Fine-Grained Instruction Placement in Polymorphic Computing Architectures

David Hentrich, Erdal Oruklu, Jafar Saniie
Illinois Institute of Technology

***Abstract*—This paper presents two computer architecture mechanisms that can be employed together to allow programs to be written where their individual instructions can (1) be executed in parallel in separate processors and (2) be arbitrarily assigned to be executed by any processor in a connected fabric of processors. This is an emerging parallel execution technique for future high performance and embedded computers. Changing the placement of a program's instructions in a processing fabric is equivalent to rearranging the computer architecture around the program itself. Reasons to change the computer architecture around the software are to improve overall execution speed and to perform load balancing. The two computer architecture mechanisms are (1) a custom dataflow instruction set and (2) a processor block called an operation cell. The most important feature of the instruction set is the high level of operational independence of the instructions (i.e. no shared memory/registers between instructions). The operation cell is a processor construct that stores an instruction and manages the execution life of that instruction. An operation cell either executes the instruction itself or forwards the execution to another operation cell in a different processor.**

***Index Terms*—Dataflow Instruction Sets, Computer Architecture, Microprocessors, Parallel Machines, Parallel Architectures, Reconfigurable Architectures, High Performance Computing (HPC), Polymorphic Computing**

I. Introduction

New strategies are needed to continue the performance scaling of high performance systems and embedded systems. One such mechanism is writing programs where the machine instructions can execute in parallel and can be arbitrarily placed/migrated on any processor in a connected fabric of processors. This paper presents a strategy to create programs that expose instruction level parallelism and allow those instructions to be individually placed ("draped") across a collection of processors. This strategy is employed as the cornerstone of a recently proposed polymorphic computing architecture [1] [2]. "A polymorphic computer is a computing machine that can dynamically arrange the underlying hardware computing architecture model in both time and space to match the computational demands of the moment." [3]

This paper presents two computer architecture mechanisms that together allow fine-grained placement of instructions in a computing fabric. These mechanisms are a custom dataflow instruction set [4] [5] [6] [7] [1] [2] coupled with processor block module called an operation cell [1] [2] [8]. After that, the overall instruction/migration mechanism across a collection of processors is discussed. At the end, performance results of a 1024 integer parallelized bubble sort algorithm that is implemented using these mechanisms are presented.

II. Dataflow Instruction Set

The first step to creating a program that can have its instructions individually migrated to separate processors is to have an instruction set architecture that has minimal dependencies on the computer architecture and the surrounding instructions. The vast majority of today's commercially produced instruction sets have the following dependencies that must be eliminated to enable instruction migration:

1) Memory dependencies,
2) Register dependencies, and
3) Implicit instruction program position execution flow dependencies.

The above dependencies are eliminated by utilizing a dataflow instruction set architecture [4] [5] [6] [7]. A dataflow instruction set architecture is a separate class of instructions set architectures from the relatively well known CISC (Complex Instruction Set Computer) [9], RISC (Reduced Instruction Set Computer) [10] [11] [12], and VLIW (Very Long Instruction Word) [13] [14] instruction set architectures. Dataflow instruction set architectures are distinguished from CISC, RISC, and VLIW by:

1) **No memory or register accesses.**
2) **No implicit execution of an instruction based on its position within a program or a program counter.**
3) **Instructions execute in data arrival order.** Instructions are executed when their data is available, not by instruction order in the program. This means that instructions may execute in parallel with one another when their input data is fully available.

In a recently proposed polymorphic computing architecture processor core [1] [2], a new dataflow instruction set was presented with the following relevant properties as discussed in [1]:

1) Each individual instruction can perform the roles of all the traditional dataflow instruction types (operators, deciders, T-Gates, F-Gates, merges, and boolean operators [4]) and the Wavescalar dataflow steer instruction [15].
2) Each instruction can execute two operations simultaneously, but only one may selected for output.
3) Each instruction can utilize and generate predicates. [16] Predicates are additional bits of decision/control information that are transmitted along with a piece of data in an inter-instruction flow of data [17] [18] [19].
4) Operations and decisions can be simultaneously executed in an atomic execution.

978-1-7281-3320-1/20/$31.00 ©2020 IEEE

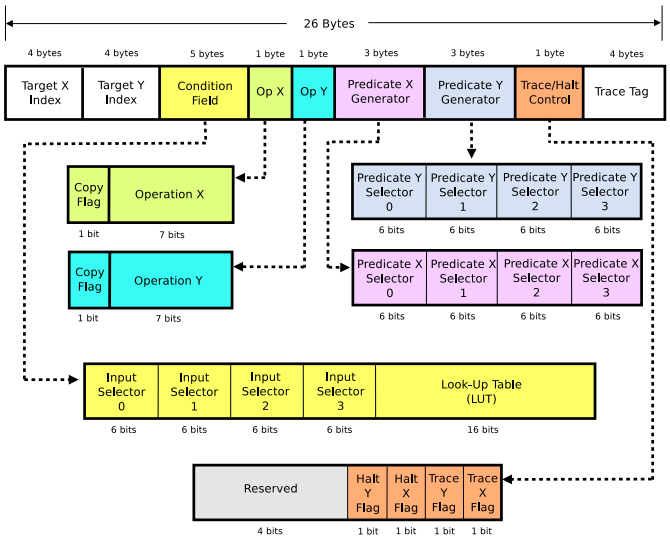

Fig. 1. Instruction Binary Format. Figure copyright ©2019 by David Hentrich. Used with permission.

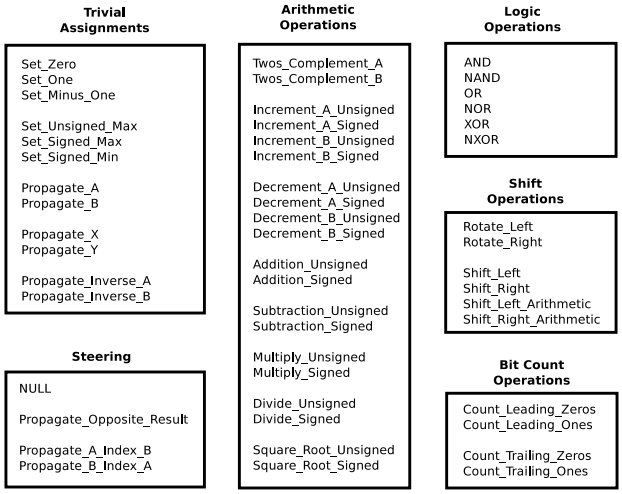

Fig. 2. Executable X Operations. Figure copyright ©2017 by David Hentrich. Used with permission.

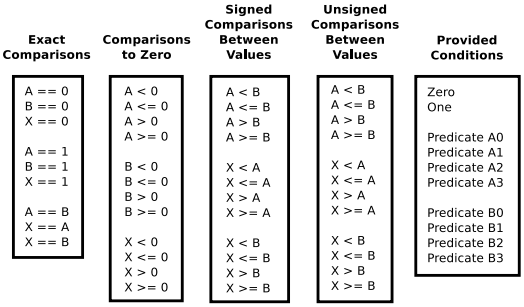

Fig. 3. Conditions. Figure copyright ©2019 by David Hentrich. Used with permission.

5) Instructions results can be routed based on decisions made during atomic execution.

The fundamental unit of transaction in the dataflow instruction set is a 3-tuple composed of a *target*, a *value*, and a set of *predicates*. A *target* specifies the address of an input into a particular instruction. There are two input locations for each instruction (A and B). Absolute addresses 0 and 1 are the special NULL (data discard) locations. A *value* is the primary piece of data in a transaction. The last field is the *predicates*. These are a set of boolean values that can be used to pass arbitrary flags between instructions.

The current instruction binary format is shown in figure 1. The instruction fields are:

1) **2 operations fields:** Each instruction performs two independent operations (X and Y) on each atomic execution. Each operation receives the same two inputs: the A and B inputs of the instruction. Only the result of one operation is selected for output. The other operation result is discarded. If the X operation result is selected for output, it is sent to the X target. If the Y operation result is selected for output, it is sent to the Y target. Note that there is a "copy bit" in each operation. If this bit is set and that operation is selected, that operation's output is sent to both the X and Y targets. In our implementation, the physical operations available to the X operation are shown in figure 2 and the Y operations are a physical subset of the X operations (i.e. the trivial assignment operations, the NULL operation, and the PROPAGATE_OPPOSITE_RESULT operation).
2) **1 condition field:** The condition field is used to determine if the result of the X or the Y operation is selected for output. The field itself is composed of 4 input selector sub-fields and a 16-bit lookup table (LUT) sub-field. Each selector is configured with one of the conditions shown in figure 3 and produces a binary result. Together, the results of the 4 input selectors form a 4-bit result (i.e. 16 possible combinations). Each possible 4-bit result maps to one bit in the lookup table sub-field. If the corresponding lookup table bit is 1, then the X operation result is output by the instruction. If the corresponding bit is 0, then the Y operation result is output.
3) **2 4-bit predicate generation fields:** Predicate generation fields are used to generate the flags that are transmitted along with each instruction output. Each operation is paired with its own predicate generation field. Each predicate bit is determined by configuring its corresponding sub-field with one of the configurations shown in figure 3.
4) **2 target fields:** A target field specifies where an instruction output will be sent. The X operation and the Y operation each have their own independent targets. i If an operation that has been selected for output has its copy bit set, it will be sent to both specified targets. Note that there is an additional mechanism associated with the X target that allows the X result to be sent to a variable location.
5) **1 trace/halt control field:** There are two debug actions

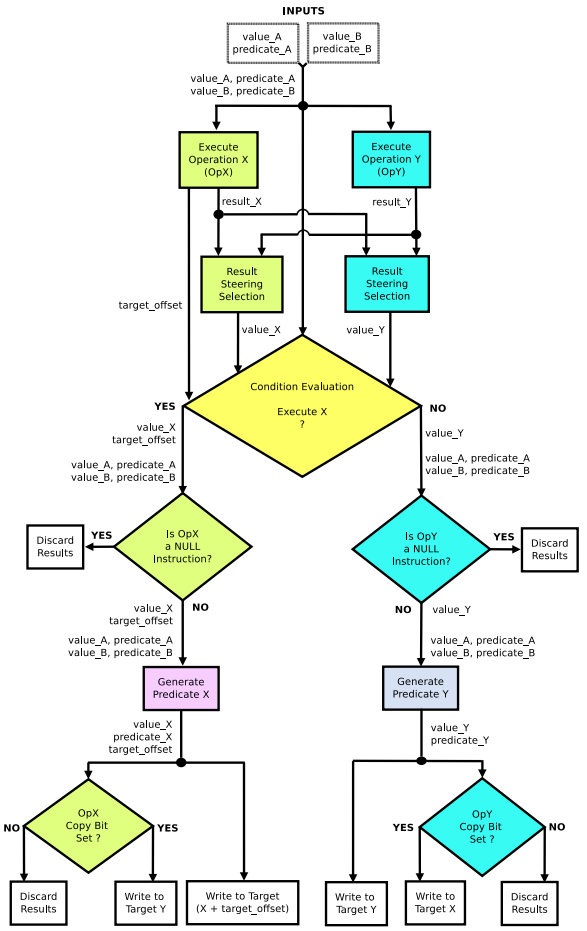

Fig. 4. Instruction Execution Flow Chart. Action colors correspond to instruction field colors in Figure 1. Figure copyright ©2019 by David Hentrich. Used with permission.

that can be taken: tracing and halting. Tracing and halting are independently selectable for each instruction operation. When an operation is selected for output and its trace flag is set, the instruction's trace tag is broadcast on a processor trace bus. When an operation is selected for output and its halt flag is set, the processor core is halted.

6) **1 trace tag field:** The trace flag field is a user defined value that is broadcast when a selected operation's debug trace flag is set.

The full atomic execution flow chart for an instruction is shown in figure 4.

III. Operation Cell Construct

The second step to creating a program that can have its instructions individually migrated to separate processors is to create a hardware processor block to manage an instruction. The block is called an operation cell [1] [2] [8] and is shown in figure 5. An operation cell independently stores an instruction and manages its state during the life of a program. An operation cell performs the following tasks:

1) Stores an instruction

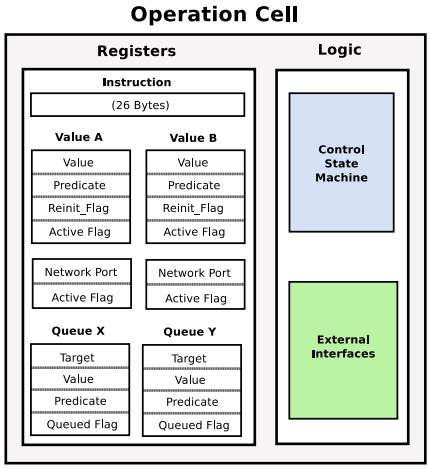

Fig. 5. Operation Cell. Figure copyright ©2019 by David Hentrich. Used with permission.

2) Stores/manages the inputs to an instruction
3) Provides a way to initialize input values
4) Provides a way to automatically re-initialize inputs values that are constants after that input value is consumed by an instruction execution
5) Stores instruction execution outputs (i.e. queues outputs) when the actual targets are full and can not accept the outputs at the time of execution
6) Manages the full life cycle of an instruction execution (i.e. decides when an instruction is ready to execute, causes the instruction to be executed, and manages the completion of the instruction execution)
7) When the operation cell's instruction is configured to run on a different processor, it forwards the inputs to the equivalent operation cell on the different processor that is configured to execute the instruction

It is this last point (forwarding) that is the enabling feature that allows instructions to be individually migrated around a processing array.

IV. Instruction Placement/Migration in a Processing Fabric

By combining the two mechanism above, instructions in a program can be individually placed/migrated across a connected set of processors.

The high level computing environment strategy is to provide an interconnected fabric of identical, operation cell based processors. The connections can be both direct and indirect. In the indirect case, a connection can hop through one or more processors. When a program is loaded into this fabric, each program instruction is loaded into the same operation cell in each processor (i.e. the instruction loaded into operation cell 3 in one processor is also loaded/assigned into operation cell 3 in every other processor in the fabric). The set of operations cells that are associated with the same instruction is called a *slice*. For each instruction, only one operation cell (i.e. only one processor in the interprocessor slice) is

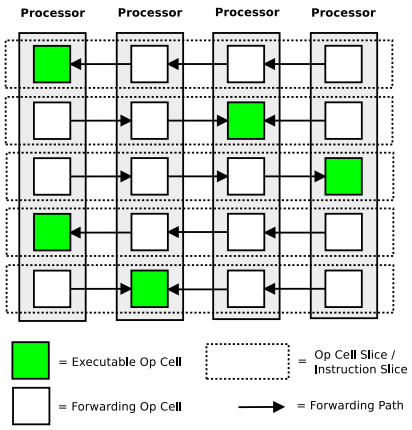

Fig. 6. Example showing a program that has been "draped" across 4 processors. Figure copyright ©2019 by David Hentrich. Used with permission.

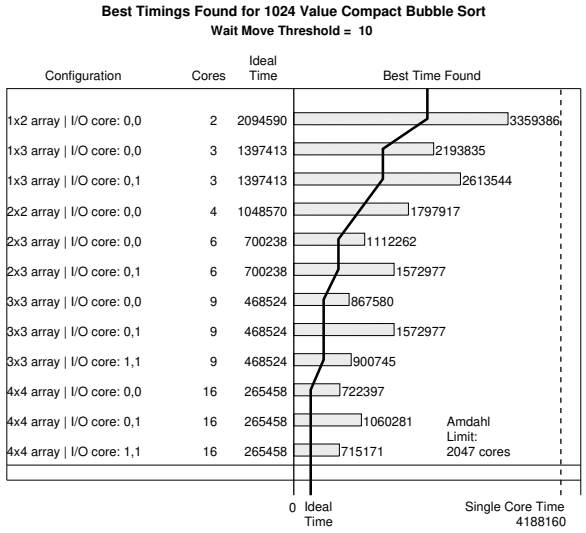

Fig. 7. Results. Figure copyright ©2019 by David Hentrich. Used with permission.

designated to execute that instruction. All the other operations cells for that instruction across all the other processors (i.e. all the other operation cells in a slice) are configured to forward their inputs to the operation cell in the processor that executes the instruction. In this scheme, when pieces of data for a particular instruction appear in one particular processor, they always fall to an operation cell assigned to that instruction in that processor. Essentially, data falls into a slice and the slice manages which processor will actually execute the instruction. Note that no single instruction needs to know in what processor any other instruction is designated to execute.

Changing the positions of a program's instructions across an array of these processors (i.e. "draping") is equivalent to changing the computer architecture under the program [1] [2]. In this scheme, the program to be executed itself is not aware that there is more than one processor. Figure 6 shows a single program whose instructions are assigned throughout a processing array.

V. Results and Conclusion

A 1024 integer parallelized bubble sort algorithm was written in the above instruction set and exercised in several different polymorphic computing array configurations. These configurations were composed of processors that were built around the above operation cells that execute the above instruction set. The exercises were conducted in a custom, cycle-accurate computer architecture simulator under ideal conditions (i.e. all discrete simulation actions such as instruction execution and cross-processor communication occur in a single clock cycle). In all exercises, the program was constant. The only things that changed from exercise to exercise were the computer architecture and the placement of the program in that computer architecture.

The following exercises were performed on the 1024 integer bubble sort program and are illustrated in figure 7. All performance measurements are times expressed in clock cycles.

1) **The baseline single core performance of the algorithm was measured.** The program was placed in a single processor with a single arithmetic logic unit. The program executed in 4,188,160 clock cycles.
2) **The ideal parallelism potential of the algorithm was measured.** In this evaluation, only a single processor core was exercised where the processor resources (i.e. the number of arithmetic logic units) were incrementally increased. This single processor core configuration is considered "ideal" because all discrete actions take place in a single clock cycle and there are no latencies due to interprocessor communication. The "ideal" execution times of the bubble sort algorithm that are associated with each processor array model (discussed below) are shown in figure 7 in tabular form and in the form of an "ideal time" line. The Amdahl Limit of this algorithm (i.e. the maximum number of cores that can be used simultaneously to improve the performance of the algorithm) is 2047 cores. Adding cores beyond this number results in no further timing improvements.
3) **The program was placed in several different polymorphic processor arrays and the execution time was measured in each configuration.** The best program instruction placements that were found in each configuration are shown in figure 7. The placements were found using an iterative placement algorithm that moved instructions to the least-loaded adjacent processor if their wait time (i.e. the number of clock cycles between when an instruction became ready to execute and when the processor's arithmetic logic unit became available) was 10 clocks or greater (i.e. the "wait move threshold").

In figure 7, notice that placement configurations of the 1024 integer bubble sort algorithm in processor arrays where interprocessor communication latencies exist all generally show execution performance improvements as the number of cores is increased.

ACKNOWLEDGMENTS

Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation.

REFERENCES

[1] D. Hentrich, E. Oruklu, and J. Saniie, "A dataflow processor as the basis of a tiled polymorphic computing architecture with fine-grain instruction migration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 10, pp. 2164–2175, Oct 2018.

[2] D. Hentrich, "A polymorphic computing architecture based on a dataflow processor," Ph.D. dissertation, Illinois Institute of Technology, May 2018.

[3] D. Hentrich, E. Oruklu, and J. Saniie, "Polymorphic computing: Definition, trends, and a new agent-based architecture," *Circuits and Systems*, vol. 2, no. 4, pp. 358–364, Oct. 2011.

[4] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," *Proceedings ofthe 2nd annual symposium on Computer Architecture (ISCA '75)*, pp. 126–132, 1975.

[5] J. Silc, B. Robic, and T. Ungerer, *Processor Architecture: From Dataflow to Superscalar and Beyond*. Springer-Verlag, 1999.

[6] S. G. Shiva, *Pipelined and Parallel Computer Architectures*. Harper-Collins, 1996.

[7] A. H. Veen, "Dataflow machine architecture," *ACM Comput. Surv.*, vol. 18, no. 4, pp. 365–396, Dec. 1986. [Online]. Available: http://doi.acm.org/10.1145/27633.28055

[8] D. R. Hentrich, "Operation cell data processor systems and methods," US Patent Application 15 844 810, Dec. 18, 2017.

[9] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. Morgan Kaufmann Publishers, 2007.

[10] D. A. Patterson, "Reduced instruction set computers," *Commun. ACM*, vol. 28, no. 1, pp. 8–21, Jan. 1985. [Online]. Available: http://doi.acm.org/10.1145/2465.214917

[11] P. Chow, "RISC-(reduced instruction set computers)," *IEEE Potentials*, vol. 10, no. 3, pp. 28–31, Oct 1991.

[12] P. Wallich, "Toward simpler, faster computers: By omitting unnecessary functions, designers of reduced-instruction-set computers increase system speed and hold down equipment costs," *IEEE Spectrum*, vol. 22, no. 8, pp. 38–45, Aug. 1985.

[13] J. A. Fisher, "The VLIW machine: A multiprocessor for compiling scientific code," *Computer*, vol. 17, no. 7, pp. 45–53, July 1984.

[14] J. A. Fisher, P. Faraboschi, and C. Young, "VLIW processors: once blue sky, now commonplace," *IEEE Solid-State Circuits Magazine*, vol. 1, no. 2, pp. 10–17, Spring 2009.

[15] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers, "The WaveScalar Architecture," *ACM Transactions on Computer Systems (TOCS)*, vol. 25, no. 2, May 2007.

[16] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley, "Dataflow Predication," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39)*, Orlando, Florida, Dec. 2006, pp. 89–102.

[17] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975.

[18] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. mei W. Hwu, "A comparison of full and partial predicated execution support for ILP processors," in *Proceedings The 22nd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure Italy, Jun. 1995, pp. 138 – 149.

[19] P. Y. T. Hsu and E. S. Davidson, "Highly concurrent scalar processing," in *Proceedings of the 13th annual international symposium on Computer architecture*, Tokyo, Japan, Jun. 1986, pp. 386–395.