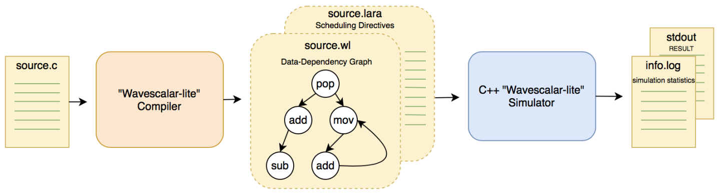
LARA: Optimized Dataflow Instruction Scheduling on a custom Compilation and Simulation Platform

***Abstract*—In this work, we explore optimized instruction scheduling for dataflow computer architectures [2], and propose a compiler-driven algorithm, 'LARA', based on widely-deployed liveness analysis and register allocation algorithms. To evaluate this scheduler and other canonical techniques, we constructed a complete compiler and architectural simulator for a custom variant of the WaveScalar [1] architecture, called WaveScalar-Lite. With this framwork, ee show that LARA has tangible benefits when compared to existing schemes. In addition to analyzing our primary results, we evaluate our extensive WaveScalar-Lite infrastructure and highlight it's potential utility for other research groups.**

***Index Terms*—Dataflow, Instruction Scheduling, Architecture Simulation, WaveScalar, Compilers, Register Allocation**

System Architecture. The WaveScalar-Lite system architecture consists of a complete compiler and a cycle-accurate architectural simulator. The compiler takes as input a C program and outputs a custom assembly format that can be executed directly by the simulator. For the LARA algorithm, the compiler also outputs additional scheduling directives to be consumed by the simulator. The simulator executes our custom WaveScalar-Lite binary format and collects relevant statistics.



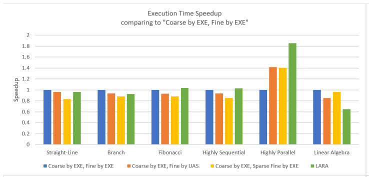
Compiler Architecture. In the compiler frontend, the C source program and header files are tokenized by the parser and lexer modules using the Menhir OCaml parser generator. After performing typechecking, the internal representation (IR) goes through a series of elaboration steps which consolidates C constructs. After elaboration, a control flow graph (CFG) is constructed which separates the linear program into basic blocks connected by control flow edges.

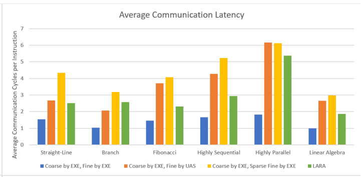
The WaveScalar-Lite backend starts by folding the program into SSA form. SSA form requires that temporaries only be written a single time in the program. Dataflow dependency logic is vastly simplified when each value has a single producer, and this also allows our architectural simulator to forward values greedily without being concerned about redefinitions. Formatting an arbitrary program into SSA form requires the insertion of specialized ϕ instructions [5], which merge renamed temporary values from across branches. We compute a dominance frontier [6] for each basic block and apply known iterative techniques to infer optimal ϕ placement [7]. After SSA conversion, *Waves* are formed by coalescing maximal groupings of basic blocks that do not contain internal loops; In essence, these are feed-forward segments of the program. Dataflow dependencies are then extracted by tagging each instruction with the consumers of the value produced, and then folding the CFG into a dataflow tree. This dataflow tree is linearized and unparsed as WaveScalar-Lite assembly.

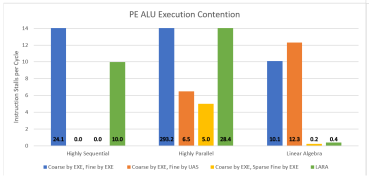
Simulator Architecture. The WaveScalar-Lite architecture is structured similarly to that in the original WaveScalar paper, and consists of tiled processing elements (*PE*s) placed in arrangements of *Pod*s, *Domain*s and *Cluster*s. PEs individually are structured similarly to a 5-stage pipelined RISC-V processor. To execute a program, we begin by setting a global timestamp to 0 and initialize all ready instructions with this value. The non-zero flag value signifies to the PEs that these instructions can execute, and thus they will fire and forward their values once the input operands are satisfied. When this data is forwarded, they are forwarded as (data, flag) pairs, where the flag acts as a timestamp on the data. As soon as the branch to the next wave is resolved, we increment the timestamp and progress. Timestamps are needed for dynamic resolution of ϕ instructions, as the ϕ must decide which on the input operands to multiplex onto the output, and should chose the most recent of the present inputs, subject to the constraint that all producer instructions have fired if able.

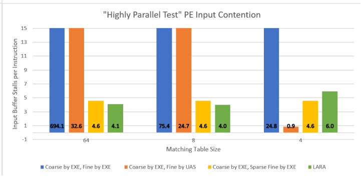
LARA Algorithm. Liveness analysis is traditionally used to choose smart register placements for active variables, but this same methodology also gives insight into what variables are physically in-flight at any program point. Therefore, if two variables are live at the same time, this means that they can potentially be forwarded to their consumer at the same time, and hence their producers should be physically located on different processing elements for paralellism. To achieve this, the compiler performs liveness analysis on the CFG through an iterative CFG-traversal algorithm. Afterwards, we construct a variable interference graph, an edge existing if two variables are live at the same time. Then, we greedily color this graph to assign a different processing element to each conflicting variable, and by extension their producers. Being in SSA form, we are guaranteed to have a chordal interference graph, which in turn means that our greedy coloring algorithm is provably optimal [9].

Selected Results. We compared the LARA algorithm to several commonly-used instruction schedulers [4] on a series of benchmark programs which we compiled and simulated on the aforementioned systems. LARA provides a reasonable speedup on









parallel code due to significantly less input contention than the commonly-used Coarse-by-exe/Fine-by-exe scheduler. As a trade-off, LARA suffers from higher latency since producer instructions are distributed for parallelism versus being clustered.

References

[1] S. Swanson, K. Michelson, A. Schwerin and M. Oskin, ”WaveScalar,” Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36., San Diego, CA, USA, 2003, pp. 291-302, doi: 10.1109/MICRO.2003.1253203.

[2] Hurson, Ali, et al. “Dataflow Computers: Their History and Future.” Wiley Encyclopedia of Computer Science and Engineering, 2007, doi:10.1002/9780470050118.ecse102.

[3] Yeager, K.C., et al. “The Mips R10000 superscalar microprocessor.” IEEE Micro, vol. 16, no. 2, 1996, doi:10.1109/40.491460.

[4] Mercaldi, Martha, et al. “Instruction Scheduling for a Tiled Dataflow Architecture.” ACM SIGPLAN Notices, vol. 41, no. 11, 2006, pp. 141–150., doi:10.1145/1168918.1168876.

[5] Cytron, Roy, et al. ”Efficiently computing static single assignment form and the control dependence graph.” ACM Transactions on Programming Languages and Systems, 1991, doi:10.1145/115372.115320

[6] ”On loops, dominators, and dominance frontiers.” ACM Transactions on Programming Languages and Systems, 2002, doi:10.1145/570886.570887

[7] Allen, F.E., et al. ”A program data flow analysis procedure.” Communications of the ACM, 1976, doi:10.1145/360018.360025

[8] Shamizi S., Lotfi S. (2011) Register Allocation via Graph Coloring Using an Evolutionary Algorithm. In: Panigrahi B.K., Suganthan P.N., Das S., Satapathy S.C. (eds) Swarm, Evolutionary, and Memetic Computing. SEMCCO 2011. Lecture Notes in Computer Science, vol 7077. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-27242-4_1

[9] Raman, Rajiv, et al. ”Approximating interval coloring and max-coloring in chordal graphs.” ACM Journal of Experimental Algorithmics, 2005, doi:10.1145/1064546.1180619

[10] Leung, Allen, et al. ”Static single assignment form for machine code.” ACM SIGPLAN Notices, 1999, doi:10.1145/301631.301667