The RAW Benchmark Suite: Computation Structures for General Purpose Computing

Jonathan Babb, Matthew Frank, Victor Lee, Elliot Waingold,
Rajeev Barua, Michael Taylor, Jang Kim, Srikrishna Devabhaktuni, Anant Agarwal

MIT Laboratory for Computer Science
Cambridge, MA 02139
www.cag.lcs.mit.edu

Abstract

The RAW benchmark suite consists of twelve programs designed to facilitate comparing, validating, and improving reconfigurable computing systems. These benchmarks run the gamut of algorithms found in general purpose computing, including sorting, matrix operations, and graph algorithms. The suite includes an architecture-independent compilation framework, **Raw Computation Structures** *(RawCS), to express each algorithm's dependencies and to support automatic synthesis, partitioning, and mapping to a reconfigurable computer. Within this framework, each benchmark is portably designed in both C and Behavioral Verilog and scalably parameterized to consume a range of hardware resource capacities.*

To establish initial benchmark ratings, we have targeted a commercial logic emulation system based on virtual wires technology to automatically generate designs up to millions of gates (14 to 379 FPGAs). Because the virtual wires techniques abstract away machine-level details like FPGA capacity and interconnect, our hardware target for this system is an abstract reconfigurable logic fabric with memory-mapped host I/O. We report initial speeds in the range of 2X to 1800X faster than a 2.82 SPECint95 SparcStation 20 and encourage others in the field to run these benchmarks on other systems to provide a standard comparison.

1 Introduction

One goal of MIT's Reconfigurable Architecture Workstation (RAW) project is to provide the performance of reconfigurable computing in a software environment with the traditional languages and development tools available on a workstation. We have developed a suite of twelve general purpose computing benchmarks, written in architecture-independent Behavioral Verilog [8], with which to examine issues of compiling to reconfigurable architectures. Using *raw computation structures* (RawCS) we have synthesized, partitioned and mapped these applications to a reconfigurable computing architecture.

The RAW system leverages previous multi-FPGA work, virtual wires [3], in conjunction with behavioral compilation technology, to view an array of FPGAs as a machine-independent computing fabric. Given this viewpoint, we have developed a new software system which generates computation structures based on the data dependence graph of an application. These structures, which we call RawCS, are specific to each algorithm, yet are independent of any underlying architectural details. This new software operates in a framework which allows these structures to be automatically compiled onto a large array of FPGAs without user intervention.

Since the higher levels of our system do not require details of the underlying hardware, our benchmark designs mirror algorithm structure. Each design's communication structure reflects the data-dependence graph of the algorithm. Most of the algorithms we examine produced highly parallel structures, made of many copies of simple computing elements. The computing elements are specified in Behavioral Verilog and synthesized in just a few minutes. The RawCS generator connects the elements based on the application's unrolled dependence graph.

Because the system abstracts away all the hardware dependent design issues, we are able to automatically generate large architecture-independent netlists for each benchmark. For each of the twelve benchmarks, we have generated a set of three different designs, small, medium and large that range in final size between 14 and 379 FPGAs. The resulting speedups are dependent on the amount of parallelism in the data dependence graph. We find that for these designs we execute in the range of 2X to 1800X faster than a 2.82 SPECint95 SparcStation 20.

The rest of this paper is organized as follows: After providing background on configurable computing and virtual wires technology in Section 2, Section 3 then describes Raw Computation Structures within the context of a complete reconfigurable compiler system. After Section 4 overviews the benchmark applications, Section 5 describes the prototype hardware system, and Section 6 then presents area and timing results for each problem. Finally, Section 7 and Section 8 describe related and future work in this area and Section 9 makes concluding remarks.

2 Background

Configurable computers based on Field Programmable Gate Arrays (FPGAs) are capable of accelerating suitable applications by several orders of magnitude when compared to traditional processor-based architectures (see Splash [12] and PAM [5]). This performance is achieved by mapping a user application into a gate-level netlist that may be downloaded onto programmable hardware. The programming paradigm on these machines, however, prohibits the development of automatically-compiled, architecture-independent applications because the programmer must explicitly account for machine-level details such as FPGA capacity and interconnect.

The development and commercial availability of virtual wires compiler technology [3] enables the efficient combining of multiple FPGAs for use as a single, giant sea of gates by higher-level synthesis compilation steps. This software takes as input a user supplied netlist, automatically partitions and places the design, and then intelligently multiplexes each physical wire among multiple logical wires and pipelines these connections at the maximum clocking frequency of the FPGA. While this technology has primarily been applied to in-circuit emulation and logic simulation acceleration, it has also been applied effectively to reconfigurable computing of *hardware subroutines*, where an FPGA array implements a Verilog version of a subroutine in a C program and connects to the software by remote calls from a host workstation [4].

3 Raw Computation Structures

To support compilation of the RAW benchmark suite to reconfigurable architectures, we have developed a common framework based on *raw computation structures* (RawCS). RawCS supports user-level hardware by dynamically modifying the underlying architecture as a function of each application. In some cases, such as matrix multiply and sorting, these modifications are a function of the algorithm's

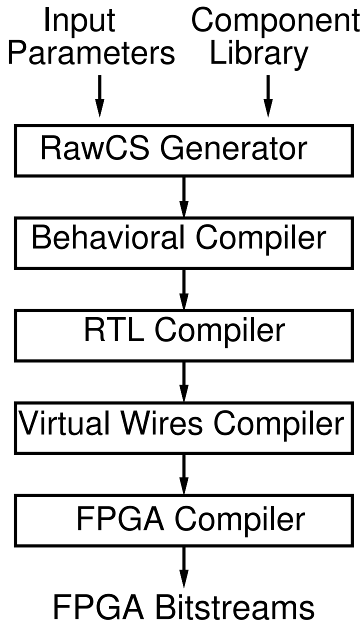

Figure 1: Compiler Flow

problem size. In other cases, such as the transitive closure benchmark, where these modifications are also a function of some of the input data for the application, RawCS compilation can be viewed as an extreme case of generation of dynamic code [11], where new processor instructions are dynamically generated based on the input data-set. Dynamic code generation is a software technique that allows specialization and optimization of code based on program input.

Software compiler optimizations like constant folding, dead code elimination and removal of branches expose instruction level parallelism and enable the software code generator to create significantly faster code. RawCS's can be additionally specialized beyond the instruction set abstraction by creating new operators, distributing memory accesses, minimizing data widths, and reducing many complex data structures to wire permutations. The resulting structures are thus tailored to a specific application.

3.1 Software Tool Flow

Figure 1 shows the overall software tool flow used to compile the RAW benchmarks suite. In comparison with standard compilation technology for microprocessor computing, the transformations are more complex. This system begins with the RawCS generator, a benchmark-specific program written in C, that takes input parameters for the

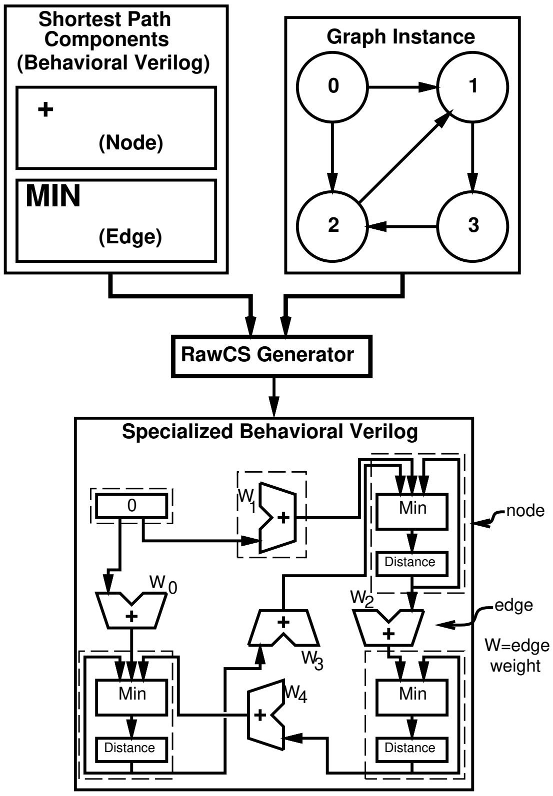

Figure 2: RawCS Flowchart for Shortest Path

benchmark, along with the benchmark's Behavioral Verilog component library, and generates specialized Behavioral Verilog code. This Verilog code describes a hardware instance for the benchmark at a high-level, with no references to any underlying technology. The next two tools consist of a behavioral compiler and an RTL compiler which together map the input Verilog into a single, gigantic netlist composed of generic logic gates in a reference technology. The virtual wires compiler then maps this netlist into multiple FPGAs. More specifically, the virtual wires compiler treats this netlist as a design to be emulated: partitioning, placing, and scheduling inter-FPGA connections to produce individual netlists for each FPGA in the emulator. Each individual netlist is then processed by an FPGA compiler that places and routes the netlist for the target FPGAs, producing FPGA bitstreams. These layers hide successive levels of detail from the RawCS generator: the FPGA layer hides the internal FPGA details; the virtual wires layers hides the inter-FPGA topology and communications as well as the FPGA gate and pin capacity; the RTL layer hides the reference technology libraries; and the behavioral layer hides state machine and datapath details.

3.2 RAW Computation Structure Generation

Each application developed in the RawCS framework entailed writing a computation structure generator in C and a library of components in Behavioral Verilog. These are described in more detail below.

Each benchmark includes an application-specific computation structure generator in the form of a C program written by the benchmark developer. The input to this generate program includes parameters specifying problem size, datapath width, and degree of parallelism. Given these inputs, the program makes calls to a common Verilog generation library to instantiate wires and computation structure components. Note that these components must currently be specified in a separate Verilog library. The output is then a single design in synthesizable Behavioral Verilog. At this point the Verilog code is application-specific and architecture-independent. It can be targeted to any multi-FPGA system.

In addition to the static, data-independent parameters such as problem size, several benchmarks also include dynamic computation structure generation in which the input data is analyzed to determine the number and arrangement of computation structures. Specifically, the generators for the shortest path and transitive closure graph problems also specialize an input graph topology. Dynamic computation structures are described in more detail in [2].

While, in our current implementation, the generate programs are hand-coded by application of known compilation techniques such as partial evaluation, loop unrolling, and speculative parallelization, a high-level compiler can automate this task. That is, given an input description of the application in a high level language such as C, a compiler can partially evaluate expressions, unroll inner loops, and flatten other types of control structures to generate computation structures without user intervention. We demonstrate the viability of such a compiler by manually performing these techniques for each of our benchmarks with the help of a C program.

Figure 2 demonstrates an example of the operation of the generate program for the shortest path application described in Section 4. This particular generator takes as input a topological description of a graph instance. Based on this topology, the generator instantiates and interconnects components from a library of generic descriptions (in Behavioral Verilog) of node and edge computation structures. The generic structures have parameters, including data widths and the number of input edges per node, that are set whenever a particular node or edge is instantiated. The generator then uses these structures to instantiate library components with the proper bus widths and operations corresponding to the input specification. In addition, the generator specifies connections between these structures corresponding to the

Benchmark	Description
bheap	Binary Heap
bubble	Bubble Sort
des	DES Data Encryption
fft	Integer Fast Fourier Transform
jacobi	Jacobi Relaxation
life	Conway's Game of Life
matmult	Integer Matrix Multiply
merge	Merge Sort
nqueens	Combinatorial N-Queens Problem
ssp	Single Source Shortest Path
spm	Multiplicative Shortest Path
tc	Transitive Closure

Table 1: The RAW Benchmarks

edges in the input graph. The generator thus creates a description of a single circuit of higher level functions which can be synthesized to gates, partitioned, and compiled to a specific FPGA technology.

3.3 Common Host Interface

Many reconfigurable computing systems support a memory-mapped interface for communication between a program executing on the host CPU and the FPGA array logic. The RAW benchmarks are implemented with a common host interface that facilitates porting to any reconfigurable computer. Each benchmark has the same top-level module. The external I/O for this top-level module is a synchronous address and data bus that supports a single address space. Each reconfigurable architecture is then expected to implement a system-specific transport layer that maps communication from this bus to the address space of the host. For example, in our experiments with the IKOS emulator, we used the SLIC EB-1 SBus card [9] and a device driver for the transport layer. Section 5 discusses our IKOS host interface in more detail.

4 Benchmark Applications

The RAW benchmarks consist of twelve programs representative of a variety of algorithms including sorting, matrix operations, combinatorial search and graph problems. These benchmarks are selected from among standard microprocessor and parallel computing benchmarks. In each case the application is implemented with multiple copies of a small computation element written in Behavioral Verilog. These computation elements are then connected, using the RawCS generator, to reflect the dependence structure of the unrolled loops. We continue by discussing each algorithm and its implementation under the RawCS system.

Binary Heap This benchmark implements the heapifying operation to convert an arbitrary binary tree structure into a binary heap in which the value at each node satisfies the heap property – each node's value is greater than the values of both children nodes. Heapifying consists of comparing the element at the current node with the elements of its children, determining if the heap property is violated, and swapping the appropriate elements if so. This algorithm makes a number of swaps proportional to the number of nodes in the tree, N. The hardware version converts the tree to a heap in a number of clock cycles that is proportional to $(\log N)^2$. Each node of the tree is implemented as a module that can read the contents of its left and right children, compare its own element to these two values, and update its own as well as one of its children's elements. The tree will satisfy the heap property after $\log N$ passes. Parallelism is exploited in allowing all nodes at one level to be active simultaneously.

Bubble Sort Bubble sort, one of the simplest algorithms for sorting an array, consists of repeatedly exchanging pairs of adjacent array elements that are out of order until no such pair remains. The serial software implementation of bubble sort has a time complexity that is $O(n^2)$ in the number of elements – only one pair of numbers can be examined at a time, and n passes must be taken through the array, where n is the number of elements. In contrast, by parallelizing exchanges, the RawCS parallel hardware version sorts elements in a number of clock cycles that is linear with n, and requires an amount of hardware on the order of n.

DES The Data Encryption Standard [18] (DES) is a system developed for the U.S. government for use by the general public. The DES algorithm, a combination of substitution and permutation, derives its strength from repeated application of these two techniques, one on top of the other, for a total of 16 cycles. The software algorithm used in this benchmark was adopted from Eric Young's fast encryption package. It performs rotation to align bits for simpler permutation. As a result, this version of encryption requires only 42 operations per substitution and permutation cycle. In contrast, the RawCS implementation takes advantage of the independence between each DES encryption and decryption operation. It performs n encryptions or n decryptions simultaneously, where n is the number of hardware DES modules in the design. For each encryption, we use a lookup table to bypass the shifting and substitution operations. As a result, an individual hardware DES module is expected to achieve performance similar to the best software solution and thus achieve an $O(n)$ speedup.

FFT The Fast Fourier Transform (FFT) over the field of complex numbers[19] is a common signal processing applications with inherent parallelism. An FFT of size N can be performed in hardware in a loop of length $\log N$ where each iteration of the loop permits $N/2$ computations to run in parallel. Further, the loop of length $\log N$ can be unrolled into a butterfly network and pipelined by overlapping the successive iterations. Our hardware version employs a number theoretic version of the FFT which operates in the ring of integers modulo $2^{N/2} + 1$. This differs from the complex FFT since arithmetic operations take place in a ring of integers. The basic FFT structure, including the inherent parallelism of the algorithm, is the same for both the complex and number theoretic cases. Note that the FFT data width is $N/2 + 1$ bits.

Jacobi Relaxation Jacobi relaxation is an iterative algorithm which, given a set of boundary conditions, finds discretized solutions to differential equations of the form $\nabla^2\mathcal{A} + \mathcal{B} = 0$. Each step of the algorithm replaces each node of a grid with the average of the values of its nearest neighbors. We have implemented an integer version of Jacobi relaxation. A computation element representing each grid point simply produces an average of its four inputs. We use the RawCS generator to generate connections between each computation element and its four neighbors. The resulting circuit has a simple regular grid structure, reflective of the data dependence graph of the algorithm.

Game of Life Conway's Game of Life program is represented on a two-dimensional array of cells, each cell being alive or dead at any given time. The program begins with an initial configuration for the cells, and henceforth obeys the following set of rules: a living cell remains alive if it has exactly two or three living neighbors, otherwise it dies; a dead cell becomes alive if it has exactly three living neighbors, otherwise it stays dead.

Matrix Multiply This version of Matrix Multiply multiplies two $N \times N$ matrices in $O(N \log N)$ time given $O(N^2)$ hardware multipliers. This runtime is achieved by implementing a vector-matrix multiplier, which stores an initial matrix away, and repeatedly returns its product with an input vector. The multiplication stage of the dot products is in parallel while the additions are scheduled in a binary-tree fashion, with $\log N$ levels.

Merge Sort The hardware structure that implements the merge sort is a binary tree with the leaves of the tree each containing a single element of the data set that is to be sorted. Each node of the tree performs a comparison of its two inputs from its subnodes, stores the higher value in a register which is connected to the output of the node, and then tells the subnode that had the higher value to load a new value into its register. The subnode that gets the load signal performs the same operation, and so on, until a leaf node is hit. Once a leaf node gets a load signal, it loads its register with a zero. As data is pulled off the top of the tree, larger data values float to the top. The sort is completed in $O(N)$ time.

N-Queens The N-Queens benchmark solves the combinatorially hard chess problem of placing N queens on an $N \times N$ chessboard such that no queen can attack any other. The typical software solution to this problem uses a recursive search for a placement of the queens that meets the correct conditions. In the RAW benchmark the program is represented by N computation elements, each of which is responsible for the position of one of the queens. A token is passed back and forth between these computational elements representing control flow, much as a stack pointer would in the recursive software solution to the algorithm. Each computational element monitors the positions of the queens in other columns to determine a legal position for its own queen. During a particular cycle, only the computational element currently holding the token changes the position of its queen.

Graph Problems The RAW benchmarks for single source shortest path (ssp), multiplicative shortest path (spm) and transitive closure (tc) specialize both the algorithm and the topology of the input graph. In each case we generate computation elements for each node of the input graph and then use the RawCS generator to connect these elements in the same topology as the problem input graph. For the shortest path problems the computation element is a circuit that finds the minimum of its inputs. For the transitive closure problem the computation element is an AND function across its inputs. With the exception of ssp64-mesh, each of the graph benchmark cases is randomly generated with a maximum node in-degree of eight, and an average in-degree of four. See [2] for more details.

4.1 Limitations

Our current benchmark suite has several limitations. First, the designs are not multiplexed and thus require gate counts proportional to problem size. In addition, none of the applications have significant memory requirements. Also, all applications have very small code size. Finally, the benchmarks are not designed to evaluate I/O performance.

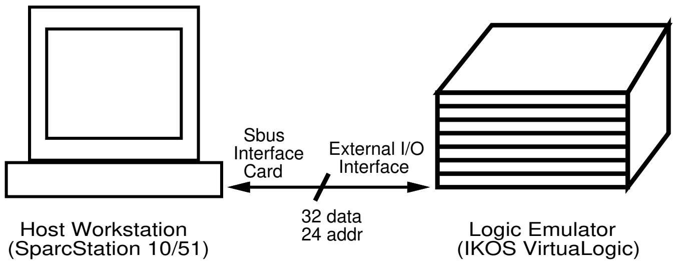

Figure 3: Prototype Reconfigurable Computing System

Tool Function	Software	time
RawCS Generator	C Program	seconds
Behavioral Compiler	Synopsys	minutes
RTL Compiler	Synopsys	ten minutes
Virtual Wires Compiler	IKOS	2 hours / board
FPGA Compiler	Xilinx	2 hours / board (10 hosts)

Table 2: Experimental Software System

5 Experimental System

5.1 Architecture

The RAW Benchmark Suite can be targeted to a variety of reconfigurable computing systems. Initially, we are targeting a prototype hardware system consisting of a VirtuaLogic Emulator (VLE) from IKOS Systems coupled with a Sun SparcStation 10/51 via a SLIC S-bus interface card [9] (Figure 3). Not shown is a SCSI interface to the emulator for downloading configurations and controlling clock speed. We are currently using a production VLE system consisting of five arrays of 64 Xilinx 4013 FPGAs each. The FPGAs on each board are directly connected in nearest-neighbor meshes augmented by longer connections to more distant FPGAs. Boards are coupled together with multiplexed I/Os. Additionally, each board has several hundred external I/Os, resulting in total external I/O connections of a few thousand.

When used as a logic emulator, the external I/O interface of the VLE is generally connected to the target system of the design under emulation. For reconfigurable computing, we have instead connected a portion of the external I/O to an Sbus interface card in the host SparcStation. This card provides an asynchronous bus with 24 bits of address and 32 bits of data which may be read and written directly via memory-mapped I/O to the Sparc Sbus. We are successfully operating this interface at conservative rates of 0.25MHz for reads and 0.5MHz for write operations given a 1MHz emulation clock, providing 1-2 Mbytes/sec rates for communication between the host CPU and the FPGAs of the emulator. This limited I/O rate allows one 32 bit read/write every 100/50 cycles of the 50MHZ host CPU.

5.2 Compiler

We have implemented a C library for building RawCS generators for each benchmark and constructed an experimental software system out of this generator library and other commercial tools. Table 2 lists the software used for each tool step and rough running times on SparcStation 20 class machines. By far the most computationally expensive step is the last FPGA compile. However, FPGA compiles may generally be parallelized over a network of workstations to provide reasonably effective compile times.

6 Results

We implemented a RawCS generator and a library of computation elements for each benchmark. For each benchmark we generated cases ranging in size from a few FPGAs to hundreds of FPGAs. Table 3 summarizes our results. The prefix of the problem specifies the benchmark name as defined in Section 4, with the following number identifying the number of computational elements. For each benchmark we list the data path width, number of computation elements, gate count and the total number of Xilinx 4013 FPGAs required. We also list the effective clock rate assuming a 25MHz internal virtual wires clock. To calculate the solution speed, we divide the clock rate by the number of clock cycles required to reach a solution. Finally, we compute speedup by dividing the FPGA speed by the best software speed on a 2.82 SPECint95 SparcStation 20. An additional metric we have added, speedup per FPGA, is the speedup divided by the number of FPGAs and measures the relative efficiency of solving the particular problem type on a reconfigurable architecture. Note that the datapath width for each benchmark varies across the suite. In some cases the width is particular to the application domain, while in other cases the width is simply a parameter chosen for this set of experiments.

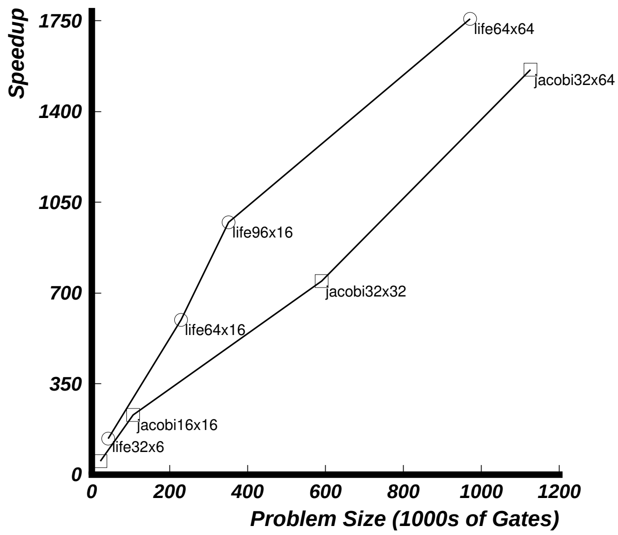

Figure 4: High Performance Scalability

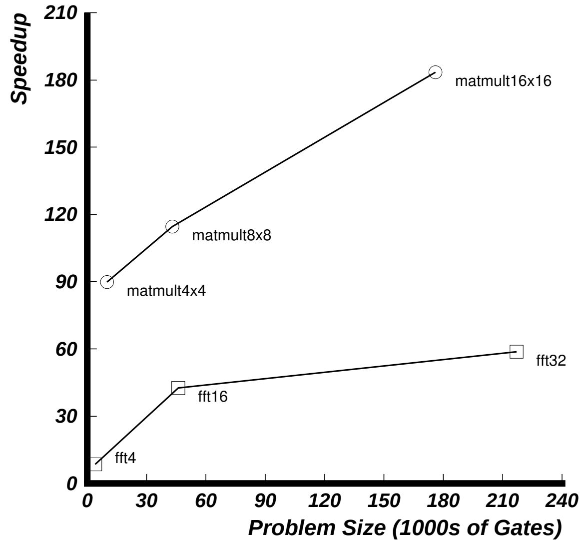

Figure 5: Medium Performance Scalability

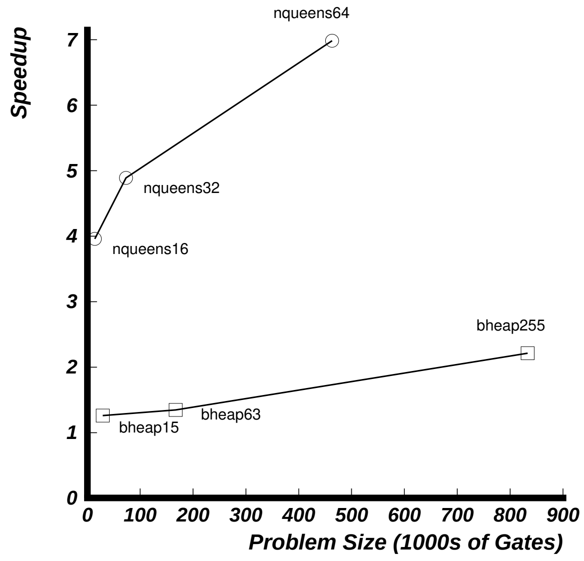

Figure 6: Low Performance Scalability

We have successfully compiled most of the smaller designs, all the way down to configuration bitstreams, and run the designs on our prototype emulation system. The execution of these designs has been validated with I/O across our memory-mapped Sbus interface.

For purposes of discussing the experimental results, we group the benchmarks into high, medium, and low performance categories as a function of the order of magnitude of speedup achieved. In general we have discovered that the level of speedup is proportional the ability of each benchmark implementation to capitalize on speedup potentials from 1) massive parallelism, 2) local communication patterns, 3) configured/specialized instructions, 4) and fine-grain data width and operators.

6.1 High Performance (100-1000X)

The group of benchmarks that achieved two to three orders of magnitude speedup includes Jacobi (jacobi), Life (life), and Transitive Closure (tc). These cases benefit from all speedup potentials, with massive parallelism, regular communication, customized instructions that unroll the entire inner loops in a single cycle, and small bit or byte-wide data widths. These benchmarks exhibit nearly linear scaling with problem size (Figure 4). In comparing Jacobi to Life, Life achieves higher performance due to the bit level operations while Jacobi operates on byte level data. Thus many more Life computational elements fit per unit of area than do Jacobi computation elements. A caveat to the bit-level performance numbers is that we are not currently implementing known improvements to the software version to take advantage of the bit-level parallelism within a microprocessor.

6.2 Medium Performance (10-100X)

The group of benchmarks that achieved one to two orders of magnitude speedup includes Bubble Sort (bubble), DES Encryption (des), Fast Fourier Transform (fft), Matrix Multiply (matmul), Shortest Path (ssp), and Multiplicative

Benchmark case	Data width (bits)	number of elements	gate count	FPGA count (X4013)	clock rate (MHz)	solution rate (KHz)	software rate (Hz)	speedup vs software	speedup per FPGA
bheap15	32	15	29k	20	2.08	130	103K	1.26	0.06
bheap63	32	63	167k	64	1.19	33.07	25K	1.34	0.02
bheap255	32	255	833k	320	0.78	12.21	6K	2.21	0.01
bubble64	32	64	142k	64	1.25	39.06	6K	7	0.11
bubble256	32	256	638k	261	0.83	6.51	370	18	0.07
bubble512	32	512	1394k	320	0.61	2.38	94	25	0.08
des4	64	4	47k	41	1.92	428	60K	7	0.17
des48	64	48	596k	219	1.32	3509	60K	58	0.27
des96	64	96	1305k	320	0.93	4938	60K	82	0.26
fft4	3	4	4k	17	2.27	568	67K	9	0.50
fft16	9	16	46k	44	2.08	347	8K	43	0.97
fft32	17	32	217k	64	1.32	188	3K	59	0.92
jacobi8x8	8	64	22k	33	2.78	43.40	833	52	1.58
jacobi16x16	8	256	106k	85	2.27	35.51	154	230	2.71
jacobi32x32	8	1024	590k	231	1.56	24.41	33	747	3.23
jacobi32x64	8	2048	1126k	379	1.56	24.41	16	1562	4.12
life32x6	1	192	42k	33	2.08	32.55	234	139	4.21
life64x16	1	1024	229k	108	1.92	30.05	50	597	5.53
life96x16	1	1536	351k	178	2.08	32.55	33	973	5.47
life64x64	1	4096	971k	354	1.47	22.98	13	1758	4.97
matmult4x4	16	16	10k	18	3.12	781	9K	90	4.99
matmult8x8	16	64	43k	42	2.78	347	3K	115	2.73
matmult16x16	16	256	176k	64	1.56	97.66	532	183	2.87
merge8	32	8	14k	24	3.12	312	120K	2.60	0.11
merge64	32	64	114k	59	1.79	25.88	13K	1.98	0.03
merge256	32	256	596k	201	1.19	4.53	3K	1.62	0.01
nqueens16	1	16	14k	17	3.57	1786	451K	3.96	0.23
nqueens32	1	32	73k	60	2.50	1250	256K	4.89	0.08
nqueens64	1	64	463k	215	1.79	893	128K	7	0.03
ssp16	16	16	44K	14	1.79	112	11K	10	0.71
ssp64	16	64	181K	56	1.14	18	658	27	0.48
ssp64-mesh	16	64	159K	46	1.56	24	758	32	0.70
ssp128	16	128	366K	118	0.78	6.1	149	41	0.35
ssp256	16	256	814K	261	0.34	1.3	25	52	0.20
spm16	16	16	156K	36	1.39	87	6.3K	14	0.39
spm32	16	32	310K	90	1.19	37	1.5K	25	0.28
tc512	1	512	187K	48	1.47	2.9	7.2	398	8.29

Table 3: Benchmark Results

Shortest Path (spm). Unlike the low performance cases, speedup for these cases did scale up with increasing problem size (Figure 5), but not linearly. Increasing speedup is reflective of the large amounts of instruction-level parallelism available while the increasing global communication overheads contribute to a slower hardware clock speed and thus non-linear speedup. Note that both FFT and DES benefit from customized datapath and custom operator widths. For example, DES implements a 64 bit algorithm while the software version must serialize 16 bit operations to perform the correct permutations of the algorithm.

To pin-point the effects of increasing global communication overhead, we created ssp64-mesh, a design with much greater locality compared to the randomly generated graphs in the other shortest path problems. We found that ssp64-mesh has a speedup of 0.70 per FPGA as opposed to 0.48 for Ssp64, nearly a 50 percent improvement.

6.3 Low Performance (1-10X)

The group of benchmarks that achieved one order of magnitude or less of speedup includes Binary Heap (bheap), Merge Sort (merge), and N Queens (nqueens). In addition to low speedup, these benchmarks also did not scale with increasing problem size (Figure 6), mainly due to a lack of available parallelism. Note that bheap and nqueens exhibit a log scale-up while merge does not scale at all. These designs are achieving a small speedup from specialization with nqueens also benefiting from bit level data width, but

Structure	Combinational	Sequential	Total
Bheap-Node	374 gates	320 gates	694 gates
Bubble-Node	589 gates	352 gates	941 gates
DES-Node	3499 gates	1584 gates	5083 gates
FFT-Node	1404 gates	192 gates	1596 gates
Jacobi-Node	275 gates	160 gates	435 gates
Life-Node	221 gates	8 gates	229 gates
Merge-Node	544 gates	256 gates	800 gates
Matmult-Node	1620 gates	64 gates	1684 gates
N-Queens-Node	379 gates	480 gates	859 gates
Ssp-Node	1435 gates	684 gates	2119 gates
Spm-Node	6629 gates	673 gates	7302 gates
Tc-Node	269 gates	73 gates	342 gates

Table 4: Computation Granularity

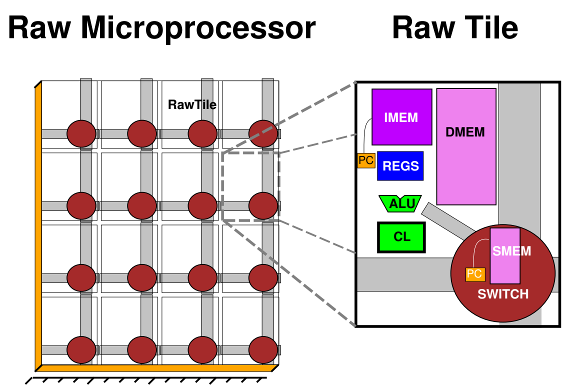

Figure 7: Tiled Raw Microprocessor

they all lack the massive parallelism found in the higher performance benchmarks. We think that a multiplexed bheap design that re-used hardware resources might achieve a more scalable speedup.

6.4 Computation Granularity

To conclude discussion of our results, we present the structure area for our synthesized library elements (Table 4). Each element is roughly one processing node in the reconfigurable solution. The granularity of computation, a function of the operator width and the inner loop complexity, translated directly into computation element size. Besides decreasing gate requirements, the size of the computation elements in each benchmark also influences the final clock rate of the resulting design – with smaller elements there is less inter-FPGA communication.

7 Related Work

A number of similar general purpose applications have previously been examined for reconfigurable computing. Iseli and Sanchez implemented the Life benchmark on the Spyder architecture [14]. Sorting algorithms have been investigated by Luk *et al* [17], Carrera *et al* [7], Amerson *et al* [1] and Hauser and Wawrzynek [13]. Priority queues have been examined by Luk [16]. Lew and Halverson [15] have used data dependency information to specify algorithms for the discrete cosine and dynamic programming solutions for the shortest path problem. The discrete cosine has also been investigated by Ebeling *et al* [10]. Bittner and Athanas [6] have implemented vector dot product. Yeh, Feygin and Chow [21] implemented a Viterbi decoder which has a similar structure to the Raw shortest path benchmark. A 3D heat equation solver, with a structure similar to the Jacobi benchmark has been implemented for the PAM architecture [5].

8 Future Work

Many of the gains in application performance found in Section 6 were offset by the additional overhead of using an exclusively fine grain medium (FPGAs) for computing. We estimate that our applications could have achieved yet another order of magnitude speed improvement if our system could be run at the clock rates (*e.g.* 500MHz) of modern processors. Our future work involves the development of an architecture and compiler for a Raw microprocessor (RawμP) [20] which provides the benefits found in reconfigurable computing environments without the costs of using only FPGA logic. As shown in Figure 7 the Raw Microprocessor is composed of a set of simple interconnected tiles, each tile comprising instruction and data memory, an ALU, registers, some configurable logic, and a programmable switch. Wide channels couple near-neighbor tiles. The hardware architecture is fully exposed to the compiler and the switch is programmable via a sequence of compiler determined instructions.

This architecture would provide exactly those features which permit reconfigurable applications to run faster than they would on a typical microprocessor without sacrificing speed for coarse-grain operations. Portions of the application with independent control flow can run in parallel on different tiles. The tightly coupled communication network permits inexpensive communication and synchronization between tiles. Finally, the configurable logic block on each tile for can be used fine grain and specialized operations.

9 Summary

This paper has presented twelve general purpose computing benchmarks for reconfigurable architectures. We have introduced raw computation structures, a framework for expressing the data communication patterns of an algorithm. By leveraging commercial virtual wires technology to hide the details of our underlying hardware, we have implemented a set of architecture-independent benchmarks in high-level Behavioral Verilog. We measured speeds on an IKOS VirtuaLogic emulator from 2X to 1800X faster than a 2.82 SPECint95 Sparc 20 for designs up to 379 FPGAs. These results show that applications written at a relatively high level can achieve the orders of magnitude speedups typical of hand-crafted reconfigurable hardware.

Acknowledgments

The research report in this paper was funded in part by ARPA contract # DABT63-96-C-0036 and by an NSF Presidential Young Investigator Award to Prof. Agarwal. Matthew Frank is in part supported by an NSF Graduate Fellowship. The VirtuaLogic emulation system was donated by Ikos Systems.

References

[1] R. Amerson, R. Carter, B. Culbertson, P. Kuekes, and G. Snider. Teramac–configurable custom computing. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 32–38, Napa, CA, Apr. 1995.

[2] J. Babb, M. Frank, and A. Agarwal. Solving graph problems with dynamic computation structures. In *SPIE Photonics East: Reconfigurable Technology for Rapid Product Development & Computing*, Boston, MA, Nov. 1996.

[3] J. Babb, R. Tessier, and A. Agarwal. Virtual Wires: Overcoming pin limitations in FPGA-based logic emulators. In *Proceedings IEEE Workshop on FPGA-based Custom Computing Machines*, pages 142–151, Napa, CA, April 1993. IEEE. Also as MIT/LCS TM-491, January 1993.

[4] T. Bauer. The Design of an Efficient Hardware Subroutine Protocol for FPGAs. Master's thesis, EECS Deptartment, MIT, Department of Electrical Engineering and Computer Science, May 1994.

[5] P. Betrin and H. Touati. Pam programming environments: Practice and experience. *Napa*, pages 133–138, April 1994.

[6] R. A. Bittner, Jr. and P. M. Athanas. Computing Kernels Implemented with a Wormhole RTR CCM. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Apr. 1997.

[7] J. M. Carrera, E. J. Martinez, S. A. Fernandez, and J. M. Chaus. Architecture of a FPGA-based coprocessor: The PAR-1. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 20–29, Napa, CA, Apr. 1995.

[8] D. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, Boston, 1991.

[9] DAWN VME Products. *SLIC Evaluation Board User's Guide for DAWN VME PRODUCTS SLIC EB-1 Version 1.0*, June 1993.

[10] C. Ebeling, D. C. Cronquist, P. Franklin, J. Secosky, and S. G. Berg. Mapping applications to the rapid configurable architecture. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Apr. 1997.

[11] D. R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *PLDI '96*, 1996.

[12] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweeney, and D. Lopresti. Building and using a highly parallel programmable logic array. *Computer*, 24(1), Jan. 1991.

[13] J. R. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Apr. 1997.

[14] C. Iseli and E. Sanchez. Spyder: A reconfigurable VLIW processor using FPGAs. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 17–24, Napa, CA, Apr. 1993.

[15] A. Lew and R. Halverson, Jr. A FCCM for Dataflow (Spreadsheet) Programs. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Apr. 1995.

[16] W. Luk. A declarative approach to incremental custom computing. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 164–172, Napa, CA, Apr. 1995.

[17] W. Luk, V. Lok, and I. Page. Hardware acceleration of divide-and-conquer paradigms: a case study. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 192–201, Napa, CA, Apr. 1993.

[18] C. P. Pfleeger. *Security in Computing*. Prentice-Hall, Inc., New Jersey, 1989.

[19] W. Press et al. *Fast Fourier Transform (FFT)*, pages 504–510. Cambridge University Press, New York, 1995.

[20] E. Waingold, M. Taylor, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktuni, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to Software: The Raw Machine. Technical Report TR-709, MIT LCS, Mar. 1997.

[21] D. Yeh, G. Geygin, and P. Chow. RACER: A Reconfigurable Constraint-Length 14 Viterbi Decoder. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Apr. 1996.