c* of the *uv* edge. To get the list $Adj[u]$ of incident edges, the data structure should store special markers with zeroed indices (*u*, 0) or the neighbor commands can be used (NGR, NSM opcodes). This allows us to find a starting position of the *u* record and then to traverse all others edges with the NEXT instruction (as shown in Figure 5.b).

CASE 3. THE GRAPH $G(V,E)$ REPRESENTATION BY AN ORDERED LIST OF INCIDENT EDGES.

It is often necessary to store a graph with a list of edges in order of their weights. Since the vertices weight in general cases is not the unique value, the compound key $G.KEY$ = (*c*, *u*, *v*) can be used. The $G.VALUE$ field can store any useful information (as shown in Figure 5.c).

Stage 2. Modifying the algorithm: In the next stage, all algorithm blocks should be defined in terms of data structure operations. This also includes appropriate operand specification for SPU instructions and getting the results for the further algorithm steps. In such commands as Search, Delete, Max/Min, Next/Prev and Neighbors, the S2C data queue (Figure 2) will store result keys and values, which can be used by the CPU algorithm. All MISD instructions also change the SPU status register to define whether the execution was successful. As a result of this stage, the modified algorithm can be implemented in the acceleration mode.

Stage 3. Splitting algorithm into threads: To use the MISD mode, the general algorithm should be split into two interrelated threads in order to perform them on the CPU and SPU processors. To ensure that the equivalence of the MISD algorithm results with the generic one, the synchronization primitives should be included in the CPU thread. This could be realized with PUT and GET commands. The addressing mode for every operand should be defined as well. Here, we demonstrate an example of the search command execution.

Data ← SRCH(G, key)		
CPU thread		**SPU thread**
PUT(key)	$\Longrightarrow$	
Data = GET()	$\Longleftarrow$	SRCH(G,?)

As fixed above, such a code can work only in the MISD mode when operand tags could be set into the undefined value "?". When the SEARCH(G, ?) operation is loaded into the SPU, the FIU block checks tags and begins to wait for the second operand from the CPU. The PUT operation on the CPU side sends this operand to the SPU data queue. Following this, the FIU sends the operation and all operands to the execution stage.

7. Algorithm example

We will show a simple example of how the BFS algorithm can be implemented on the MISD system. BFS is a graph-traversing algorithm used to define the sequence of graph vertices in order when current neighbor vertices are exploring rather than the next-level neighbors [7]. We will use the graph $G(V,E)$ representation by the adjacency list (Case 1). Vertex marker (u, 0) will indicate the flag information, as u vertex has been already explored before: $G.KEY$ = (u, i), i = 0..$|Adj[u]|$, $G.VALUE$ = (v, c) when $i \neq 0$, $G.VALUE$ = ($|Adj[u]|$, $flag$) when $i = 0$.

The queue Q is also required to store the vertex sequence in the explored order, so $Q.KEY$ = (j) and $Q.VALUE$ = (u), where j is the auto incremented value. Let the s vertex be the starting point. The following algorithm is ready to be implemented in the MISD acceleration mode.

Algorithm 1 BFS(G,s) /Acceleration mode/

```
j=0                                        ▷Queueing index
INS(Q,0,s)                                 ▷Add initial vertex s to Q
repeat
    cur = MIN(Q)                           ▷Obtain the first from Q
    DEL(Q,cur.KEY)                         ▷Delete it from Q
    t = SRCH(G, (cur.DATA, 0))             ▷BFS use a FIFO
    count = t.VALUE.|Adj[u]|               ▷Define the vertex degree
    for i=1 to count do                    ▷For all edges
        v = SRCH(G,(t.KEY.u,i))            ▷Read next edge
        if v.DATA.flag == 0 then           ▷If v not explored
            j=j+1                          ▷Increment index for Q
            INS(Q,j,v.KEY.u)               ▷Add v to Q
            INS(G,(v.KEY.u,0),(0,1))       ▷Write flag=1 to v
        end if
    end for
until CNT(Q) = 0                           ▷While Q is not empty
```

Now, the algorithm can be split into the CPU and SPU threads. We are using an asterisk (*) to note those SPU commands, which are fully independent from the CPU thread. Commands with two asterisks (**) have dependencies only from SPU results and could be concatenated from keys and values in the compound register addressing mode.

Algorithm 2 BFS(G,s) /MISD mode/

```
/*CPU thread*/                             /*SPU thread*/
j=0
PUT(s)                                     INS(Q,0,?)
repeat
    PUT(1)                                 JT(?,@1)
    cur=GET()                              @1:MIN(Q)*
    PUT(cur.KEY)                           DEL(Q,?)**
    PUT((cur.DATA, 0))
    t=GET()                                SRCH(G,?)**
    A = t.KEY.|Adj[u]|
    for i=1 to A do
        PUT(0)                             @2:JT(?,@4)
        PUT(t.KEY,i)
        v=GET()                            SRCH(G,?)**
        if v.DATA.flag == 0 then
            PUT(0)                         JT(?,@3)
            j=j+1
            PUT(j)
            PUT(v.KEY)                     INS(Q,?,?)
            PUT(v.KEY,0)
            PUT(0,1)                       INS(G,?,?)**
        else
            PUT(1)
        end if
    end for                                @3:JT(1,@2)*
    PUT(1)
until GET() = 0                            @4:CNT(Q)*
PUT(0)                                     JT(?,@1)
```

Here, we can see that 54% (7 from 13) of all SPU commands can be executed independently in parallel with the CPU thread. We also received the same results for some important applications on graphs and networks, such as Dijkstra algorithm to find the shortest path, Ford-Fulkerson algorithm, graph traversal algorithms, Kruskal, and Prim algorithms, and some others.

8. Experiments results

The SPU was implemented on the Virtex FPGA platform with on-chip PowerPC405 CPU. Experiments were carried out to measure the productivity of the implemented MISD principles. We used the first SPU version with the following parameters:

- 32 bits for keys and values
- The maximum number of keys in the structure $2 * 10^6$
- The maximum number of data structures controlled by SPU - 7
- DSM capacity: 256 MB, DDR SDRAM, 64 bits memory bus
- SPU and CPU frequencies: 100 MHz

To compare the effectiveness of MISD algorithms with similar algorithms on other hardware platforms, the soft B+tree implementation has been developed for an embedded Microblaze microprocessor, Intel x86 microarchitecture (Pentium and Core i5 microprocessors), and ARM11 microarchitecture. Due to the significant frequency difference between FPGA and VLSI technologies, we measured clock counts instead of the execution time. This allows us to understand the improvements of the new principles without the technological impact. It is also significant to understand the improvement of the multi-core technology in order to compare it with the MISD architecture. Memory management features of the SPU should be also taken into account. Thus, we used both multi-core and single-core systems to compare them with MISD.

We measured the performance of some basic instructions and compared it with the same instruction realized by the software of other architectures. These tests consist of the loop of single-command execution in order to understand dependencies between cycle count and the data structure elements number. The tests show good performance, which was 3x up to 164x (shown in Figure 6). Then, we implemented the series of algorithms to understand all system performance for some algorithms. Experiments show 1.5x efficiency improvement of the MISD system on Kruskal's algorithm and 2.4x improvement on the Prim's algorithm. As shown in Figure 6, the MISD system reached a significant performance improvement in all experiments.

Along with this, we should note the high efficiency of multi-core processors due to their high parallelism: the performance of 4x cores was only 1.5 times lower than that of the MISD. However, we can also note that only the first version of the SPU was used, which does not contain the conventional cache memory, and many internal processes are far from optimal yet. So the next optimized SPU version is developing now to reach a significantly greater efficiency.

Table 1. Algorithm and operation acceleration for the MISD architecture (measured in clock cycles)

Experiment	Acceleration
Delete (MISD with Microblaze)	164.4
Insert (MISD with Microblaze)	42.7
Search (MISD with Microblaze)	31.4
Delete (MISD with Intel Pentium 4)	22.8
Dijkstra's Algorithm (MISD with Intel Pentium 4)	19.4
Search (MISD with Intel Pentium 4)	15.3
Depth-First Search (MISD with ARM11)	12.9
Breadth-First Search (MISD with ARM11)	12.3
Delete (MISD with Intel Core i5)	11.8
Prim's Algorithm (MISD with ARM11)	10.3
Search (MISD with Intel Core i5)	9.8
Dijkstra's Algorithm (MISD with Intel Core i5)	7.6
Kruskal's Algorithm (MISD with ARM11)	7.8
Insert (MISD with Pentium 4)	5.7
Insert (MISD with Intel Core i5)	3.2
Depth-First Search (MISD with Intel Core i5)	3.2
Breadth-First Search (MISD with Intel Core i5)	3.0
Prim Algorithm (MISD with Intel Core i5)	2.4
Kruskal's Algorithm (MISD with Intel Core i5)	1.5

We also measured the power consumption for MISD system that was about 1.1W. This is 31 times lower than Core i5 power consumption (that is equal to 35W). Structure processor hardware complexity is equal to 1.1M gates, while the PowerPC405 microprocessor requires about 2.5M gates. All the MISD system occupy about 420 times less gates than 4x cores of Core i5.

9. Conclusion

A key outcome of the proposed principles is the new parallelism technology to divide algorithms into parts even if they are sequential or have data dependences. This allowed us to begin the designing of the SPU, programming technology, and algorithms, which are specified for discrete optimization over big data sets.

The MISD system and SPU was implemented on FPGA and successfully verified. A series of experiments show the high efficiency of the proposed principles and hardware architecture. The performed tests demonstrated the acceleration from 1.5x up to 164x times with a significantly low hardware complexity (420 times smaller than Core i5) and power consumption (31 times lower than for Core i5). This allows us to expect the justified implementation of MISD principles for many important solutions:

- in scientific researches for general optimization

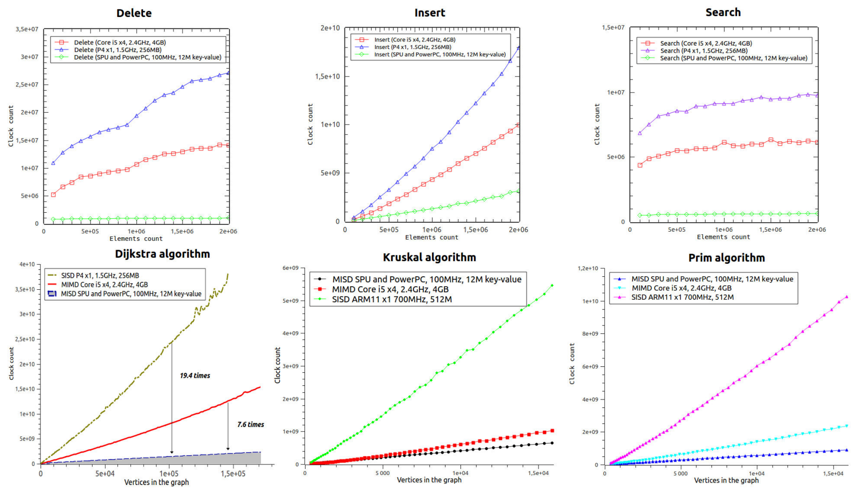


Figure 6. Comparison of the clock counts to execute basic commands and algorithms in the SPU and generic CPUs

problems on graphs, data sets, and data structures

- to plan the robotic systems movement
- for network routing and QoS resource allocation
- to support the Access Control Methods in DBMS
- to accelerate queries in DBMS
- for storage acceleration in enterprise systems
- to accelerate operating systems
- to support CAD systems

The result mentioned above also allows the implementation of the MISD architecture for big data processing. For that purpose, we plan to implement the SPU inside the enterprise OpenPOWER platform with POWER8 CPU. This will allow us to use wide keys (256 bit or more), 16GB and more for the data structure memory, and the IBM CAPI acceleration technology [3]. The other preferable way is to use SPU inside IoT devices to improve their analytical functions.

10. References

[1] P. Malik, "Governing Big Data: Principles and Practices," IBM Journal of Research and Development, vol. 57, no. 3a, 4, pp. 113, 2013.

[2] J. Preshing. (2012, Feb 8). A Look Back at Single-Threaded CPU Performance [Online]. Available: http://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance/

[3] J. Stuecheli, B. Blaner, C. R. Johns and M. S. Siegel, "CAPI: A Coherent Accelerator Processor Interface," IBM Journal of Research and Development, vol. 59, no. 1, pp. 7:17:7, Jan.Feb. 2015.

[4] J. Glossner, P. Blinzer and J. Takala, "HSA-enabled DSPs and accelerators," in 2015 IEEE Global Conference on Signal and Information Processing (GlobalSIP), Orlando, FL, 2015, pp. 14071411.

[5] H. T. Kung and C. E. Leiserson, "Algorithms for VLSI Processor Arrays," in Introduction to VLSI Systems, C. Mead, L. Conway (eds.). Reading, MA: Addison-Wesley, 1980, sec. 8.3.

[6] D. Knuth, The Art of Computer Programming: Volume 1: Fundamental Algorithms, 3rd ed. Addison-Wesley, 1997, p. 672.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, 3rd ed. MIT Press, 2009, p. 1312.

[8] F. Sun and L. Wang, "Performance analysis of B+-Tree and CSB+-Tree in main memory database," in 2014 IEEE Workshop on Electronics, Computer and Applications, Ottawa, ON, 2014, pp. 265268.

[9] A. S. Tanenbaum and T. Austin, Structured computer organization, 6th ed. Prentice Hall, 2013, p. 801.

[10] J. Rao and K. A. Ross, "Making B+-Trees Cache Conscious in Main Memory," in Proc. of the ACM SIGMOD Conference, 2000, pp. 475486.

[11] S. Kumar, A. Shriraman, V. Srinivasan, D. Lin and J. Phillips, "SQRL: Hardware Accelerator for Collecting Software Data Structures," in Parallel Architectures and Compilation Techniques (PACT), 2014.

[12] M. Lavasani, H. Angepat and D. Chiou, "An FPGA-based In-Line Accelerator for Memcached," in IEEE Computer Architecture Letters, vol. 13, no. 2, pp. 5760, Jul.Dec., 2014.

[13] A. Becher, F. Bauer, D. Ziener and J. Teich, "Energy-aware SQL query acceleration through FPGA-based dynamic partial reconfiguration," in Proc. of the 24th International Conference on Field Programmable Logic and Applications (FPL). IEEE Publ., 2014, pp. 18.

[14] B. Sukhwani, M. Thoennes, H. Min, P. Dube, B. Brezzo, S. Asaad and D. Dillenberger. "A Hardware/Software Approach for Database Query Acceleration with FPGAs," International Journal of Parallel Programming, vol. 43, no. 6, pp. 11291159, 2015.

[15] K. D. Hsiue, "FPGA based hardware acceleration for a key-value store database," M. Eng. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, 2014. Available: http://hdl.handle.net/1721.1/91829, accessed 01.11.2015

[16] A. Y. Popov, "The study of the structure processor performance in the computer system with multiple instruction streams and single data stream, Engineering Journal: Science and Innovation, no. 11, 2013. DOI: 10.18698/2308-6033-2013-11-1048.