Generating Hardware from Java Using Self-Propagating Flowpaths

Darrin M. Hanna, Bryant Jones, Lincoln Lorenz, and Mark Bowers

Abstract—**The performance of software executed on a microprocessor is adversely affected by the basic fetch-execute cycle. A further performance penalty results from the load-execute-store paradigm associated with the use of local variables in most high-level languages. Implementing a software algorithm directly in hardware such as on an FPGA can alleviate these performance penalties. Such implementations are normally developed in a hardware description language such as VHDL or Verilog. Previous work has been completed to create a compiler for converting high-level stack-based languages to VHDL for use on an FPGA or ASIC. This allowed for special-purpose processors to be generated efficiently from high-level algorithms with minimal design time. Introduced in this paper is a significant optimization to the original flowpaths – we have completely eliminated the controller and modified all operations to control themselves. These new self-propagating flowpaths execute faster and are less resource intensive. Comparisons to previous examples show that the new design exhibits, on average, a decrease in execution time of 32%, operating frequencies of 1.6 times higher, and a 33% decrease in power consumption. These flowpaths can be generated from languages with a stack-based intermediate representation including Java, C++, C#, and VB.**

Index Terms—**Field Programmable Gate Arrays, Program Compilers, Embedded Systems, Object-oriented Design.**

I. Introduction

Over the past ten years, field-programmable gate arrays (FPGAs) have become increasingly popular in the area of embedded systems. Due to lower costs and an increase in the resources available with lower-end models, FPGAs can be used in a wide range of applications. FPGAs have shown to be optimal for use in high-performance systems while reducing power consumption.

A special-purpose processor (SPP) or custom digital circuit implemented on an FPGA is an ideal replacement for a microcontroller. Custom hardware such as SPPs can realize an algorithm more efficiently than a general-purpose microcontroller with load-execute-store overhead. However, SPPs increase in size, requiring more logic for larger algorithms, while a microcontroller can execute as large an algorithm as the program memory can hold using a fixed amount of logic. For others, a SPP on an FPGA can be used as a coprocessor to a microcontroller to help speed up particular functions or sub-procedures.

Manuscript received March 31, 2011.

Darrin M. Hanna, Bryant Jones, Lincoln Lorenz, and Mark Bowers are with the School of Engineering and Computer Science at Oakland University in Rochester, MI (dmhanna@oakland.edu).

Designing a SPP is much more difficult than writing software in a high-level programming language to execute on a microcontroller. In order to design a SPP for a particular algorithm, a designer must learn how to develop hardware using a hardware description language such as VHDL or Verilog. Further, design of a SPP for a lengthy algorithm can be time consuming and requires a skilled computer engineer to do so efficiently.

One common idea to decrease development time is to use a high-level language to develop hardware. Several techniques have been introduced that use this concept. Techniques for generating SPPs such as Handel-C, often require learning a new or significantly altered language, and have the bottleneck of often being register based as described in [1].

Using the method introduced in [1, 3] SPPs can be generated from algorithms written in high-level stack-based intermediate representations (IR). This has the advantage of being generated from an unmodified high-level language. This is also more efficient than previous methods that use registers for each variable. The SPPs generated using this technique are called flowpaths. An embedded system can be designed and implemented rapidly using a high-level programming language.

Our previous method generated flowpath SPPs with two basic components, a datapath and a state controller. An optimization of this architecture is to distribute the controller into each low-level operation to allow for smaller, more efficient designs that can operate at higher frequencies.

In this paper, an optimization of the flowpaths architecture is introduced using a stateless self-propagating method that results in improvements for both speed and chip utilization. Outlined in Section 2 is the new stateless self-propagating architecture. Section 3 shows results using several benchmarking algorithms, comparing efficiency in an embedded system. Sections 4 and 5 describe additional benefits of flowpaths. The paper closes by providing concluding remarks and future work.

II. Self-Propagating Flowpaths

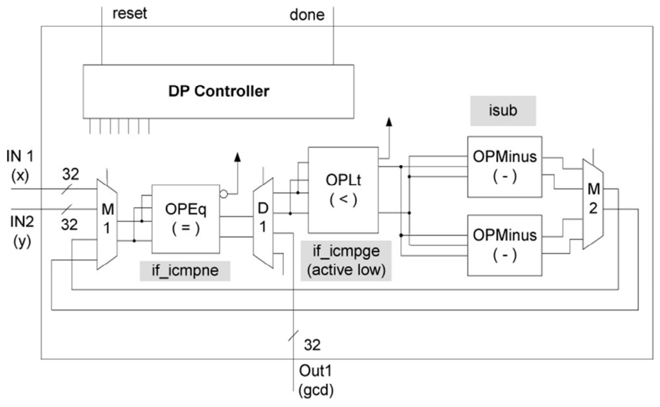
Software programs written in a stack-based language can be converted directly to circuits called flowpaths [3]. Stack-based programming languages inherently minimize the use of local variables. This is in contrast to other methods that have been developed for converting register-based code into circuits by converting each variable into a register and each assignment and access into a sequential operation. Those methods suffer from fan-out and routing issues and therefore operate at lower clock rates [1]. Several software-programming languages compile to an intermediate representation (IR) that is stack-based. Examples include the stack-based Java bytecode which is compiled from Java, and the Common Interface Language (CIL) that languages compatible with the .NET framework compile to, such as C++, C#, VB, and J#. Any of these languages could be represented similarly in hardware. The flowpaths compiler described here currently uses Java. The Java Virtual Machine (JVM) is a stack machine that runs on a microprocessor and executes Java bytecodes. Therefore, instead of executing bytecodes on a JVM, flowpaths completely eliminate the JVM by creating custom hardware that implements the program. Java bytecodes are translated to hardware operations, also known as OPs. A function-call within a Java program translates to a datapath which contains a series of connected OPs. Since flowpaths is IR-based, the generated hardware is represented in a human-readable way, unlike similar tools which often generate hardware that is obscure and difficult to modify.

Self-propagating (SP) flowpaths uses a system of cascading enables to avoid the need for an overall state controller. The nature of the hardware generated is such that algorithms or parts of algorithms execute in sequence where one operation after another is active. In this scheme, no overall controller is necessary; no single, overarching entity requires knowledge of every operation in the datapath. Rather, this knowledge is intrinsic to the individual OPs, and is therefore distributed.

A. Control Signals

Each OP is triggered by an "enable" signal, and its completion is conveyed with a "done" signal. Self-propagation is achieved by wiring the done signal from a given OP to the enable signal of the successive OP. An initial enable pulse to the system starts the cascading enables. The changing execution stack and the locals stack flow alongside this cascading status. Both combinational and sequential operations adhere to this format. The overall architecture including the new control signals is explained in Fig 1 using an example of a greatest common divisor (GCD) algorithm. A simplified flowpath to compute the GCD requires three OPs: an equality detector (OPEq), a magnitude comparator (OPLt), and a subtraction OP (OPMinus). A path using multiplexers and branches connects these OPs. The top of Fig 1 shows the GCD with the original flowpath including the state controller, and the bottom of Fig 1 shows the SP flowpath with cascading enables.

Conditional branching was previously controlled using the main state controller. Two boolean results were received from the conditional OP to notify the controller which OP should be enabled next. In this new architecture, conditional OPs simply produce two done signals, representing two paths the flow could take. When the two paths converge again, a multiplexer



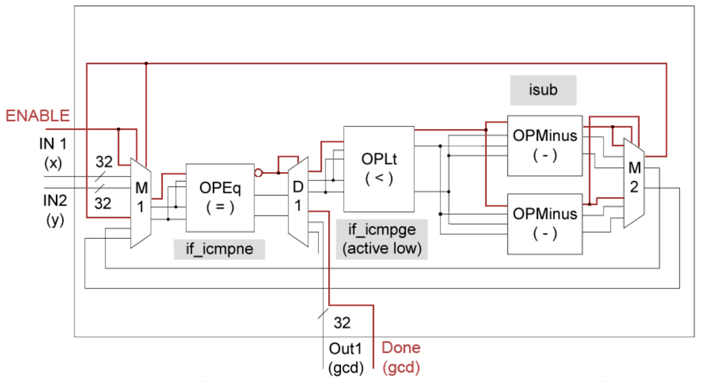


Fig 1. Comparison of the original controller (top) and SP (bottom) flowpath architectures

is used to select the appropriate flow to propagate onward. A one-hot select line is used for the multiplexer, which is driven by the done control signal output by the last OP in the active path. This is demonstrated in Fig 2.

Software loops, such as those generated by the *while* and *for* statements in Java, are very similar to conditional branches, the difference being that they contain an unconditional branch at the bottom of the statement. Unconditional branches are simply represented as connections between two OPs. If there is a conditional check within a loop there is a possibility that three paths will flow to a multiplexer: an initial entrance path, a loop condition path, and the conditional OP path.

B. Memory

Memory operations simply assume control of the memory when activated. Currently, since only one OP in a given

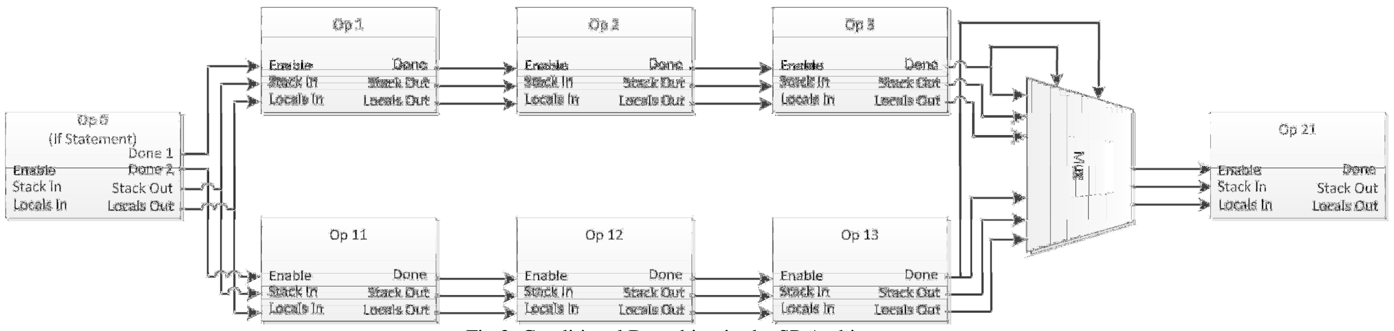


Fig 2. Conditional Branching in the SP Architecture

datapath is active at any one time, no memory arbitration scheme is necessary. Every memory OP within a given datapath is multiplexed to the memory controller. If every memory OP were directly multiplexed into the memory controller, problems would occur with routing as the design increased in complexity. In the case of a method call, OPs are multiplexed within the datapath itself and a single set of memory control signals are routed to the datapath that calls it. The top method of the hierarchy is wired directly to the memory controller. Multiplexing within a single datapath occurs through a sub-multiplexing routine where operations are multiplexed into groups before the final multiplexing stage to the single output. Relative to a datapath, a sub-method call with memory appears as a normal OP with memory. In order to support multithreaded designs with parallel flowpaths, a memory arbiter is needed.

III. Results

Several examples varying in difficulty were tested to verify functionality and performance relative to both the original flowpaths architecture and a microcontroller-based embedded system. Euclid's greatest common divisor (GCD), a quicksort, the Sieve of Eratosthenes, a complex FFT, Linpack, and the Mandelbrot fractal were tested. The GCD is a small algorithm with relatively simple constructs, such as branching, subtracting, comparing, and method calls. The Sieve of Eratosthenes and quicksort both require the use of memory, with quicksort requiring the most. FFT, Linpack, and the Mandelbrot fractal use fixed-point arithmetic.

The flowpaths produced by the compiler have been experimentally verified by simulation in Xilinx ISE version 12.3. Since all of the algorithms were written in standard Java, it is easy to verify functionality. Additionally, since the process does not alter the Java language, the exact same source code is compiled on every platform. This adds to the relevancy of performance comparisons and it aids in debugging.

Euclid's GCD was compared in both architectures of the flowpath compiler and the jStamp j-80 [4], a custom architecture that natively executes Java bytecodes at 73.7 MHz. The GCD of the values 12,365,400 and 906 was calculated. Table I displays the results of the implementation using a Xilinx Spartan 6 XC6SLX75. The new architecture showed a large decrease in the number of clock cycles necessary, along with a significant increase in the maximum execution frequency. In the original design, the algorithm required 112 slices, and the SP design only required 77 slices of the device.

Method	*Data Bus*	*Clock Cycles*	*Time(ms) @ Max Freq*	*Max Freq (MHz)*	*Energy (mW·ms) @Max*
Original Flowpath	32 bit	95,648	0.637	150	148
SP Flowpath	32 bit	54,652	0.273	200	54
jStamp	32 bit	2,690,000	36.5	73.7	6862

Table I. Relative performance comparison of GCD

The Sieve of Eratosthenes algorithm for finding all of the prime numbers less than 2048 was executed using several different methods to provide a relative performance comparison. The algorithm was compiled to a flowpath using both the original and SP architectures, and the jStamp. Results comparing the architectures are provided in Table II. Both flowpaths targeted a Xilinx Spartan6 XC6SLX75. The original flowpath generated hardware that requires 496 slices, occupying 4% of the chip. The SP flowpath, however, only requires 268 slices at 2% consumption. This space savings is expected as a result of the removal of the state controller.

Method	*Data Bus*	*Clock Cycles*	*Time(ms) @ Max Freq*	*Max Freq (MHz)*	*Energy (mW·ms) @Max*
Original Flowpath	16 bit	22,116	0.211	105	29
SP Flowpath	16 bit	16,023	0.1282	125	12
jStamp	32 bit	943,000	12.8	73.7	2406

Table II. Relative performance comparison of Sieve of Eratosthenes

A comparison of the two compilers was also done for quicksort using an identical series of 4000 random data values. The algorithm used was an iterative version, since recursion is not yet supported in flowpaths. Both designs were implemented using the same Xilinx Spartan6 XC6SLX75. Implementation results comparing the architectures are provided in Table III.

Method	*Data Bus*	*Clock Cycles*	*Time(ms) @ Max Freq*	*Max Freq (MHz)*	*Energy (mW·ms) @Max*
Original Flowpath	16 bit	659,671	13.19	50	1279
SP Flowpath	16 bit	486,520	3.892	125	366
jStamp	32 bit	37,520,000	509.1	73.7	95,711

Table III. Relative performance comparison of QSort

To demonstrate the quick design prototyping capabilities of a flowpath, a 1024-point complex FFT was created in Java. Implementing the same algorithm in hardware would take a considerable amount of time and expertise. Using flowpaths, a moderately efficient FFT implementation can be created for use in an embedded system. This was implemented targeting a Xilinx Spartan 6 XC6SLX75T FPGA. The generated hardware required 8,349 slices, utilizing 71% of the chip. The same Java algorithm was implemented on a jStamp embedded processor. The FFT utilized a 32-bit fixed-point notation for computations. Results are summarized in Table IV. In comparison to the jStamp, the flowpath FFT showed superior performance. Since the FFT algorithm can be effectively parallelized, the serial version generated by the compiler is not expected to achieve optimal results.

Method	*Data Bus*	*Clock Cycles*	*Time(ms) @ Max Freq*	*Max Freq (MHz)*	*Energy (mW·ms) @Max*
SP Flowpath	32 bit	268,891	3.841	70	714
jStamp	32 bit	17,500,000	237.6	73.7	44,669

Table IV. Relative performance comparison of 1024-point complex FFT

The classic benchmarking algorithm, Linpack, which computes the solution to a system of linear equations, was generated using the flowpath compiler. Implementing the same hardware in custom VHDL would be considerably expensive in terms of time and expertise. This generated flowpath was implemented targeting a Xilinx Spartan 6 XC6SLX150T FPGA. The hardware required 15,632 slices, utilizing 67% of the chip. The flowpath was compiled using 32-bit fixed-point notation for the computations, and the results are shown in Table V. Times are given for the solution of a linear system of size 100x100. As shown with the FFT algorithm, Linpack showed an extreme performance increase in comparison to the jStamp equivalent.

Method	*Data Bus*	*Clock Cycles*	*Time(ms) @ Max Freq*	*Max Freq (MHz)*	*Energy (mW·ms) @Max*
SP Flowpath	32 bit	4,276,400	122.2	35	34,065
jStamp	32 bit	2.064×10^8	2800.0	73.7	526,400

Table V. Relative performance comparison of Linpack

The Mandelbrot set is a fractal image produced by iterating a quadratic polynomial across points in the complex plane. A Mandelbrot calculation unit was generated using the flowpath compiler. This unit operates upon a point in the Mandelbrot set until the exit conditions for that particular point are reached. The generated flowpath was implemented targeting an Altera Cyclone II EP2C35F672C6 FPGA. The flowpath was compiled using a 32-bit fixed-point number system. This was profiled against both a custom VHDL component written by hand, and the same Java code running in a jStamp.

The generated flowpath consumed 3217 logic elements (9% of the chip), while the custom VHDL component consumed 725 logic elements (2% of the chip). Performance results for calculating a single point in the Mandelbrot set for 255 iterations are summarized in Table VI.

The flowpath version is on the same order-of-magnitude as the custom VHDL version, in terms of speed, power consumption and resource usage. Additionally, since the flowpath was generated directly from Java, the development time was substantially faster. Both hardware versions greatly exceeded the performance of the jStamp processor.

Method	*Data Bus*	*Clock Cycles*	*Time(ms) @ Max Freq*	*Max Freq (MHz)*	*Energy (mW·ms) @Max*
SP Flowpath	32 bit	6,893	0.0656	105	15.55
Custom VHDL	32 bit	1,276	0.0111	115	2.02
jStamp	32 bit	199,000	2.7	73.7	319

Table VI. Relative performance comparison of Mandelbrot calculation

Ratiometric comparisons between original flowpaths and SP flowpaths were calculated and are shown in Table VII. Examples shown are GCD, QSort, and Sieve. Data was unavailable for other examples.

Experiment	*Time*	*Max Freq*	*Energy*
GCD	0.45	1.3	0.36
Sieve	0.61	1.19	0.41
Qsort	0.29	2.5	0.29

Table VII. Ratiometric comparison of SP flowpaths vs. original flowpaths

IV. Options for further space reduction

Occasionally, a hardware designer may approach size limitations with a specific design. Depending on how large the design is, several choices can be considered to work around such an issue. Choices may include: exploring optimization methods, inserting a soft-core processor, dynamic reconfiguration, or partitioning a device over multiple FPGAs. One space optimization is to remove repetitive OPs in a datapath. A series of similar OPs could also be considered redundant in terms of space utilization. Another optimization to save space would be to insert an elastic processor capable of computing the OPs needed, in their place. Elastic cores are also ideal for complex pieces of an algorithm that are executed relatively few times. Using SP flowpaths, a design could be easily partitioned by splitting the flowpath into sections and using the simple interface to every OP as a bus to an adjacent FPGA.

V. Design Flexibility

Flowpaths have the capability to make use of custom hand-crafted VHDL blocks for use into the flowpath. An interface can be created in Java to describe the custom VHDL component. The Java method would be empty and only defines the inputs and output of the block. The compiler will recognize this as a custom block and insert it into the generated flowpath. This can be used to define the interconnection to multiple custom VHDL blocks. This concept is further described in the paper [5].

VI. Future Work

Future work includes defining a metric for determining and minimizing the critical delay path of the system. Improved optimization techniques for enhancing the speed and reducing the size of the flowpaths generated by the compiler are also being explored. These optimizations include hardware component reuse and further reduction in unnecessary clock cycles through optimization of the operations and memory usage. Furthermore, future work includes integrating CIL, another stack-based IR, to the compiler, allowing for a wide range of .NET languages to be compiled to hardware.

VII. Conclusion

This paper shows how standard stack-based programs, such as Java bytecodes, can be compiled directly to flowpaths without a centralized controller. A refined architecture was introduced here that demonstrates improved efficiency in the areas of execution speed, maximum clock frequency, power dissipation, and the amount of logic used. Using this methodology, not only is the performance increased, but also the development time is significantly decreased.

Flowpaths can outperform microprocessors at lower clock frequencies and therefore consume less energy than microprocessors or microprocessor cores. Even in situations where the FPGA requires power on the same order of magnitude as a processor, the energy required to perform a function is significantly less since special-purpose processors, including flowpaths, greatly reduce the execution time and number of clock cycles required. Energy consumption is compared in the last column of each table in Section 3. On average, flowpaths running on FPGAs consumed over 50 times less energy than a Java microcontroller.

The space reduction and performance increase of SP flowpaths makes generation of SPPs for complex algorithms such as FFT or Linpack practical to implement in an embedded hardware system. Highly complex algorithms implemented in flowpaths have shown to be superior to an identical algorithm executed on a jStamp embedded processor. Furthermore, designs can be created easily with minimal design time, and the resulting hardware is easily understandable and modifiable by a hardware designer.

References

1. D. M. Hanna and R. E. Haskell, “Flowpaths: Compiling Stack-Based IR to Hardware,” Microprocessors and Microsystems, vol. 30, pp. 125 - 136, 2006.
2. D. M. Hanna and M. Duchene, "Executing Large Algorithms on Low-Capacity FPGAs using Algorithm Partitioning and Runtime Reconfiguration," Journal of Microprocessors and Microsystems, vol 31/5 pp 302-312, August 1, 2007.
3. D. M. Hanna, M. Duchene, L. Kennedy, and B. Carpenter, "A Compiler to Generate Hardware from Java Byte Codes for High Performance, Low Energy Embedded Systems," The 2007 International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, NV, June 25 - 28, 2007.
4. Systronix, “JStamp: Real-time Native Java Module,” 2003.
5. D. M. Hanna, B. Jones, L. Lorenz, and M. Bowers, “Flexible Embedded System Design Using Flowpaths,” Submitted to the International Conference on Embedded Systems and Applications, 2011.