SiMulPro Core Emulations Show Near Linear Performance Scale-up with Inner Loop Optimization of an O(N^2) Function

Earle Jennings[1]

[1] CTO, QSigma, Inc.
305 Calle Estado, Santa Fe, NM 87501, USA
ewj@ix.netcom.com

Abstract. The SiMulPro® core is a practical non-von Neumann architecture. It is implemented in C++, targeting the Programmable Logic (PL) fabric of a Xilinx SOC, containing the host computer. The core is experimentally confirmed as a Time Division Multiple Access (TDMA) computer. A first and second experiment test an O(N^2) C function (from a standard reference, solving a real world problem). A data memory array, required by TDMA, is implemented in the PL fabric using 128 instances of data memory cells, each performing up to 1 read and/or 1 write on each clock. The data memory array responds to requests from, and delivers results to, the core stripes. Experimental results are shown for N=1,000, 4,000 and 10,000 runs. The first experiment confirms near linear scaling from 1X to nearly 128X, by the core operating a single time slot, in a single stripe, to 128 time slots operating 16 stripes, when performing optimized inner loops. The single time slot, in a single stripe, is comparable to a microprocessor operating without cache, superscalar interpreter, or multi-thread controller. The second experiment extends the time slot decision indicator tag, required by TDMA, for each stripe. This experiment confirms a further 2X performance improvement, with near linear scaling from 1X to nearly 256X. This core implements what is required for C, performs the optimized 3 inner loops of the function with nearly 100% efficiency, and simplifies application code conversion.

Keywords: Non-von Neumann architecture, hardware-software codesign, compiler, FPGA emulation, cache, superscalar interpreter, multi-thread controller, microprocessor, big data, operating system, C, C++, Fortran, TDMA, VLIW, virus, rootkit, core, SOC.

1 TDMA Application to Computer Architectures

TDMA is a staple of communications technology since the 1970's. "In time-division multiple access (TDMA), the available channel bandwidth in its entirety is used by every user, but the users take turns in making use of the channel in a timely manner. In other words, the channel is sequentially time-shared among many users through nonoverlapping time slots in a circular manner (i.e., one after the other)." [14]. Unlike most communication implementations, the SiMulPro core prototype is implemented in the PL fabric. It is partitioned into time slots formed from a strict succession of pipes executed on successive clock cycles. The core uses a single clock signal, operating across the core. The clock is implemented as a single call from the host to the PL fabric interface function named Target_Interface_App, in Fig. 1.

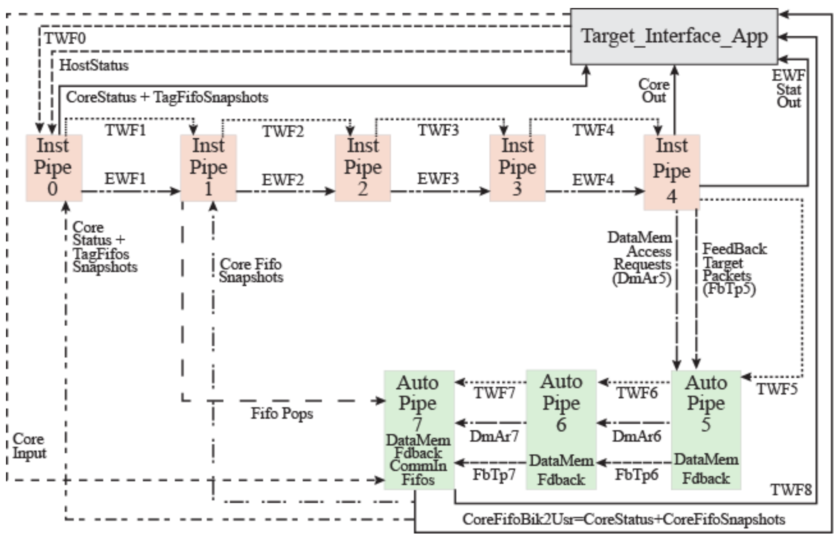
Target_Interface_App
TWF0
HostStatus
CoreStatus + TagFifoSnapshots
Core Out
EWF Stat Out
TWF1
TWF2
TWF3
TWF4
Inst Pipe 0
Inst Pipe 1
Inst Pipe 2
Inst Pipe 3
Inst Pipe 4
EWF1
EWF2
EWF3
EWF4
Core Status + TagFifos Snapshots
Core Fifo Snapshots
DataMem Access Requests (DmAr5)
FeedBack Target Packets (FbTp5)
Auto Pipe 7 DataMem Fdback CommIn Fifos
Auto Pipe 6 DataMem Fdback
Auto Pipe 5 DataMem Fdback
TWF7
TWF6
TWF5
DmAr7
DmAr6
FbTp7
FbTp6
Fifo Pops
Core Input
TWF8
CoreFifoBlk2Usr=CoreStatus+CoreFifoSnapshots

Fig. 1. The TDMA core as a strict succession of pipes.

On the first call to Target_interface_App, Task Wave Front 0 (TWF0) stimulates Inst Pipe 0. On the next call (clock), Inst Pipe 0's response to TWF0 of the previous call generates TWF1 sent to Inst Pipe 1, and so on. When the TWFs command the core to RunProgram, then each of Inst Pipe 0 through Inst Pipe 3 responds with new Execution Wave Fronts (EWFs). Inst Pipe 4 copies its input EWF4 to create its EwfStatOut on the next clock.

Fig. 2 shows the two dimensional organization of the core into pipes and stripes. Each stripe has a pipe component for each of the Pipes 1-7, and acts as a separate channel in the TDMA organization. Each instance of Pipe 1 Stripes 0 to 0xf respond to a single TWF and EWF in one clock shown in Fig. 1. All subsequent data processing instruction (inst) Pipes 2-4 respond to a single TWF and EWF. This continues for the TWF from Auto Pipes 5 to 7.

Time slots encapsulate the latency of response to Process Index Waves (PIWs) described in Fig. 3. There are 7 pipes after Pipe 0, so the response to an issued PIW traverses Pipes 1 to 7 and is processed for rapid response by the Time slot Controllers (TsCtls) in Pipe 0 of each active stripe to generate a new EWF in one clock cycle. Therefore, the response latency to PIW2 is 7+1 clocks, so each stripe needs eight TsCtls. The upper entries with ProcIndex 0 form the first PIW through the Instructed Pipes 0 to 4, the upper entries for Auto Pipes 4 to 7 are the automated response to stimulus initiated by Pipe 4 which affects feedback and local data memory access. The Process Index (ProcIndex) determines the local instruction fetched by each instructed resource of each instructed pipe.

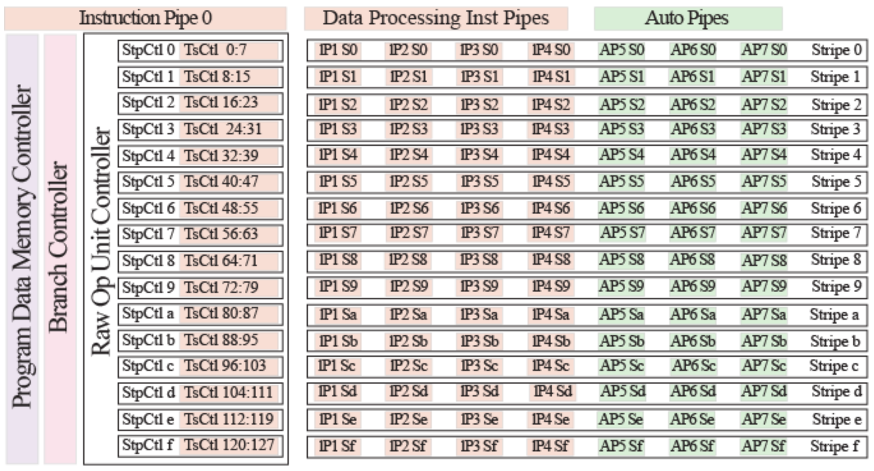
Instruction Pipe 0
Data Processing Inst Pipes
Auto Pipes
Program Data Memory Controller
Branch Controller
Raw Op Unit Controller
StpCtl 0 TsCtl 0:7
StpCtl 1 TsCtl 8:15
StpCtl 2 TsCtl 16:23
StpCtl 3 TsCtl 24:31
StpCtl 4 TsCtl 32:39
StpCtl 5 TsCtl 40:47
StpCtl 6 TsCtl 48:55
StpCtl 7 TsCtl 56:63
StpCtl 8 TsCtl 64:71
StpCtl 9 TsCtl 72:79
StpCtl a TsCtl 80:87
StpCtl b TsCtl 88:95
StpCtl c TsCtl 96:103
StpCtl d TsCtl 104:111
StpCtl e TsCtl 112:119
StpCtl f TsCtl 120:127
IP1 S0 IP2 S0 IP3 S0 IP4 S0 AP5 S0 AP6 S0 AP7 S0 Stripe 0
IP1 S1 IP2 S1 IP3 S1 IP4 S1 AP5 S1 AP6 S1 AP7 S1 Stripe 1
IP1 S2 IP2 S2 IP3 S2 IP4 S2 AP5 S2 AP6 S2 AP7 S2 Stripe 2
IP1 S3 IP2 S3 IP3 S3 IP4 S3 AP5 S3 AP6 S3 AP7 S3 Stripe 3
IP1 S4 IP2 S4 IP3 S4 IP4 S4 AP5 S4 AP6 S4 AP7 S4 Stripe 4
IP1 S5 IP2 S5 IP3 S5 IP4 S5 AP5 S5 AP6 S5 AP7 S5 Stripe 5
IP1 S6 IP2 S6 IP3 S6 IP4 S6 AP5 S6 AP6 S6 AP7 S6 Stripe 6
IP1 S7 IP2 S7 IP3 S7 IP4 S7 AP5 S7 AP6 S7 AP7 S7 Stripe 7
IP1 S8 IP2 S8 IP3 S8 IP4 S8 AP5 S8 AP6 S8 AP7 S8 Stripe 8
IP1 S9 IP2 S9 IP3 S9 IP4 S9 AP5 S9 AP6 S9 AP7 S9 Stripe 9
IP1 Sa IP2 Sa IP3 Sa IP4 Sa AP5 Sa AP6 Sa AP7 Sa Stripe a
IP1 Sb IP2 Sb IP3 Sb IP4 Sb AP5 Sb AP6 Sb AP7 Sb Stripe b
IP1 Sc IP2 Sc IP3 Sc IP4 Sc AP5 Sc AP6 Sc AP7 Sc Stripe c
IP1 Sd IP2 Sd IP3 Sd IP4 Sd AP5 Sd AP6 Sd AP7 Sd Stripe d
IP1 Se IP2 Se IP3 Se IP4 Se AP5 Se AP6 Se AP7 Se Stripe e
IP1 Sf IP2 Sf IP3 Sf IP4 Sf AP5 Sf AP6 Sf AP7 Sf Stripe f

Fig. 2. The TDMA channels of the core of Fig. 1 are the 16 stripes.

Fig. 3 and Fig. 4 have two separate uses based upon two separate interpretations. The first interpretation assumes the first PIW is initiated on the first clock and the second PIW is initiated on the second clock. In this interpretation, within a single Stripe Controller (StpCtl) of Fig. 2, these PIWs originate from successor TsCtls. For example, PIW0 may originate from TsCtl 7 and PIW1 from TsCtl 0 within the same StpCtl.

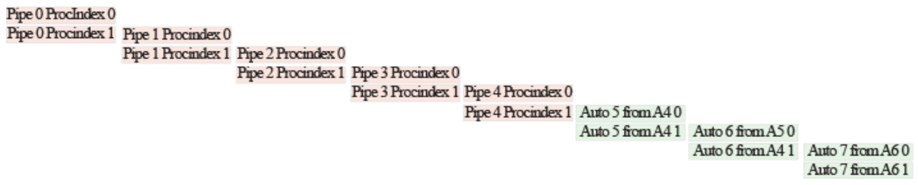

Fig. 3. Shows two Process Index Waves (PIWs) in one stripe, taking 8 clock cycles to traverse the pipes of the core.

The second interpretation of Figs. 3 and 4 assumes that the PIWs originate from the same TsCtl, so they are separated in time by 8 clock cycles. A stripe successively initiates 8 PIWs before the results of the first PIW are available for the next PIW in the same time slot. Programming the time slot is based upon this second interpretation and forms the second use mentioned above.

As shown in Fig. 1, the 8 pipe stages imply the response latency is 8 clock cycles, therefore the number of TsCtls required for the stripe is 8 instances. The Stripe Controller hardware of each stripe successively selects which of its 8 TsCtls generates the stripe's specific EWF component shown in Fig. 6 hereafter.

The TDMA users of the core are Raw Operational (Op) Units, which perform the data processing for a program up to the level of an inner loop. Each Raw Op Unit (ROU) is decomposed into PIWs in at least one time slot of at least one stripe. Programming them can be seen and reported as flat time models of the time slots as shown in Fig. 4.

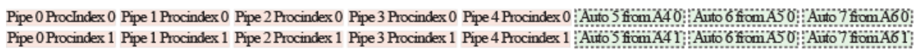

Fig. 4. Shows the flat time description/programming of the PIWs within a time slot.

The ROUs implementing inner loops are decomposed into independent subloops, as the TDMA users of a stripe. Each Stripe Controller in Fig. 2 delegates these users to its TsCtl to time share the stripe. For a given time slot in a stripe executing an inner loop, the stripe's active (selected) TsClt provides the stripe's EWF component with a ProcIndex to access a VLIW instruction residing in each of the instructed resources, and a tag with at least two runtime test bit results. The ProcIndex and tag for the stripe direct the execution of the instructed resources in the PIW and/or the choice of the next ProcIndex by the TsClt at the PIW's completion of the time slot. If the TsClt determines its subloop is done, another test bit result may be used to direct branching by the Branch Controller shown in Pipe 0 in Fig. 2.

Von Neumann computers use branching to affect conditional processing of instructions, which is primary to the semantics of C. TDMA computers affect conditional data processing instructions using a combination of traditional branching, local enable/disable test conditions, and selection of the next PIW to perform within the TsCtl. This TDMA mechanism delivers a large improvement in the bandwidth of computation and local data memory access, which cannot be achieved by von Neumann computers [15]. The body of this paper describes the implementation of the SiMulPro core as a TDMA computer on the PL fabric of the demonstration SOC. The inner loops of the C function are derived and experimentally confirmed as the subloops controlled by the TsCtls and performed by their stripe. The core is experimentally confirmed to provide a near linear speedup of these inner loops enabled by the local data memory in the PL fabric providing up to 128 read and write accesses on each clock cycle.

The first experiment implements the compute intensive inner loop in Fig. 9, and is confirmed to reduce the execution time of the main 2 layer loop runs by a factor of between 95X and about 120X, depending upon N. The bigger N, the closer to 128X. A second experiment confirms reducing the inner loop body from two Process Index Wavefronts (PIWs) in each time slot to essentially one PIW, as shown in Fig. 11. This improves the speedup of the test program runs from 1X to nearly 253X (getting closer to 256X, the larger N becomes), discussed in section 5.

2 Prototype Objectives

The technical objectives are to configure the prototype to run a real world function demonstrating a proof of concept for the following features of the SiMulPro core:

- A TDMA core can be built. This requires automated fixed latency paths for feedback and local data memory access. Hardware determined data availability triggers data processing actions through the generation of EWF components for each successive Time Slot in each active stripe.
- Near linear performance scalability within the SiMulPro core for functions with TDMA optimized inner loops without caches, superscalar interpreters and/or multi-thread controllers.

- Minimal power for given performance: The implemented core, operating in a TDMA manner upon the pipes, delivers essentially 100% performance of the inner loops of the demonstration function. Note that if something is not used (such as floating point in this test function) it can be turned off to minimize power.
- Security: The implemented core, by construction, is immune to infection by viruses and root kits entering from data memory devices, data memories and data operations [9]. Two other avenues of attack are removed in these experiments: The function return stack does not reside in the data memory, cannot be altered by data operations, and is immune to virus and rootkit infection. Function pointers are not supported, because the functions do not reside in data memory.
- Implementing the semantics of C programs [3] is another objective. This enables simplified conversion of existing programs and supports extension to most HPC computing application languages.

The requirements are derived from extending hardware-software codesign to include systems, compiler, application development and emulation engineering insights.

2.1 Systems Engineering Requirements

Von Neumann computer controls are replaced by a TDMA instruction processing mechanism, optimizing the pipe utilization for the inner loops of $O(N^{p>=2})$ C functions [1], [2], & [17]. To achieve this, all pipe components must operate in a strictly sequential manner, without any disruptions, such as distribution of reset, interrupt, or fault signal(s), and without waiting for the (variable) delays of caches, superscalar interpreters and/or multi-thread controllers [8], [15], & [16].

Several time critical system functions need replacing by hardware primitives, removing their fundamental bottlenecks [20] see Section 1.7. Data availability, and at least one completion flag, must be integrated into the hardware to avoid otherwise inevitable operating system delays. Caches, superscalar interpreters, and multi-thread controllers [8] and [16] must be replaced with much simpler hardware, operating far more efficiently, as components required for the TDMA mechanism. A local data memory array is implemented using 128 instances of data memory cells, each able to perform 1 read and 1 write on each clock cycle, accessing a ram of 512 words (of 32 bits). The data memory array responds to requests sent from, and delivered to, the stripes compatible with the TDMA principles. Replacing the superscalar interpreter makes compiler and application development requirements, discussed below, essential. The replacement of multi-thread controllers requires a discussion of simultaneous multi-program implementations of the SiMulPro core, is beyond the scope of this paper.

2.2 Compiler Engineering Requirements

The prototype implements a C function's semantic structure in the hardware configuration. This is achieved for the test program through a series of canonical transformations, targeting a specific compiler internal representation, implemented in the hardware controllers of Pipe 0 of Fig. 2. This is consistent with contemporary compiler technology [4] to [6]. The example C application program is a quick find solution to a connectivity problem found in Sedgewick [2] part 1, pages 11-13 & 20-21, also [16] & [17]. The quick find's first application forms part of the compilation of EQUIVALENCE statements in Fortran compilers [16]. This program is labelled Sedgewick_1_1 hereafter. Sedgewick_1_1 is an O(N^2) function with 3 O(N) inner loops. In listing 1, the inner loop bodies are shown in ***red bold italics***. The second inner loop shown in the listing is the most time consuming. It inhabits the loop body of an O(N) outer loop. This inner loop iterates on reading an indexed component of a vector id[i] and compares that against a value t. If the comparison condition is met, a second value idq is written to that vector component id[i]. It is tested against a final value N before the index increments. The first and third inner loops are optimized in the experiments to access eight consecutive data memory addresses per stripe. In the first inner loop as block writes, and in the third, as block reads, of eight consecutive data addresses. Listing 1 is a conversion from the original C function, and does not require any alteration to the C programming language.

Listing 1. Listing of Sedgewick_1_1.c main program function used in the experiments.

```
typedef unsigned int   Uint;
void input_p_and_q(Uint* p, Uint* q);
void output_id(Uint idi);
Uint i, p, q, j, t, idq, N, L, idi, id[NMAX]; /* variables */
int Sedge_1_1_main(Uint N, Uint L) {
  for (i = 0; i < N; i++) id[i] = i;// first inner loop 1 writing
  for (j=0; j<L;) {        // outer O(N)loop does not count here
    input_p_and_q(&p, &q);
    p %= N;     q %= N;
    t = id[p]; idq = id[q];
    if (t != idq) { j++; // outer loop only increments when this true
      for (i = 0; i < N; i++){
        idi = id[i];  if (idi == t) id[i] = idq; // 2nd main inner loop body
      }
    }
  }
  for (i=0; i<N; i++) output_id(id[i]); // 3rd inner loop reading & output
  return 0;
}
```

2.3 Application Development Engineering Requirements

Application development is the critical path to product deployment, so minimizing the development time is essential. Consequently, application development tools must include the ability to determine internal states, run one pipe and see the results, as well as incremental tests, and examination of the configuration of a program. All of these are required features of the TWF, which provides overall core control. Detailed discussion of these topics is beyond this paper's scope. Other application restrictions: Only unsigned integer data types of 16 and 32 bits are supported. The following are unsupported: Inline functions. Register variables, because registers are unneeded, and register blocks break TDMA rules.

Continue statements are unsupported, and can be converted to if-statements. Switch statements are unsupported, and can be rewritten as if-statements. Goto statements are unsupported, and can be rewritten using returns and encapsulating unstructured code in a function. These heuristic rules can evolve into canonical (compiler) transformations.

2.4 Emulation Engineering Constraints and Requirements

The prototype confirms the hardware objectives, the feasibility of the TDMA constrained pipe structure, and tests a set of canonical transformations required for compiler application support for C. Its emulation is constrained by the C/C++ compiler support and interface requirements of the vendor provided tools. The decision not to use Verilog or VHDL allows initial modeling to be carried with commonly used development language environments running without specialized tools on commonly available computers. The 2017.4 release of the Xilinx development tools for a ZCU-102 demonstration system is used. The implementation source code uses only C language constructs, while using C++ compilers to build the emulations for three reasons. First the Xilinx C++ compilers have a significant constraint in implementing inheritance classes in C++. There can be only one level of virtual member functions, whereas two levels are required to support these functions for instructed resources (1st level) and instruction pipes (2nd level). Experience shows C++ compilation provides better type checking, minimizing problems compiling to the PL fabric. C++ supports const (constant) semantics, enabling multi-level data structure constants. The interface to the core emulation is through the function named Target_Interface_App (Fig.1), which has compiled instantiations in the PL fabric and host computer. Each call of the interface function represents one clock cycle of the core being emulated. There are inputs and outputs to the core which enter or leave the 8 pipe stages. To facilitate development, these inputs and outputs are organized into a flat time model snapshot of the stimulus and response of the PL fabric core on the host side similar to Fig. 4. This flat time perspective is also the basic programming model of execution of one time slot for one stripe.

3 Detailed Definitions and Terms

The experimental SiMulPro core [9-11] operates in the PL fabric of a Xilinx ZCU-102 SOC demonstration system. The core implementation acts as a TDMA computing device with 8 Pipes (Figs. 1-3). Fig. 2 shows the core has 16 Stripes (shown horizontally) in Pipes 1 to 7 (shown vertically), each Stripe with an independent Stripe Controller (StpCtl) in Pipe 0. Each StpCtl contributes to the EWF propagating from Pipe 0 through Pipe 4, which stimulates the Automated Pipes 5 through 7 providing feedback and local data memory access. EwfStatOut of Fig. 1 is a delayed copy of Ewf4 entering Inst Pipe 4 on the previous clock (call of Target_Interface_App).

Fig. 2 shows the instruction processing mechanism implemented by the following in Inst Pipe 0: a Program Data Memory Controller, a Branch Controller, a Raw Operational (Op) Unit Controller interacting with the 16 StpCtl. Each StpCtl contains and interacts with 8 Time Slot Controllers (TsCtl). Fig. 5 shows each StpCtl selects one of it’s TsCtls to inject a ProcIndex, a tag (short 16 bit), and possibly one or more data words (32 bits), into the EWF to control its stripe within the Time Slot.

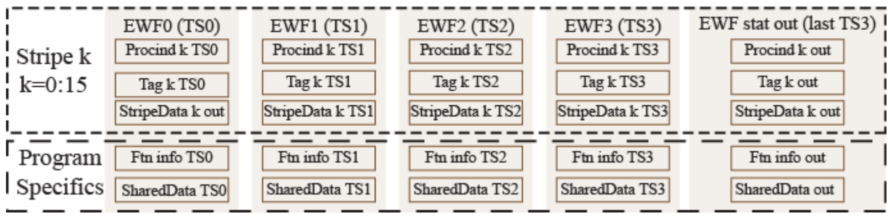

Fig. 5. EWF components for each Stripe k and the shared program specifics.

Each StpCtl interacts with its TsCtls to initiate successive EWFs traversing the stripes in each of the subsequent pipes in a TDMA scheme as a succession of Time Slots (TSs) governed by the ProcIndex and tag in each stripe component of each EWF. The tag designates the TS, and up to 6 bit flags for inner loop implementations of functions occurring in one or more stripes. The flags can be set by the TsCtl generating the EWF component, or by stripe arithmetic resources of Pipes 2 or 3.The ProcInd k is an 8 bit index, with index value 0xff reserved for a noop. Each of the StripeData and SharedData components are 8 (32 bit) words. The Ftn Info (32 bit word) delineates several things, only two are required for these experiments: A Call of Ftn 0 (main program function) starts a program. A return from Ftn 0, ends it.

Fig. 1 shows the core containing a sequence of pipe stages traversed in a fixed sequential mechanism forming two wavefronts. The first is for control and configuration, the TWF, which enters the PL fabric going first to Pipe 0 and then successively traversing all the other pipes. The second originates in Pipe 0 initiating a second, data processing, Execution Wave Front (EWF). The EWF successively traverses Inst Pipe 1 to 4. When the TWF causes a program to run, the EWF contains all information needed to execute the program traversing the pipes, with its bit flags encapsulated in the tag. Before delving into the coding of the main inner loop 2 of the Sedgewick_1_1 program, the instructed and automated resources of Figs 1-3 are shown in Fig. 6. Stripe k in Inst Pipe 1 contains instructed resources Pass1 k, ComIn k, Data Fout k, Tag Fout/Merge k, and Rd Channel (Chan) k. Stripe 0 also contains instructed resources to feedback and pass shared data, Shared Data Fout 1 0 and Shared Pass 1 0.

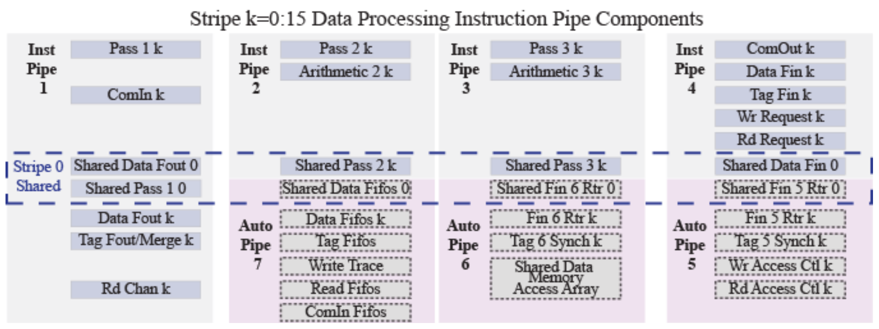

Fig. 6. The instructed resources of Inst Pipe 1-4 and automated resources of Auto Pipes 5-7.

Fig. 7 shows the tag for a Stripe k containing a Tag Time Slot Number, which is set in Pipe 0 and fixed in the PIW progression from EWF 1 to EWF Stat Out. The Tag test part is set by the originating Time slot Controller (TsClt) of its stripe. Individual Tag tests can be altered by instructed resources in Inst Pipes 1 through 3. For example in Stripe k, Tag Fout/Merge k 1, in Inst Pipe 1 can modify one or more tag tests. Arithmetic 2 k, in Inst Pipe 2 and/or Arithmetic 3 k, in Inst Pipe 3 can also modify tag tests.

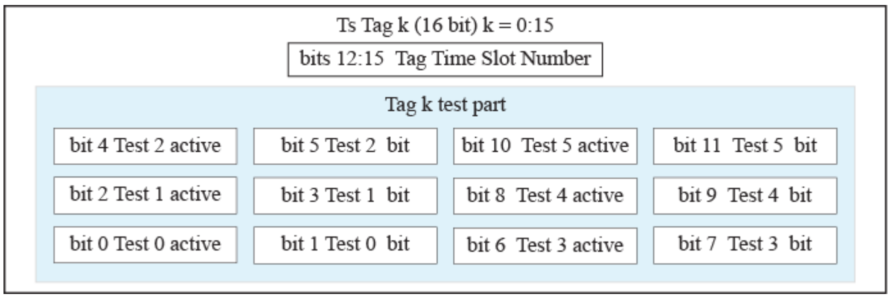

Fig. 7. Shows the Time Slot (TS) Tag for Stripe k of Fig. 5, where it is denoted Tag TS k.

4 Main Inner Loop Coding of First Experiment

Many languages such as Fortran, C, and C++ possess iterative (looping) structures. The C language [3], [21] (sections 1.1 and 1.2) provides a reasonable cross section consisting of three constructs, while-loops, do-while loops and for-loops. A program contains functions which may use any of these constructs. Any loop contains an iterative code body. That iterative code body may include a second loop construct inside itself, etc.. An innermost loop construct in a function is defined to be an **inner loop** of the function. The starting point for the experiments is 2nd main inner loop of Sedgewick_1_1, with the loop body shown in red bold italic font. The main inner loop becomes, through this derivation, one Raw Op Unit (ROU). This paper uses a derivation of the quick-find program 1.1 in Sedgewick [2]. A random number generator Numerical Recipes 3rd edition [1] is used to generate random numbers used in the tests. The particular implementation is from section 7.1.7, the ranlim32 algorithm implemented found on page 357.

Listing 2. The main inner loop of Listing 1 becomes the main ROU.

```
for (i = 0; i < N; i++){idi = id[i];  if (idi == t) id[i] = idq; }
// inner loop 2 body
```

Suppose now that we are going to use NumTS ranging from 1 to 128 to represent the tests to be run for a coding of the main inner loop. Then let TopStripe = NumTS/8. Then the main inner subloops are as follows:

Listing 3. The 2nd main inner subloops for StripeK=0:TopStripe and 0<=TS <NumTS.

```
For(StripeK,(TS=StripeK:min(7,NumTS-StripeK*8) )
  (itmp = TS; i < N; i+=NumTS){
    idi = id[itmp];  if (idi == t) id[itmp] = idq;
}// inner subloop 2 body
```

Listing 4 shows address calculation of the first statement denoted as the add of i to the base address of the id vector (denoted idAdr), creating a temporary value idiAdr. A variable in C, such as i of Listing 1, is required to have a data memory location holding its contents, which is read from data memory when needed, [3] page 62. Instead of the variable, temporary values are used in the calculations of each stripe and Time Slot (TS). Temporary values do not have memory locations where they are read, or where they are stored when changed. Each stripe and TS simultaneously exercises different, temporary values of what in Listing 1 is the i variable. In optimizing this inner loop, the variable i is only written when the loop is done, to comply with C [3]. So i is replaced by itmp[k], a temporary version of i in each TS of each active stripe, Stripe K. The main inner subloop body of Listing 4 shows two memory access operations, the first reads a component of the id vector, and the second may write to that component.

Listing 4. The main inner subloops version 2 for StripeK=0:TopStripe and 0<=TS <NumTS.

```
For(Stripe K,TS)(itmp[K] = TS; itmp[K] < N; itmp+=NumTS){
  idiAdr = idAdr+itmp[K]; idi = *idiAdr; // ProcIndex Wave PIW 0
  if (idi == t)           *idiAdr = idq; // ProcIndex Wave PIW 1
} // iAdr = &i;           *iAdr   = N+1; // ProcIndex Wave PIW 2
// inner subloop 2 body
```

Due to the latency of the core response, and the TDMA use of this to fill each time slot in each stripe, the first PIW 0 couples seamlessly with PIW 1 to form the successively initiated PIWs in the same time slot of each stripe being used. Next, allocate the data portions of the stripe EWF components, the shared EWF data components, the stripe and shared feedback memory access, as well as, the tag tests are completed before specifying what the instructed resources of Inst Pipe 0 to Inst Pipe 4 perform. The automated pipes respond to the instructed resources of Pipe 4. Table 1 shows the allocations for both PIWs 0, 1, and 2, which completes the main inner loop for the program in experiment 1. Experiment 2 merges the actions of PIW 0 into part a and PIW1 into part b of ProcInd 3 (PIW3). PIW 2 is used to terminate the main inner loop in both.

Table 1 shows the following for experiment 1: PIW 0 and 1 perform the main subloop body, and PIW 2 clears the FIFOs and writes the last value of the index i into data memory. Table 1 also shows the PIW 0 and 1 merged into PIW 3 for experiment 2, which reuses PIW 2 to clear the First In First Outs (FIFOs) and write the last value of the index i into data memory. PIW 0 becomes the PIW 3a instruction part. PIW1 becomes the PIW 3b instruction part. These two experiments require an understanding of two distinct branch-like mechanisms discussed below and shown in Figs. 8 to 11.

Experiment 1 uses the initial VLIW coding of TsCtl's VLIW instruction of the main inner subloop. What is described within each Time Slot of each stripe is implemented as two PIWs, PIW 0 feeds PIW 1, after being processed by Pipes 1-7 (8 clock cycles).

When a subloop continues, PIW 1, after 8 clock cycles, feeds the next PIW 0 again, and so on. After the last cycle of the subloop's last iteration, PIW 2 writes the last value of i to data memory to support for-loop C semantics and clear the shared FIFOs, which need to be cleared for continued support of TDMA operations.

Table 1. EWF, Shared data, & Feedback Allocations for PIW 0,1 & 2 of Experiment 1 and for PIW 3 and 2 of Experiment 2. NU = Not Used.

				Experiment	1	2
Allocations	EWF 1	EWF 2	EWF 3	EWF 4	PIW 012	PIW 3 2 ab
Stripe k=0:15 data 0	itmp[k]	itmp	itmp	idiAdr=idAdr+itmp read idiAdr into Rd[k] 6 idq stored at iAdr	0-- -1-	a-- -b-
Stripe k data 6	NU	idi	idi	idi	-1-	-b-
Stripe k data 7	NU	idiAdr iAdr	idiAdr iAdr	idiAdr iAdr	-1- --2	-b- --2
Tag k Test 0 start	Used	Used	Used	Used	0--	a--
Tag k Test 1 last	Used	Used	Used	Used	-1-	-b-
Tag k Test 4 end of loop	Used	Used	Used	Used	-1-	-b-
Tag k Test 5	NU	NU	NU	idi == t	-1-	-b-
Stripe k Rd Chan 0	 pop inject	 idi (EWF 2)		Always Requested Always Used	0-- -1-	a-- -b-
Stripe k Wr Chan 0	 pop			Request by Tag test 5	-1- 0-- --2	-b- a-- --2
Stripe k Feedback 0 idiAdr	 pop inject	 idiAdr	idiAdr idiAdr	idiAdr fed back 0	0-- -1-	a-- -b-
Shared data 0	NU	idAdr iAdr	idAdr iAdr	idAdr iAdr	0-- --2	a-- --2
… data 6	NU	idq itmp[0]	idq itmp[0]	idq write itmp[0] at iAdr	-1- --2	-b- --2
… data 7	NU	t	t	t	-1-	-b-
… Feedback 0 idAdr	peek inject pop				0-- --2	a-- --2
… … 6 idq	peek inject pop				-1- --2	-b- --2
… … 7 t	peek inject pop				-1- --2	-b- --2

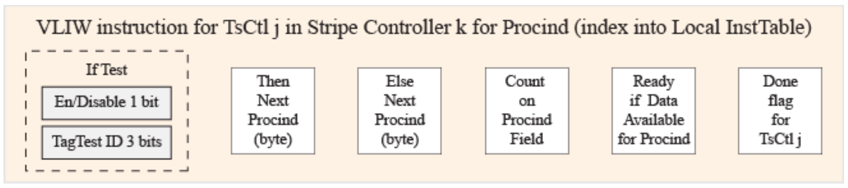

Fig. 8. Shows the VLIW instruction for a Time slot Controller (TsCtl) of a Stripe Controller (StpCtl) k for the ProcInd indexed, local VLIW instruction memory of StpCtl[k] which is distributed to the TsCtl needing it, saving the TsCtl's from each having a local VLIW memory.

The Time slot Controller VLIW format designates an If Test, which may indicate an active Tag Test, controlling which of the two Next ProcInd values follow the current

ProcInd. When the Then Next and Else Next ProcInd fields are the same, then the next process index occurs irrespective of any specified If Test. Fig. 9 and Table 2 show what is required for the three process indexes needed for Stripe 0 and the two for Stripes 1 to 0xf. In Fig. 9, the top row of boxes represents successive PIW 0's with incremental values of itmp. The bottom row of boxes represents successive PIW 1's, which receive results of the PIW 0 delivered through each stripe's automated resources. In the bottom right hand box PIW 2, clears the shared feedback resources of Stripe 0 and stores the final temporary value of itmp at the address of the i variable in data memory.

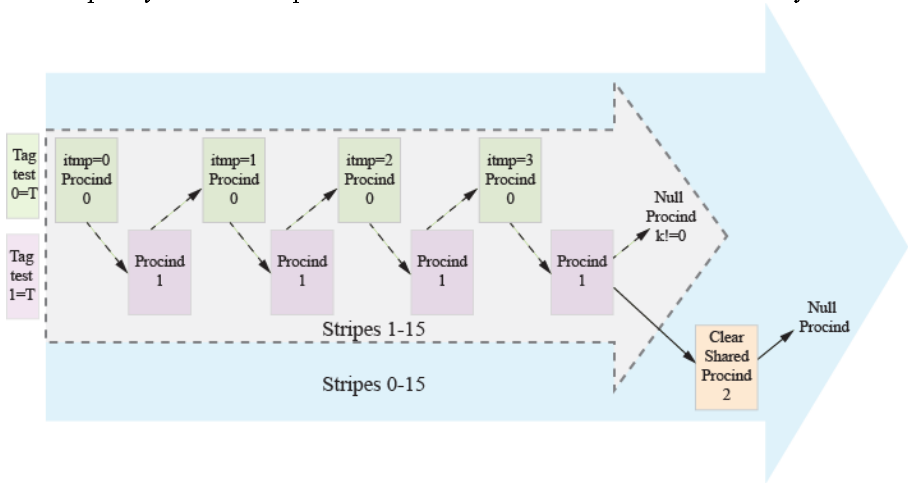

Fig. 9. The active TsCtl outputs for Stripe Controllers k=0 to k=0xf in Pipe 0, output the ProcInd values for 8 PIWs which propagate through Pipes 1-7. This is 4 iterations of the main inner subloop for Stripe k==0 and for any of the other active Stripes k != 0.

The actions performed in Stripe 0 differ from those of the other stripes in the last iteration's last process index generated. The reason for this is that during the execution of this ROU, the shared feedback is used to generate the t, idAdr, and idq temporary values. These are loaded before the start of the main inner loop, so that there is no chance of an access collision while running the main inner loop in the data memory array. At the end of the main loop, the semantic rules of C require the inner loop index (i) be stored in data memory, shown in listings 1-3. To maintain the TDMA condition, all shared temporary values fed back must be cleared from their destination FIFOs. Only Stripe 0 should do this to avoid potential, data memory access collisions.

Table 2. Experiment 1 VLIW instruction summary of ProcInd branching in each active TsCtl

ProcInd (PIW)	ProcInd If Test	Then Next ProcInd	Else Next ProcInd
0	0 (don't care)	1	1
1	Tag Test 4 is end of loop	0	2 (k == 0) Null ProcInd (k != 0) (done)
2 (Stripe k==0)	Null Test	Null ProcInd	Null ProcInd (done)

Referring to Fig. 9 and Table 2, each active Time slot Controller (TsCtl) begins with ProcInd = 0. Whether if test 0 is true or not, the Then Next and Else Next agree as ProcInd (1). The next ProcInd is 1 on the next PIW in each time slot in every active stripe.

The Tag if test for ProcInd 1 is TagTest 4 for all TsCtl in all active stripes.

In Stripe k == 0 for ProcInd 1, the then next ProcInd is 0, and the else next ProcInd is 2. If TagTest 1 is true, the next ProcInd 0 else next ProcInd is 2.

In Stripe k = 1 to 0xf (15) for ProcInd 1, then next ProcInd is 0, and the else next ProcInd is Null ProcInd. If Tag test 1 is true, the next ProcInd 0 else Null ProcInd (the TsClt is done).

In Stripe k == 0 for ProcInd 2, then next ProcInd is Null, and else next ProcInd is Null. Irrespective of If Test, the next ProcInd is null and the TsCtl is done.

Experiment 2 confirms a VLIW instruction requirement for data processing instructed resources leads to a 2X performance improvement in each test run. Each local VLIW instruction of the instructed resources of Pipes 1 to 4 must comply with the format shown in Fig. 10.

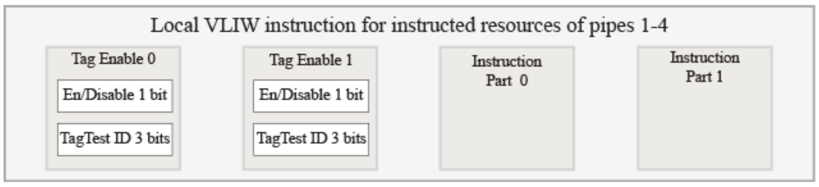

Fig. 10. Each local VLIW instruction has 2 tag enable components controlling separate parts.

Table 3. Description of the effect of each Tag Enable field on its instruction part.

Tag Enable 0			Part 0	Tag Enable 1			Part 1
En/ Disable	Test ID	Test Value		En/ Disable	Test ID	Test Value	
1	0-5	0	NU	1	7	X	NU
1	0-5	1	Used	0	7	X	NU
0	0-5	0	Used	0	6	X	NU
0	0-5	1	NU	1	6	X	Used

Tag Enable fields act independently from each other to control use or non-use (NU) of their respective parts of the VLIW instruction of Fig. 10. For example, Tag Enable 0 uses one of the 6 tag tests shown in Fig. 7 in conjunction with its Enable/Disable field to use, or not use (NU), part 0. When Tag Enable 1 designates test ID 7, which always makes part 1 Not Used. When Tag Enable 1 designates test ID 6, it can always use (Enable =1), or always Not Use (En/Disable =0) Part 1. Each kind of instructed resource may have distinct instruction part components from any other instructed resource of a stripe. For example, Tag Fout/Merge may differ from the other instructed resources of the stripe. Alternatively, Pass 2,k and Pass 3,k may have the same format in some situations, possibly with both parts essentially the same. Also, Arithmetic 2,k and 3,k may have their instruction parts separately tuned, so Part 0 favors local address calculations, and Part 1 does more extensive arithmetic. Fig. 11 shows a straight forward way, with minimal overhead, to multiplex the performance of the PIWs of Fig. 9.

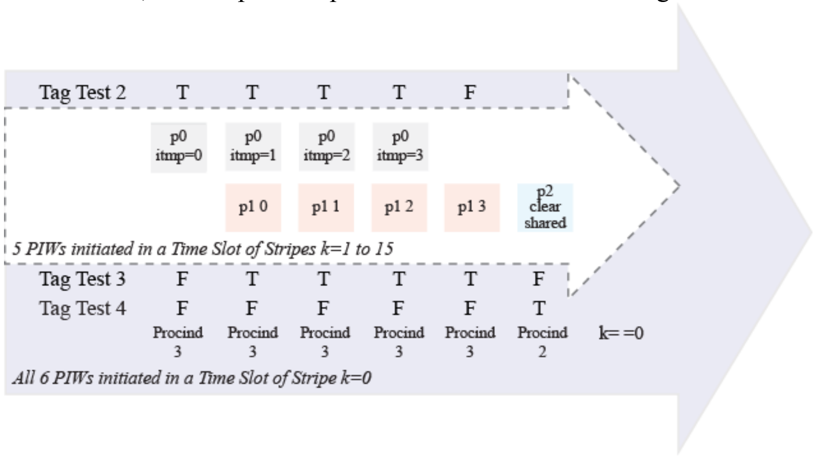

Fig. 11. Shows the performance improvement using the local instruction formats of Fig. 10.

Recall PIW 0 and PIW 1 shown in listing 4. For consecutive iterations, PIW 0 of the first and PIW 0 of the 2nd iteration do not interact, because the memory locations are distinct addresses located in separate data memory cells in the data memory array. A single PIW can integrate the operations of PIW 0 and PIW 1. This requires that the TsCtl's of each stripe initiate Tag Test 0 and 1 as shown here, when executing the ROU of the main inner loop to do the following: On the first iteration, only assert Tag Test 0. On the last iteration, only assert Tag Test 1. In between, assert both Tag Test 0 and 1. How this is implemented is discussed in the next section.

Table 4. Experiment 2 VLIW instruction summary of ProcInd branching

ProcInd	ProcInd If Test	Then next ProcInd	Else next ProcInd
3 (Stripe k=1 to 15) (Stripe k = 0)	4 4	3 3	Null ProcInd 2 (Stripe k==0)
2 (Stripe k == 0)	NU	Null ProcInd	Null ProcInd

5 Instructed Resource Coding for Experiments 1 and 2

Each of the active Stripes k uses the same Shared data, namely the address of the id vector in PIW 0, as well as idq and t in PIW 1. The id[] vector is located in global data memory, but it is often dynamically allocated. Each active time slot calculates idiAdr from itmp, its version of loop index i. An add of idAdr and itmp is performed in Arithmetic 3 k from Table 5, Pipe 3. Assume that Stripe 0 generates and directs the shared program data. While these subloops are running, it is best to avoid any other accesses of the data memory array, to fetch the values of idAdr, idq or t, because these can collide with access of the id[] contents. These are fetched once before the start of the main inner loop and placed into shared feedback for use during the loop. Each feedback targets one of eight First In First Out (FIFOs) for Stripe k=0 to 0xf(15), or the shared FIFO block of 8 FIFOS of Stripe 0. Peeking a FIFO returns its 1st active content, without altering the first position. Popping a FIFO returns the 1st active content, but the FIFO's 2nd active content becomes its new 1st content. If that content is injected, then it enters the pipe's outgoing EWF data for it's stripe. If not injected, the content is thrown away.

Table 5. Inst Pipe coding for PIW 0 for Stripe k=0 to 15 in Experiment 1 becomes the instruction part0 (p0) coding for PIW 3 of Experiment 2.

Pipe	Active Stripe [k] Actions	Temp value
0	Stripe [k] TsCtl[TS] calculates itmp[k] into Ewf1 Stripe[k] Data[0] Generates Ewf1 Stripe[k] ProcInd = 0 (ProcInd = 3 in Experiment 2)	itmp
1	Stripe [k] Pass 1 k passes EWF 1 Stripe[k]Data[0] to EWF 2 Stripe[k]Data[0] Stripe[0] Shared peek Fout 0 injects idAdr into EWF 2 shared Data[0]	itmp idAdr
2	Stripe [k] Pass 2 k passes EWF 2 Stripe[k]Data[0] to EWF 3 Stripe[k]Data[0] Stripe[0] Shared Pass 2 0 passes EWF 2 to EWF 3 for shared Data[0]	itmp idAdr
3	Stripe [k] Arithmetic 3 k Adds itmp (Data 0) to idAdr (Shared Data0) to create idiAdr as EWF 4 (Data 0)	idiAdr[k]
4	Stripe[k] Rd Req port 0 receives idiAdr from Stripe[k] Ewf 4 Data[0] fetching *idiAdr as idi[k] as Rd Chan[k] port 0 on next PIW	 idiAdr[k] idi[k]

Table 6. Inst Pipe coding for PIW 1 for Stripe k=0 to 15 in Experiment 1 becomes the instruction part 1 (p1) coding for PIW 3 in Experiment 2.

Pipe	Active Stripe [k] Actions	Temp Value
0	Stripe [k] TsCtl[TS] Generates Stripe [k] Ewf 1 ProcInd 1	
1	Stripe [k] Rd Chan FIFO 0 pops idi[k] & injects into EWF 2 Stripe[k]Data[6] Stripe[0] Shared Fout 0 peeks idADr, t, idq from FIFO 0, 6,7 into EWF 2 Shared Data[0,6,7]	idi[k] idAdr, t, idq
2	Stripe [k] Pass 2 k passes EWF 2 Stripe[k]Data[6] to EWF 3 Stripe[k]Data[6] Stripe[0] Shared Pass 2 0 passes EWF 2 to EWF 3 shared Data[0,6,7]	idi[k] t, idAdr
3	Stripe [k] Arithmetic 3 k compares for == EWF 3 Stripe[k]Data[6] With EWF 3 Data[6] to form EWF 4 Tag Test[0] = (idi[k] == t) Stripe[0] Shared Pass 3 0 passes EWF 3 shared Data[7] to EWF 5 shared Data[7]	idiAdr[k] TgTest 0 idq
4	If (Tag Test0) Stripe[k] gens Wr Req idiAdr[k] from Stripe[k] Ewf 4 Shared Data[6] of Shared Data[7] (idq) sent to Wr Stripe[k] port 7	idiAdr[k] id[itmp]

Table 7. Inst Pipe coding for PIW 2 for Stripe k=0 used in both experiments.

Pipe	Active Stripe [0] Actions	Temp Value
0	Stripe [0] TsCtl[TS] Generates Stripe [k] Ewf 1 ProcInd 1	
1	Stripe[0] Shared Fout 0 pops from FIFO 0, 6,7 not injected Stripe[0] Pass 1 0 injects iAdr, iLast into EWF 1 Shared Data [6,7]	 iAdr, iLast
2	Stripe[0] Shared Pass 2 0 passes EWF 2 to EWF 3 shared Data[6,7]	iAdr, iLast
3	Stripe[0] Shared Pass 2 0 passes EWF 2 to EWF 3 shared Data[6,7]	i iAdr, iLast
4	Stripe[0] gens Wr Req idiAdr from Ewf 4 Shared Data[6] of Shared Data[7] iLast sent to Wr Stripe[k] port 7	*iAdr = iLast

PIW 2 is used to clear the shared FIFOs used by the two PIW's of Table 2 and 3 for Experiment 1 and the PIW 3 of Experiment 2. The last value of the C variable i is updated by writing its last value to data memory. Note that by planning the allocations appropriately, there is significant code reuse, and a near 2X speedup in performing the main inner loop for any specific number of time slots employed.

Table 8. Inst Pipe coding for PIW 3.

Pipe/ Test	Active Stripe [k] Actions	Temp Value
0	Stripe [k] TsCtl[TS] calculates itmp[k] into Ewf1 Stripe[k] Data[0] See ProcInd discussion in Table 7 Calculates Test2, Test3, Test4	Itmp Tag Tests 2,3,4
1 Tst2	See p0 pipe 1 in Table 2	itmp, idAdr
Tst3	See p1 pipe 1 in Table 3	idi[k], idAdr, t, idq
2 Tst2	See p0 pipe 2 in Table 2	itmp, idAdr
Tst3	See p1 pipe 2 in Table 3	idi[k], t, idAdr
3 Tst2	See p0 pipe 3 in Table 2	idiAdr[k]
Tst3	See p1 pipe 3 in Table 3	idiAdr[k], TgTest 0. idq
4 Tst2	See p0 pipe 4 in Table 2	idiAdr[k], idi[k]
Tst3	See p1 pipe 4 in Table 3	idiAdr[k], id[itmp]

6 Experimental Test Conditions and Results

Sedgewick [2] does not specify a random number generator, which leads to the selection of one from Numerical Recipes [1]. The random number generator was made into a C function Qrand. Qrand was tested running on both Windows 11, Mac OS 10, and the host computer of the Xilinx SOC, the outputs generated from the same seed values were identical for the length of runs in the experiments. In Sedgewick_1_1, N stands for the size of a vector labeled id in Listing 1. Since speedups of a factor of up to 256 were being tested, rather than report results for N = 100, 1000, and 10000, the following values are reported, N=1,000, N=4,000, and N=10,000. To confirm correctness of the calculations, a memory snapshot of the id array was performed after L iterations of the inner loop, where L= 3/4 N, = 750, 3000 and 7,500 for N=1000, 4000, and 10000. The main loop is O(L*N= ¾ N^2)=O(N^2) (See [2] part 1, pg 40 Defn 2.1).

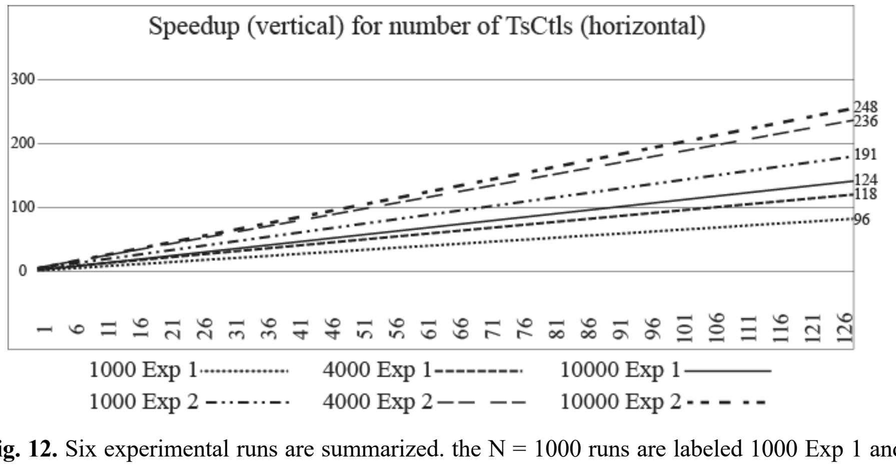

Fig. 12. Six experimental runs are summarized. the N = 1000 runs are labeled 1000 Exp 1 and 1000 Exp 2. The Exp 1 indicates Experiment 1. The Exp 2 indicates Experiment 2.

The program begins in a configured state having loaded the Sedgewick_1_1 interface and PL fabric implementation. The implementation addresses operation from one stripe with one TsCtl of Fig. 2, incrementally filling the stripe with 8 TsCtls, and continuing in this fashion until the 16 stripes are filled with 128 TsCtls.

7 Conclusion

The tested function has simple inner loop bodies, making it possible to detail how that function's inner loops can be optimized for a speedup of between 95X and about 120X in Experiment 1, depending upon N. The bigger N, the closer to 128X. Experiment 2 reduces iterations of the inner loop body from two PIWs in each time slot to essentially one using the VLIW format of Fig. 10, improving the speedup for the test program from 1X to nearly 253X (getting closer to 256X, the larger N becomes).

These performance results are achieved without the need of caches, superscalar interpreters and multi-thread controllers, providing maximum performance with minimal power consumption, [9], [10], [11]. These cores can be extended for all $O(N^{p>=2})$ functions found in Numerical Recipes [1] & [17] and the graph related functions of Sedgewick [2], which have HPC benchmarks [18] & [19]. Most big data programs can also be efficiently converted to run on systems built with the SiMulPro cores, because the semantic constructs of C are found in most, if not all, HPC languages used to implement these programs. These experiments provide evidence that the SiMulPro core is a practical and efficient alternative to von Neumann computers.

By construction, SiMulPro Cores are immune to infection by viruses and rootkits from data memory devices, data memory access and data operations [9]. Further security protection results from removal of the function return stack from data memory and removal of function pointers as discussed above.

References

1. Press, W, Teukolsky, S., Vetterling, W., and Flannery, B.: Numerical Recipes: The Art of Scientific Computing. 3rd edn. Cambridge University Press, Cambridge, UK, (2007).
2. Sedgewick, R.: Algorithms in C, Parts 1-5. 3rd edn., Addison-Wesley Publishing Company, Boston, Mass., (1998).
3. Kernighan, B., Ritchie, D.: The C Programming Language. 2nd edn., Prentice Hall, PTR., Upper Saddle River, New Jersey, USA, (1988).
4. Cooper, K., Torcson, L.: Engineering a Compiler. 2nd edn. Morgan Kauffman, Burlington, Mass, USA, (2012).
5. Waite, W., Goos, G.: Compiler Construction. Springer Verlag, New York. (1984).
6. Wolfe, M.: High Performance Compilers for Parallel Computing. Addison-Wesley Publishing Company, Redwood City, CA, USA. (1996).
7. Hoglund, G., Butler, J.: Rootkits: Subverting the Windows Kernel. Addison-Wesley, Upper Saddle River, New Jersey, USA. 6th printing (2008).
8. Shen, J., Lipoasti, M.: Modern Processor Design: Fundamentals of Superscalar Processors. Waveland Press, Inc., Long Grove, IL. (2013).

9. Jennings, E.: Securing Data Centers, Handheld Computers, and Networked Sensors against Viruses and Rootkits. In: Editor, F., Editor, S. (eds.) 2017 IEEE International Conference on Rebooting Computing (ICRC), DOI: 10.1109/ICRC41351.2017, pp. 1-8. IEEE (2017).
10. Jennings, E.: Core Module Optimizing PDE Sparse Matrix Models with HPCG Example, published with open access at SuperFri.org, DOI: 10.14529/jsfi170205, Supercomputing Frontiers and Innovations, https://superfri.susu.ru/index.php/superfri/article/view/136/231, last accessed 2021/11/26, (2017).
11. Jennings, E.: The Simultaneous Transmit And Receive (STAR) Message Protocol, published with open access at SuperFri.org, DOI: 10.14529/jsfi170204, Supercomputing Frontiers and Innovations, https://superfri.org/index.php/superfri/article/view/135, (2017).
12. van Leeuwen, J., Tarjan, R.: Worst-case analysis of set-union algorithms, Journal of the ACM, Vol. 31, No. 2, 245-281, (1984).
13. Hopcroft, J., Ullman, J.: Set-merging Algorithms, SIAM J. Comput. 2(4), 294-303, (1973).
14. Grami, A.: Introduction to Digital Communications, 2016 cited by Time Division Multiple Access - an overview | ScienceDirectTopics, https://www.sciencedirect.com/topics/engineering /time-division-multiple, last accessed 2021/11/15.
15. Sterling, T.: "Declaration of Interdependence through Non von Neumann Architecture", 2020/7/22,https://www.hpcwire.com/2020/07/22/a-declaration-of-interdependence-through -non-von-neumann-architecture/, last accessed 2021/11/15.
16. Patterson, D., Hennessy, J.: Computer Organization and Design: The Hardware/Software Interface (RISC-V ed), Morgan Kauffmann, an imprint of Elsevier, Cambridge, MA. (2018).
17. Press, W, Teukolsky, S., Vetterling, W., and Flannery, B.: Numerical Recipes in C: The Art of Scientific Computing. 2nd edn. Cambridge University Press, Cambridge, UK, (2002), downloaded version last accessed in 2017.
18. Graph 500 Steering Committee, Graph 500 Benchmark Specification, https://graph500.org /?page_id=12, last accessed 2021/11/26, (2017).
19. Heroux, M., Dongarra, J., and Luszczek, P., HPCG Benchmark Technical Specification. United States: N. p., 2013. Web. doi:10.2172/1113870. https://www.osti.gov/biblio/1113870-hpcg-benchmark-technical-specification, last accessed 2021/11/26, (2013).
20. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J., MPI-The Complete Reference, Vol. 1. The MPI Core, 2nd edn, MIT Press, Cambridge Massachusetts, (1998).