IMPLEMENTING SCIENTIFIC SIMULATION CODES HIGHLY TAILORED FOR VECTOR ARCHITECTURES USING CUSTOM CONFIGURABLE COMPUTING MACHINES

David Rutishauser, Virginia Polytechnic Institute, rutishau@vt.edu

Abstract

The motivation for this work comes from an observation that amidst the push for Massively Parallel (MP) solutions to high-end computing problems such as numerical physical simulations, large amounts of legacy code exist that are highly optimized for vector supercomputers. Because re-hosting legacy code often requires a complete re-write of the original code, which can be a very long and expensive effort, this work examines the potential to exploit reconfigurable computing machines in place of a vector supercomputer to implement an essentially unmodified legacy source code. Custom and reconfigurable computing resources could be used to emulate an original application's target platform to the extent required to achieve high performance. To arrive at an architecture that delivers the desired performance subject to limited resources involves solving a multi-variable optimization problem with constraints. Prior research has shown that designing an optimum hardware implementation of a given application under hardware resource constraints is a type of problem for which general efficient solutions have not been found. The premise of this approach is that the goal of applying reconfigurable computing resources to the implementation of an application, maximizing the performance of the computation subject to physical resource constraints, can be achieved practically by assuming a computational paradigm, such as vector processing.

This research contributes a formulation of the problem and a methodology to design a reconfigurable vector processing implementation of a given application that satisfies a performance metric. A generic, parametric, architectural framework for vector processing implemented in reconfigurable logic is developed as a target for a scheduling/mapping algorithm that maps an input computation to a given instance of the architecture. This algorithm is integrated with an optimization framework to arrive at a specification of the architecture parameters that attempts to minimize execution time, while staying within resource constraints. The flexibility of using a custom reconfigurable implementation is exploited in a unique manner to leverage the lessons learned in vector supercomputer development. The vector processing framework is tailored to the application, with variable parameters that are fixed in traditional vector processing. Benchmark data that demonstrates the functionality and utility of the approach is presented. The benchmark data includes an identified bottleneck in a real case study example vector code, the NASA Langley Terminal Area Simulation System (TASS) application.

Background and Related Research

Custom reconfigurable computing has been an active area of research from the introduction of its concepts [1] to the present day where enabling advancements in configurable processing elements [2] have allowed for a wide variety of experimentation. At the center of RC applications is the problem of defining an appropriate architecture for a given problem subject to limited resource constraints. It has been shown that the related problem of resource constrained scheduling belongs to the class of NP-complete problems, which may not have tractable solutions [3]. In order to solve practical instances of such problems, constraints on the problem scope and heuristics can be used. This research pursues finding solutions to the problem of architecture definition and scheduling under constraints by limiting the solution space to vector computations. Solutions of this form have utility for a wide range of existing scientific codes targeted for vector processing architectures. In addition, lessons learned from years of vector computer development can be leveraged with the additional unique flexibility of an RC implementation.

It is desirable to introduce few modifications to a legacy vector code desired to be run on a custom platform. Gokhale and Stone [4] developed a method of introducing pragmas, or directives into a C source code as input to a hardware complier. The complier targets a hybrid General Purpose(GP)/Reconfigurable(RC) processor, which allows for the majority of the source code to be executed on the GP processor with a standard compiler and sections identified by the pragmas to be synthesized to a hardware implementation in the RC logic. Several choices exist for reconfigurable logic devices with embedded general purpose processors, and this research uses the Xilinx Virtex II pro [5] for demonstration. (Since the beginning of the project, the Virtex-5 is soon to be released, with even more features for implementing hybrid GP/RC systems).

The PipeRench system [6] scopes the mapping problem by constraining the solution space to pipelined computations. The hardware resources are virtualized, which relieves the compiler from fixed resource constraints. This research further constrains the solution space to pipelined vector computations, and fixed resource constraints are an input to the mapping process. Both approaches define a parametric architecture template as a target for mapping.

Weinhardt and Luk [7] leverage lessons learned from vectorizing compilers for their pipeline vectorization scheme. Loops in an input source code are transformed to a directed flow graph, where performance-enhancing measures such as chaining found in traditional vector processors are included. This research incorporates chaining in a similar manner, with an additional ability to increase the potential amount of chaining by manipulating pipeline delays. In [8] Paar and Athanas demonstrate the feasibility of implementing a scientific simulation model typical of the legacy codes targeted for vector computers in custom reconfigurable logic.

The starting point of this research is defined with the following assumptions/statements:

1. The input problem definition is a legacy code augmented with pragmas to define areas for custom vector implementation.
2. The input code is already optimized for vector implementation.
3. The runtime bottlenecks in the input code are previously identified, and provide the candidates for custom implementation.
4. The research problem starts with a pseudo-assembly representation of the input problem sections that have been identified for custom implementation with pragmas. The pseudo-assembly emulates a generic register-register vector computer assembly language.

Problem formulation/Approach

The problem statement for this research is as follows: Determine a vector core processor implementation that minimizes wall time to execute N vector instruction sequences, given a set of resource constraints, where wall time is (number of clock cycles)* (clock period).

The formulation of the problem involves the following steps and associated components: 1) Definition and representation of computation to be performed in custom architecture, 2) Definition of architectural template with parameters that can be specified for a particular implementation, 3) A scheduler/mapping algorithm to apply an input computation to an instance of the template and compute performance estimates, 4) An optimization scheme to guide the choice of template instance parameters to meet a performance metric while meeting resource constraints, 5) HDL code generator to produce the hardware implementation of an instance of the architectural template, and 6) A microcode generation scheme to execute the input computation on the architecture instance. A diagram of the process with the steps labeled is shown in Figure 1.

An example of the pseudo-assembly in step (1) is the following vector load from a location in memory to a register.

vl,v1,mem(1),63,X1

In the above example, vl is the opcode for a vector load, v1 is a (virtual) destination register, mem(1) specifies a memory access to address 1, 63 is the vector length of the operation, and X1 is an annotation. The assembler converts the pseudo-asssembly to a Data-Flow Graph (DFG) representation., where the nodes of the graph are vector operations, and the edges are data dependencies.

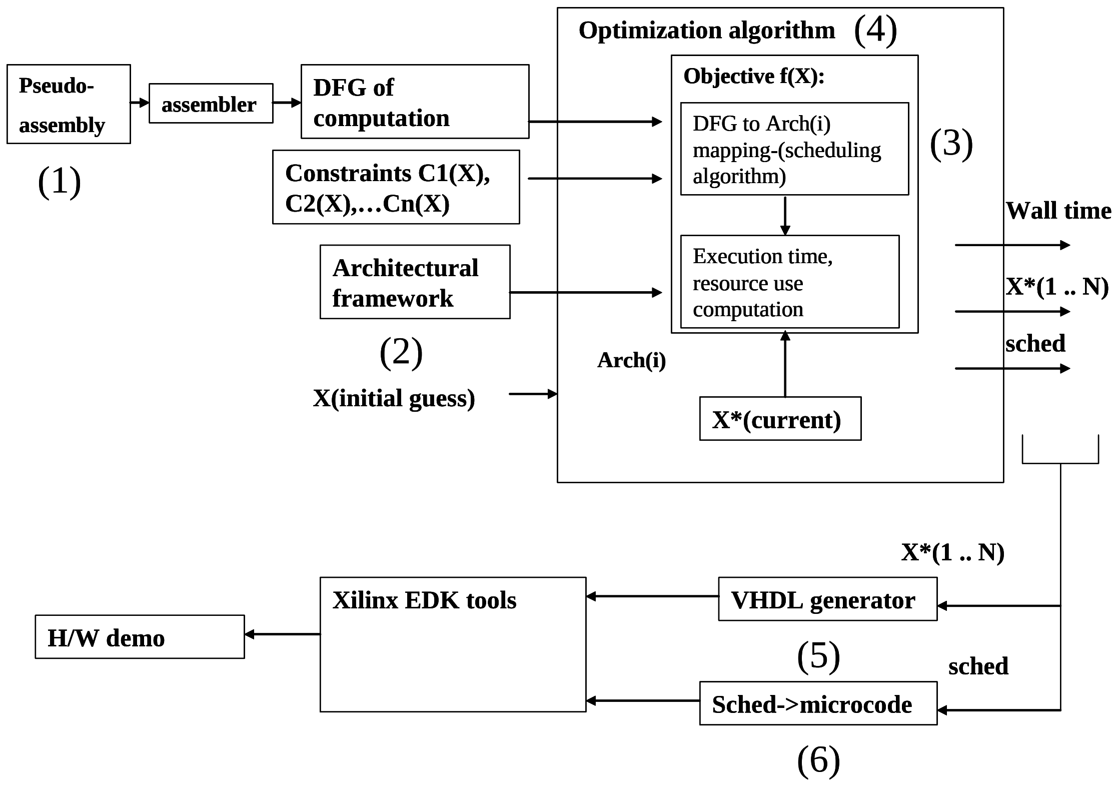


Figure 1: Problem process overview

The architectural template is a vector processing core with a simple interconnect between pipeline vector functional units. The template is specified with a set of parameters that include number of vector load/store units, number of registers, number of floating-point vector adders, number of floating-point vector multipliers, number of interconnect busses, and Maximum Vector Length (MVL). In most traditional vector processors, the MVL is fixed, which determines the number of iterations a vector sequence must be performed to complete a given vector size, for example, if the vector length n > MVL. The ability to vary parameters such as the MVL illustrates capabilities unique to a custom RC implementation. Figure 2 shows the template. Note that many other templates could be used. The template, an initial guess of its parameters X, the input problem DFG, and a set of target implementation-specific hardware constraints provide the input to an optimization loop, (4) in Figure 1.

The objective function f(X) of the optimization loop is a mapping of the problem DFG to an instance of the architecture. The outputs of f(X) are the operation and resource schedules, the number of core cycles to execute, and hardware resource usage for the particular mapping. The current scheduling algorithm is a greedy heuristic; operations whose input data are ready are scheduled to the first available resources until all available resources are exhausted. The optimization algorithm performs non-deterministic "guesses" of the architecture specification X^* that yields the smallest execution time. Simulated annealing is used as the optimization algorithm, but other methods appropriate for discrete input variables to perform constrained minimization could be used. Once an architecture has been determined its parameters provide input to a VHDL autocoder (Figure 1, step (5)) to produce a design of a custom peripheral that will interface to the GP processor. At this point, the target platform must be assumed, and the Xilinx Virtex II pro, with an embedded PowerPC processor, is used as a demonstration for this research. Xilinx Embedded Design ToolKit (EDK) is used to build the demonstration as an embedded PowerPC application with a custom peripheral implemented in configurable logic. More details

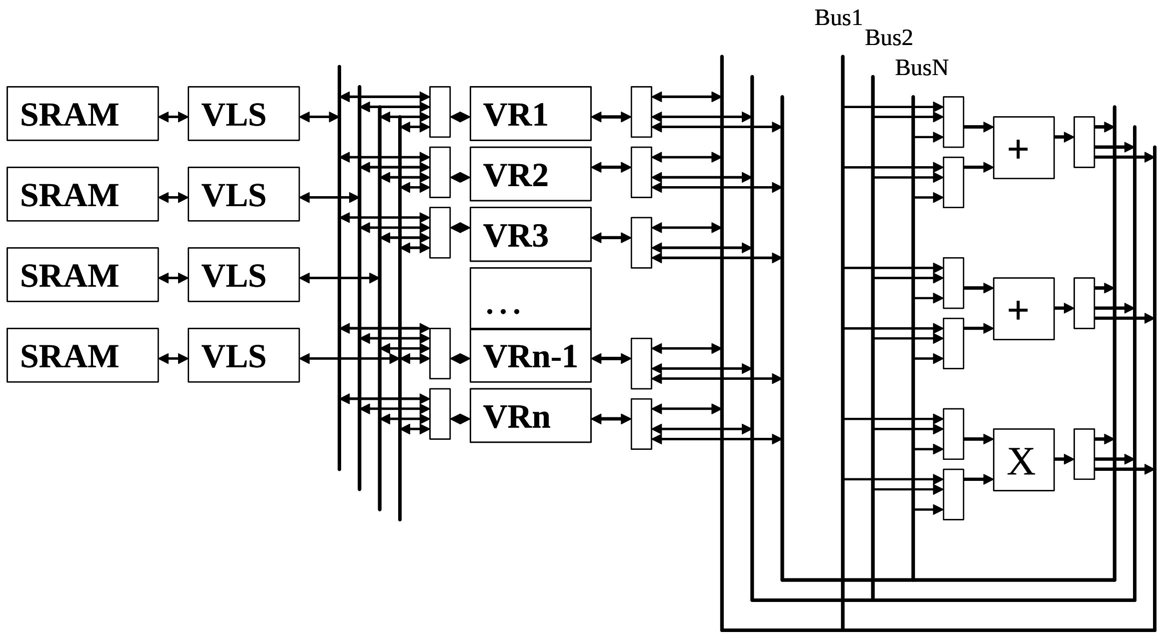


Figure 2: Parametric architecture template for a custom vector processing core

of the demonstration platform are discussed in the next section. Step (6) in Figure 1 is microcode generation from the schedule output of the optimizer. The current vector core design is driven by long microcode instruction words. Current efforts are focused on partitioning this function between hardware and software, and managing the code size. The instruction word has bit fields for each element in the architecture template, so as the template grows to accommodate an input problem on the available resources, the microcode word length increases, which increases the bandwidth necessary between the GP processor and RC fabric.

In order for the mapping algorithm to compute resource usage estimates, the synthesis tools were run for different specifications of architecture parameters and the FPGA resources of interest were measured as a difference from each specification to a baseline. An example of the data collected is shown in Table 1.

Demonstration Platform

Figure 3 is a block diagram of the demonstration design used for this research. The demonstration board is a DN6000K10S prototyping board from the Dini Corporation [9]. Among many interfaces, the board includes four independent SRAM blocks and DDRAM interfaced with a Virtex II pro vp70 chip. The architecture template instance for a particular demonstration is implemented as a custom peripheral in the EDK environment, which interfaces to the PowerPC via the Processor Local Bus (PLB) [10]. The IOCM and DOCM shown in Figure 3 are the Instruction and Data On-Chip Memory, respectively. A UART provides a terminal interface to the embedded C-code application running on the PowerPC. Control words from the PowerPC application interface to the custom peripheral core through a FIFO.

Experiment Design

The experiments chosen to test the research approach are a matrix-by-matrix multiplication and a basic loop from an actual vector code for a

Table 1: Architectural Template Component Resource Usage

	Slices	LUTs	FFs	18X18Mult	BRAM	startup (cycles)	cycles/element	VLIW bits
vp70	33088	66176	66176	328	328	-	-	
X( )								
baseline	3530	4201	3530	4	58			78
(1) VLS	134	235	174	0	0	3	1	
(2) VREG	0	50	50	0	1	5	1	85
(3) VADD	229	140	392	0	0	5	1	90
(4) VMULT	229	140	392	4	0	8	1	90

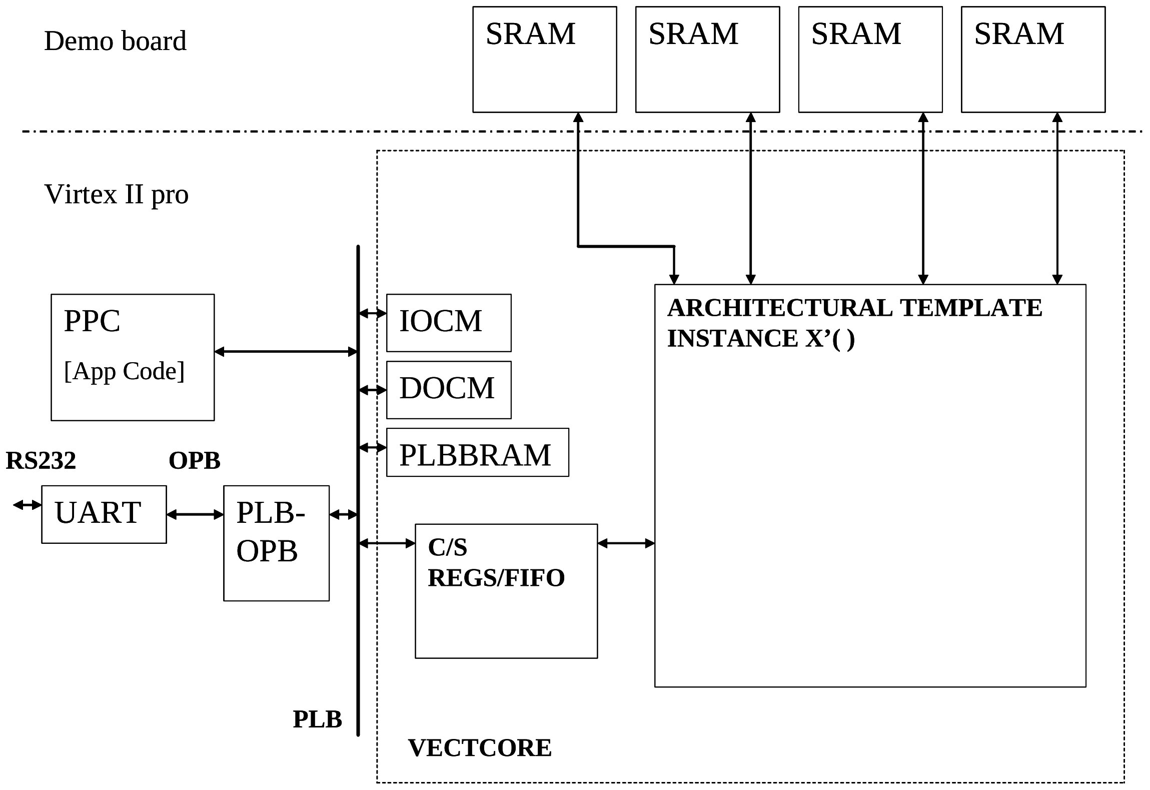


Figure 3: Demonstration board components

weather simulation. The matrix-by-matrix multiplication is a typical operation in many physical models, and it is a computation-bound problem, meaning there are many operations between memory references. The basic weather simulation loop is a known performance bottleneck from NASA Langley's Terminal Area Simulation System (TASS), a 3-dimensional large-eddy simulation used to model various atmospheric events that can present hazards to aviation [11]. TASS was originally coded specifically for the Cray vector architecture. The

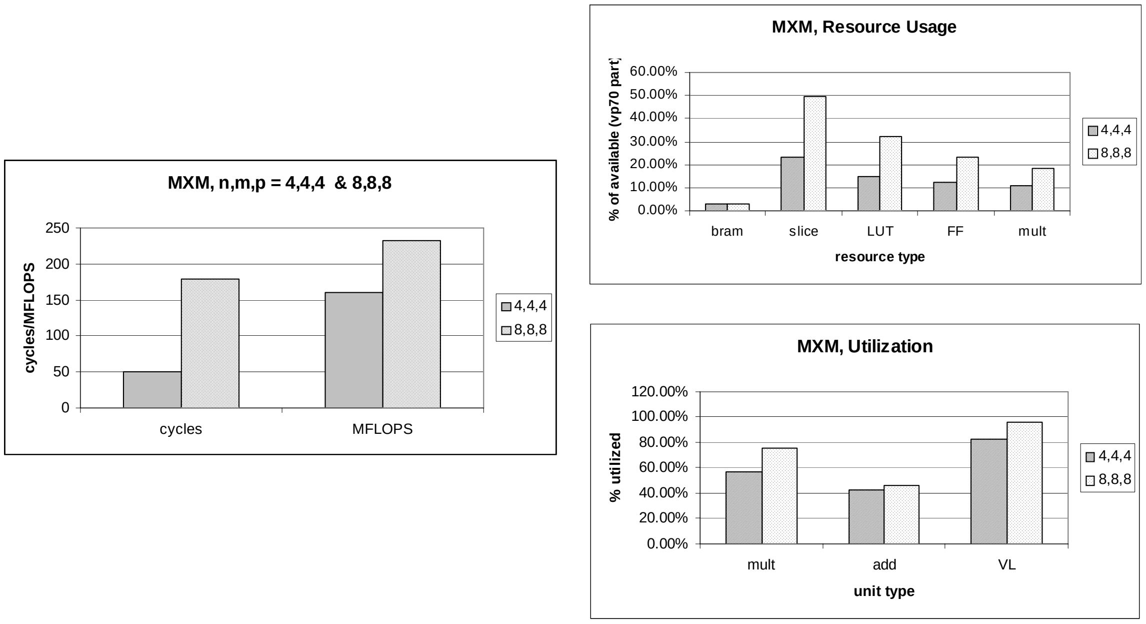


Figure 4: Matrix-by-matrix multiplication results

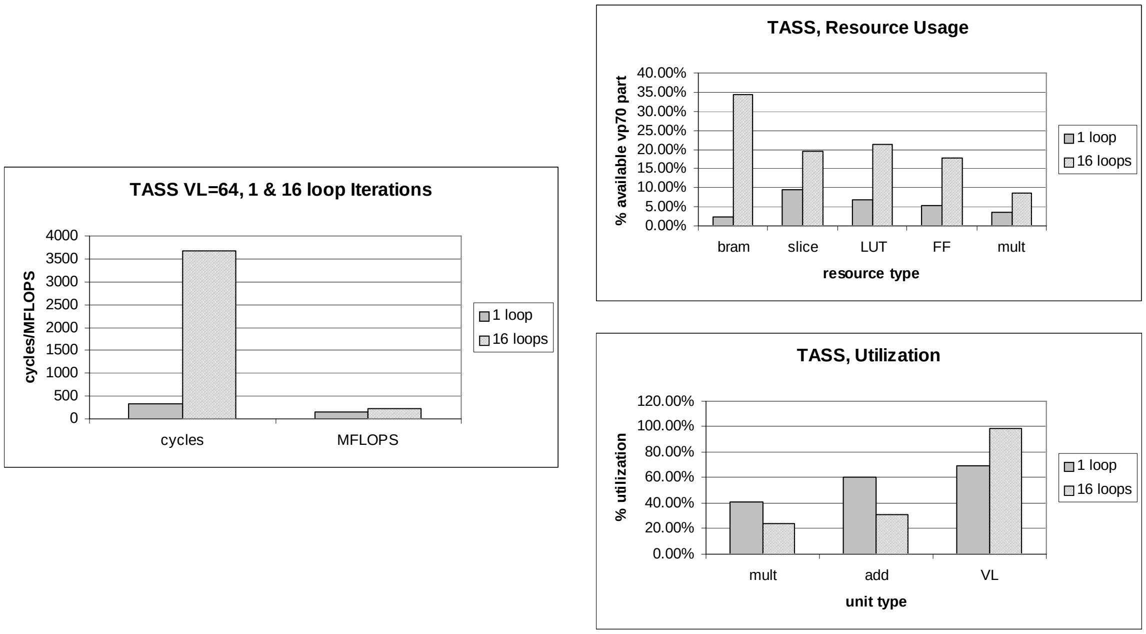


Figure 5: TASS basic loop results, vector length=64

metrics recorded for these test cases are the execution time in peripheral core clock cycles, the floating-point operations per second FLOPS, the resource usage, and the architecture component utilizations. Figures 4 and 5 show the results for the two test cases. The matrix multiply was performed on 16-element (4X4) and 64-element (8X8) matrices.

Discussion

Figure 4 shows that for the two matrix multiplication cases shown, doubling the problem size more than triples the execution time. The compute-bound nature of the problem is implied in the resource usage data, since a small amount of BRAM is used, implying few registers. Also apparent from the utilization chart, the vector load/stores have higher utilization than the other functional units, meaning that the movement of data to and from the core was limiting the speed of the computation. For this example, the number of load/store units was limited to four, to be consistent with the limitations of the demonstration board.

Figure 5 shows one iteration and sixteen iterations of the TASS loop on a 64-element vector. In this case, there is roughly a one-to-one relationship between the execution time and problem size. The FLOPS do not increase with problem size, which suggests a memory-bound characteristic to the problem. This observation is further supported by the resource usage and utilization plots. The register (BRAM) resource increases dramatically with the problem size, and the functional unit utilization actually decrease with problem size when the load/store utilization approaches maximum utilization. Current efforts are to look at larger problem sizes to identify problem size-dependent trends that could be useful data for improving the process.

Future Work/Lessons Learned

Work continues on completion and refinement of the process outlined in Figure 1. The scheduler, autocoders, and assembler designs are being revisited to allow improved performance on larger problem sizes. One major bottleneck in the current approach is the communication of microcode words across the Processor Local Bus from the PowerPC to the vector core. Development is in process to reduce the bandwidth of this interface.

As more problem types and sizes are examined, the goal is to identify the limits of the problem space where this solution approach is effective. Areas for improvement will be identified, and lessons learned documented. Lessons learned to date include the following:

1. *All development tools, including debugging, should be tested in a relevant configuration as early as possible in the process.* The lack of having a consistent design tool flow and debugging resources identified early contributed to major delays in this research.
2. *Identify problems early in test hardware.* Several hardware issues with the demonstration board used for this research caused substantial delays.
3. *Component development should be tested early on" real" problem sizes.* Component development in the mapping process was done by running a prototype on a small test problem. Designs had to be re-visited when issues with running larger problem sizes were identified.

---

[1] G. Estrin, "Organization of Computer Systems-The Fixed Plus Variable Structure Computer," *Proc. of the Western Joint Computer Conf.*, Western Joint Computer Conference, New York, 1960, pp. 33-40.
[2] *Virtex-4 Family Overview*, February 2006, Xilinx Advance Product Specification, DS112. http://direct.xilinx.com/bvdocs/publications/ds112.pdf
[3] Garey, M., Johnson, D., *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, San Francisco, 1979.
[4] J.M. Gokhale, M.B. Stone, "Napa C: Compiling for a Hybrid RISC/FPGA Architecture", *Proc. IEEE FCCM* (1998), pp. 126-135.
[5] *Virtex-II Pro Platform FPGAs: Introduction and Overview*, August 2003, Xilinx Advance Product Specification, DS083-1.
[6] H. Budiu, M. Cadambi, S. Moe, M. Taylor, R.R. Goldstein, S.C. Schmit, "Piperench: A Reconfigurable Architecture and Compiler", *IEEE Computer* 33(2000), no. 4, pp. 70-77.
[7] M. Weinhardt and W. Luk, "Pipeline Vectorization", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20, no. 2.
[8] K. Paar and P. Athanas, "Accelerating Finite-Difference Analysis Simulations with a Configurable Computing Machine", *Microprocessors and Microsystems* 21 (1997), pp. 223-235.
[9] http://www.dinigroup.com/
[10] *Virtex II Pro and Virtex II Pro X Platform FPGAs: Complete Data Sheet*, Xilinx Product Specification, DS083 (V4.0), June 2004.
[11] F. H. Proctor, *The Terminal Area Simulation System. Volume I: TheoreticalFormulation*, NASA Contractor Report 4046 (1987), Available from the National Technical Information Service, Springfield, VA, 22161.

Implementing Scientific Simulation Codes Highly Tailored for Vector Architectures Using Custom Configurable Computing Machines

MAPLD Sept. 24-26,2006
Washington, DC

David Rutishauser
Virginia Polytechnic Institute

Motivation

- Recent push for Massively Parallel (MP) solutions to high-end computing problems such as numerical physical simulations
  - Silicon Graphics MP machines replaced Crays as standard supercomputing resource for NASA
- Large amounts of legacy code exist in government and industry that is highly optimized for vector supercomputers
  - Numerous fluid dynamics, thermal, and structures codes at written for previously predominant Cray resources, e.g. Terminal Area Simulation System (TASS)
- Re-hosting legacy code to an MP platform often requires a complete re-write of the original code; a very long and expensive effort

Motivation, cont.

- Potential exists to use reconfigurable, custom computing resources to emulate key features of a legacy code’s target architecture
- Goal is to achieve useful performance
  - Performance between generic implementation and supercomputer implementation
  - Small augmentations to source code
  - Costs a fraction of a supercomputer

Problem Space

- Prior research shows the issue of designing an optimum hardware implementation of a given application under hardware resource constraints is in a category of generally intractable problems
  - Resource Constrained Scheduling[1]
  - Spatial Partitioning under Constraints[2]
  - Mapping application parts to processing elements[3]
- Approach based on the thesis that the general problem is tractable if domain constrained to a particular computing paradigm
- Vector processing paradigm chosen

1. M. Narasimhan, J. Ramanujam; "A Fast Approach to Computing Exact Solutions to the Resource-Constrained Scheduling Problem“
2. R. Hudson, et al; "Spatio-Temporal Partitioning of Computational Structures onto Comfigurable Computing Machines“
3. M. Asraf, S. Bokhari; “Efficient Algorithms for a Class of Partitioning Problems”

Assumptions

- Approach assumes a source code with minor augmentations (e.g. pragmas)[1]

```
#pragma _CRI prefervector
for (i = 0; i < n; i++) {
#pragma _CRI ivdep
for (j = 0; j < m; j++)
a[i] += b[j][i];
}
```

- Legacy code already has bottlenecks identified
- Legacy code already written for a vectorizing compiler
- Research focused on implementing identified bottlenecks in custom hardware
- Hybrid general-purpose, configurable logic platform targeted

1. Cray Standard C and Cray C++ Reference Manual, 004–2179–005

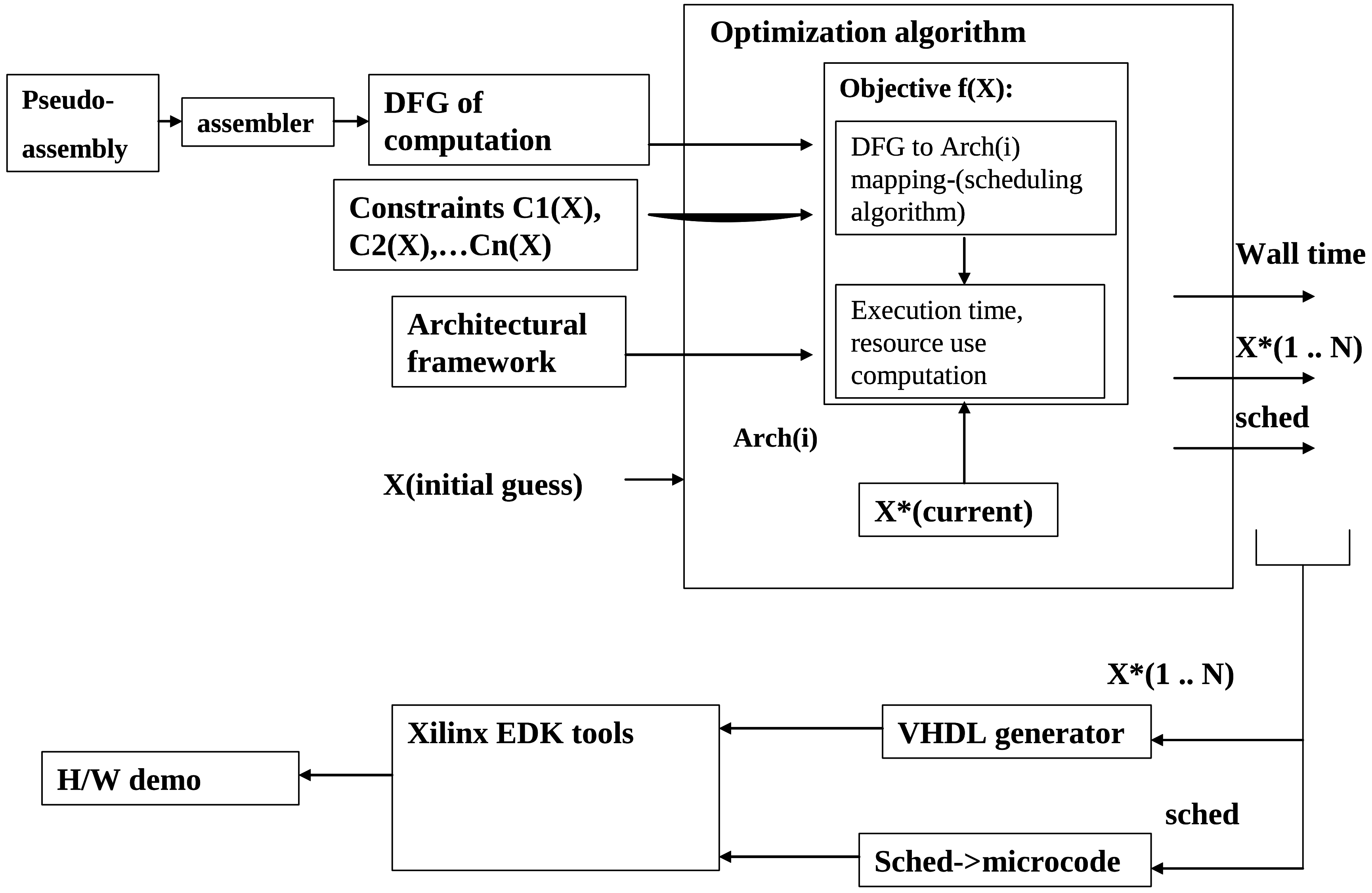
Approach, inputs

- Pseudo vector assembly representation of computation as starting point
- Inputs to optimization process
  - Data Flow Graph (DFG) of computation
  - Target platform constraints (i.e. # of slices, LUTs, etc.)
  - Architectural template (parametric)
  - Starting guess of template parameters

Optimization

- Goals of optimization
  - Determine architecture parameter set for implementation producing minimum execution time
  - Determine schedule for mapping input computation to architecture
  - Solution must remain within hardware resource constraints

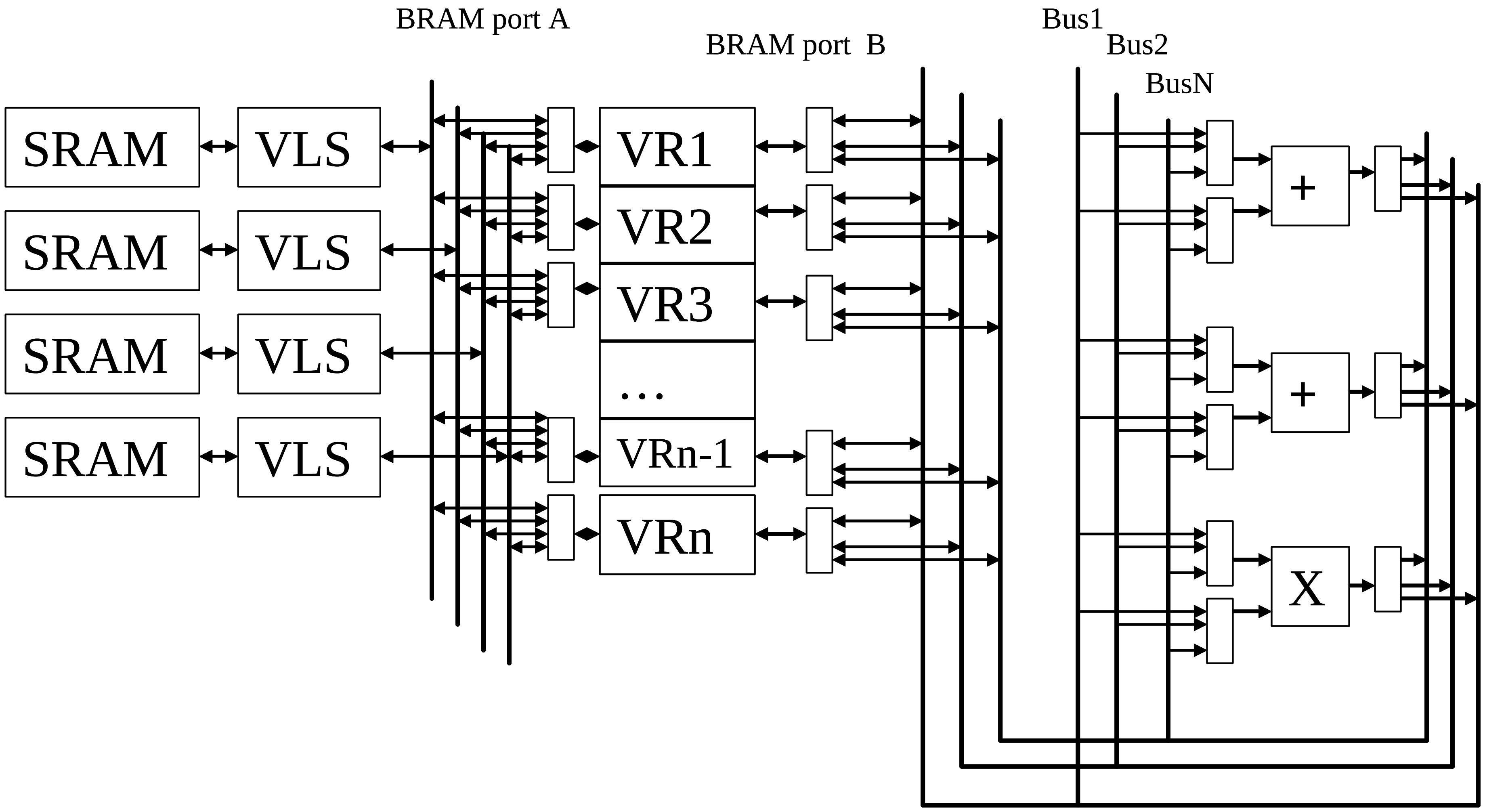
Overview of approach



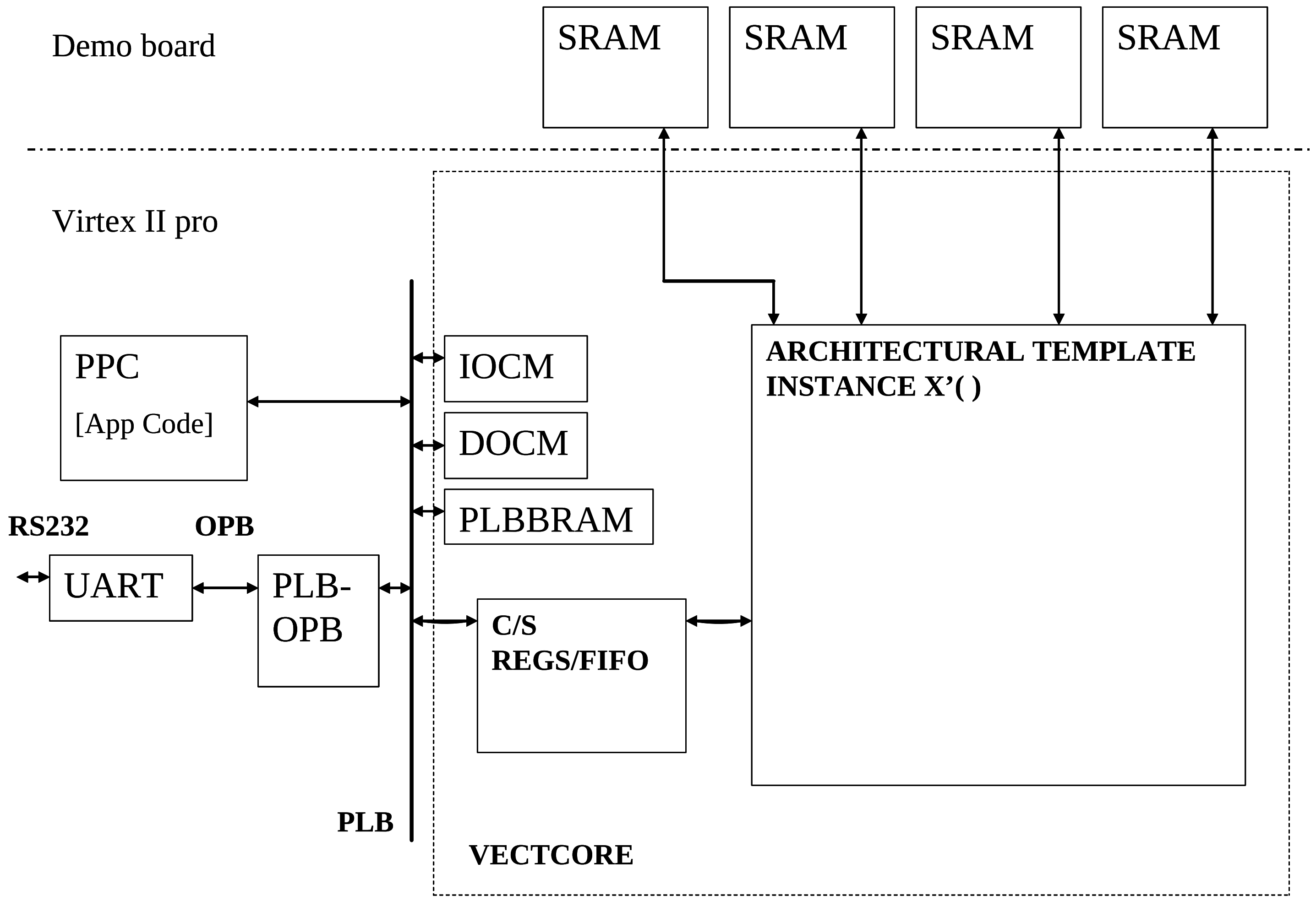
Architecture Template

- Form chosen is vector processing core
  - Assumes a vector register set (Register-Register architecture)
  - Vector functional units
  - Simple bus interconnect structure
- Parameters of core variable
  - # of registers
  - Max vector length
  - # and type of functional units
  - # of interconnect busses
- Other templates could be used

Vector Core Architecture Template

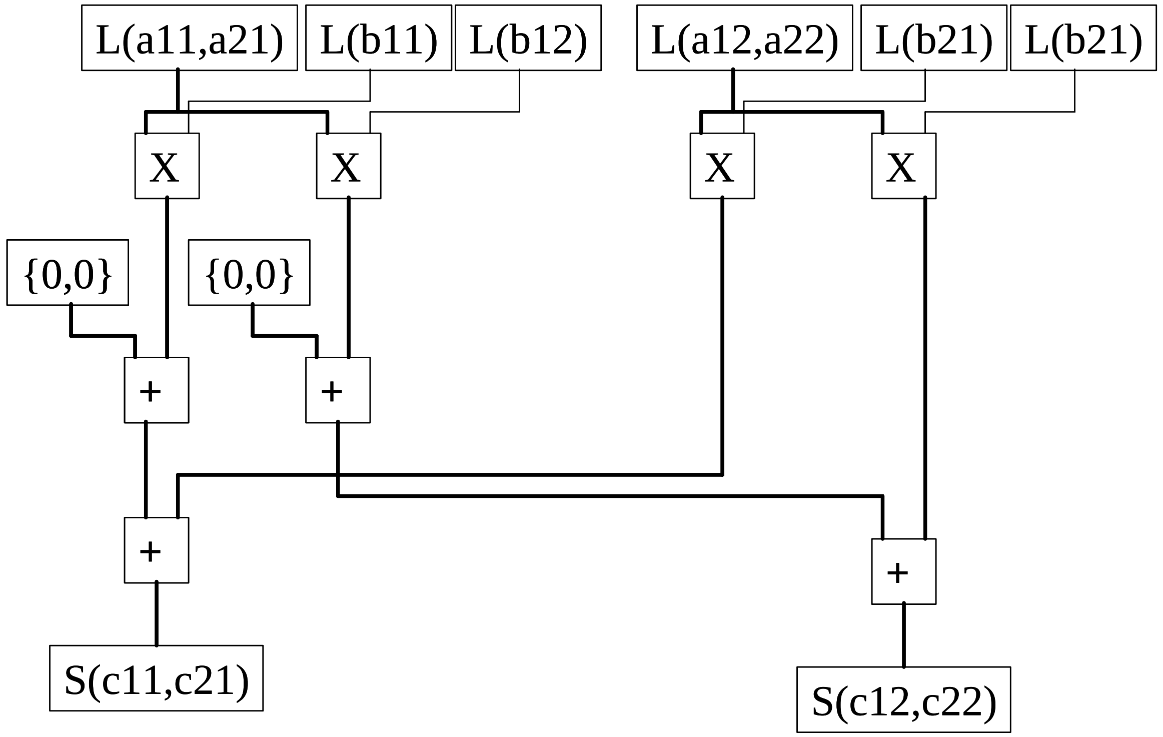


Experimental Architecture



Example

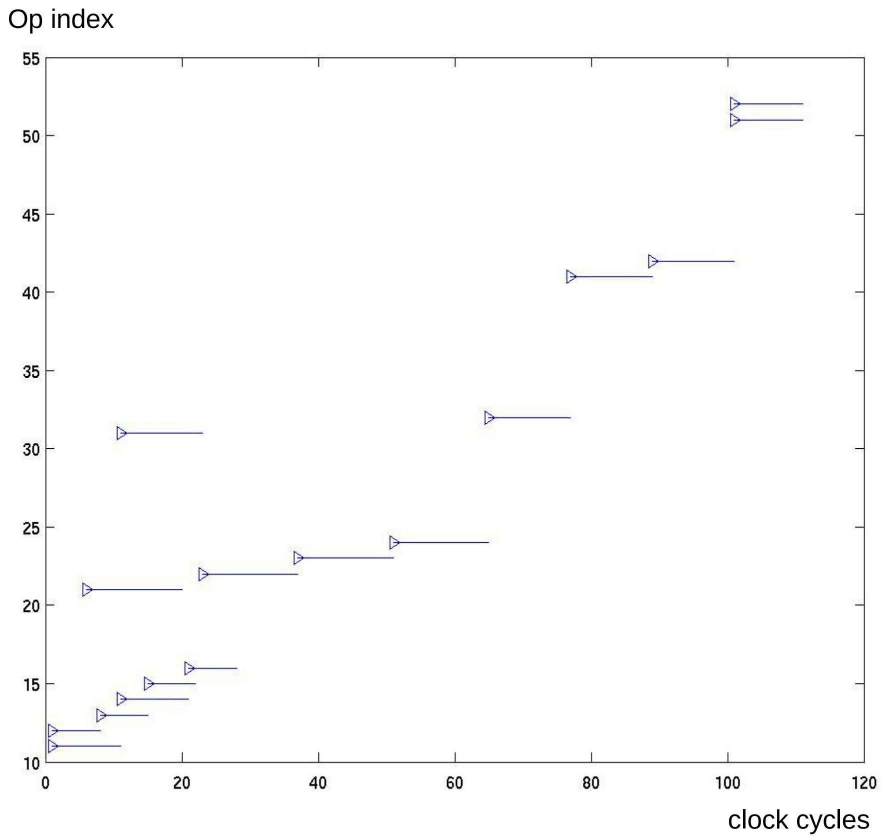
- Matrix Multiply C=AB
- DFG

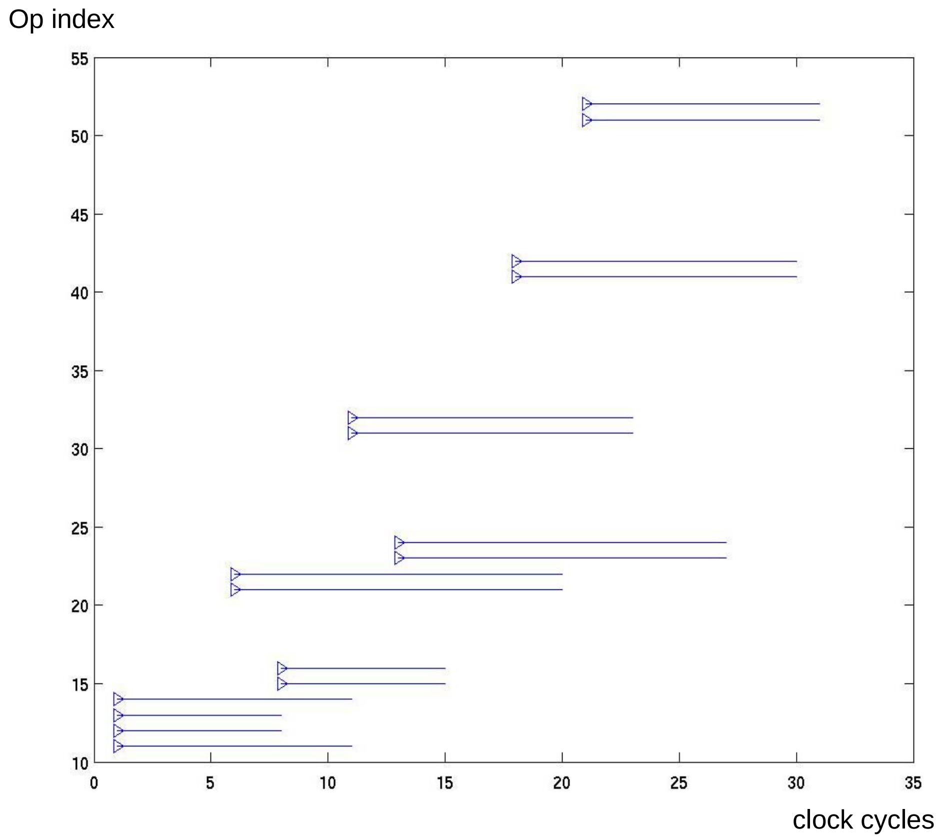


Matrix Multiplication Example

- Initial Architecture parameter guess
  - 2 load/store units, 10 registers, 2 multipliers, 2 adders, vector length of 2, 4 functional unit interconnect busses, and 2 load/Store busses
  - Execution time of 111 core clock cycles
- Post-optimization Architecture parameters
  - 4 load/stores, 8 registers, 4 multipliers, 4 adders, vector length of 2, 16 functional unit interconnect busses, and 4 load/store busses
  - Execution time of 31 core clock cycles

Example Schedule, Pre & Post Optimization





Resource Estimates

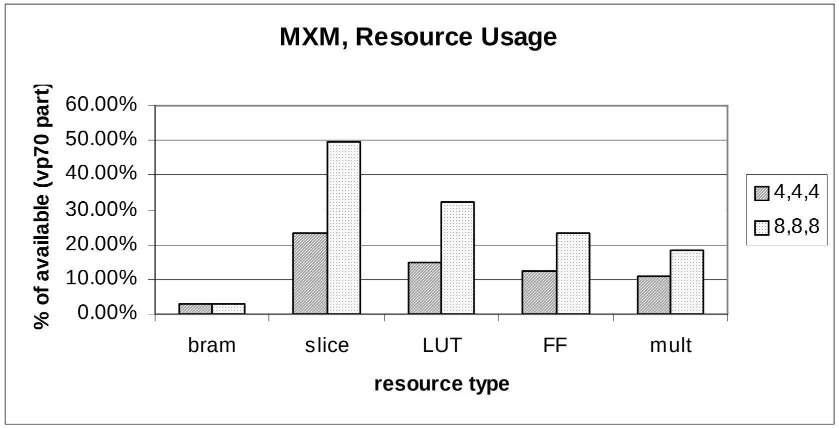
	Slices	LUTs	FFs	18X18Mult	BRAM	startup (cycles)	cycles/element	VLIW bits
vp70	33088	66176	66176	328	328	-	-	
X( )								
baseline	3530	4201	3530	4	58			78
(1) VLS	134	235	174	0	0	3	1	
(2) VREG	0	50	50	0	1	5	1	85
(3) VADD	229	140	392	0	0	5	1	90
(4) VMULT	229	140	392	4	0	8	1	90

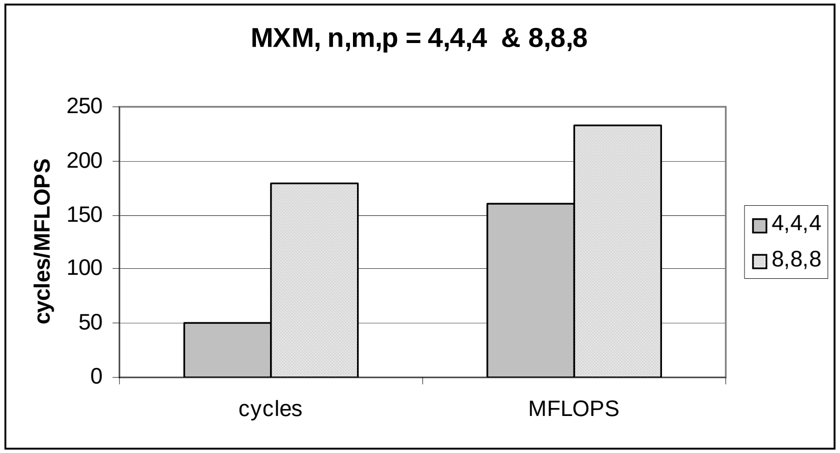
* Baseline includes debug harware

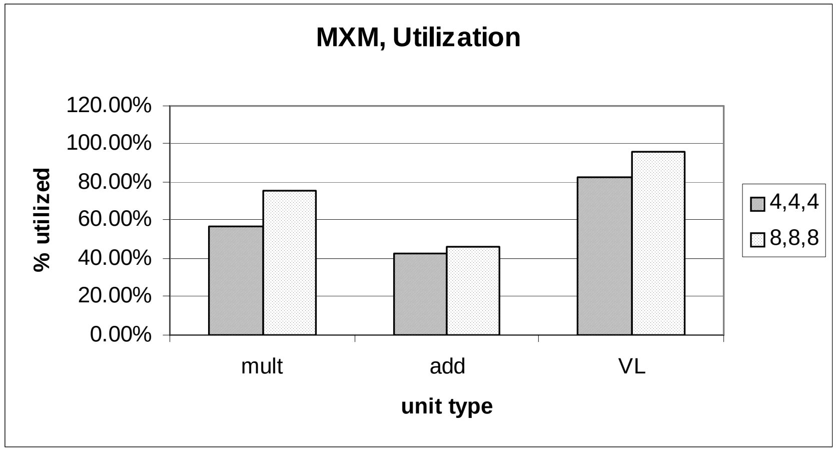
Analysis Cases

- Typical matrix operations
  - NxN matrix-matrix multiplication
    - compute-bound problem, lots of ops per memory op, not many dependencies
- TASS code case study
  - Basic loop from known performance bottleneck

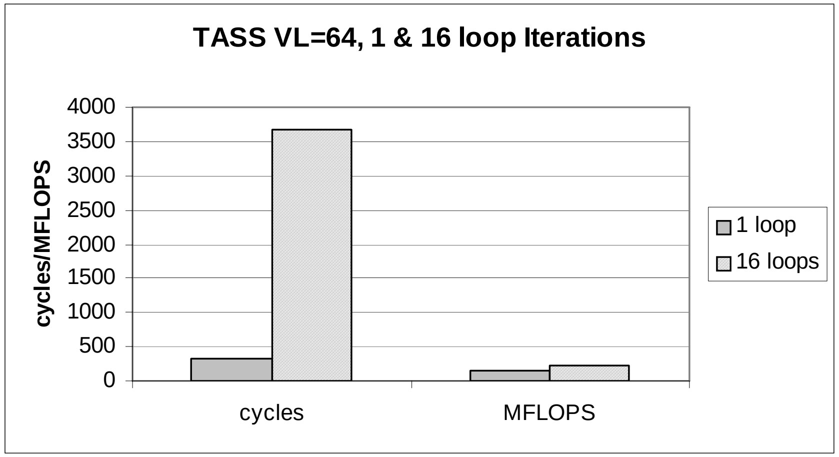
Matrix Multiply, n,m,p=4,4,4;8,8,8

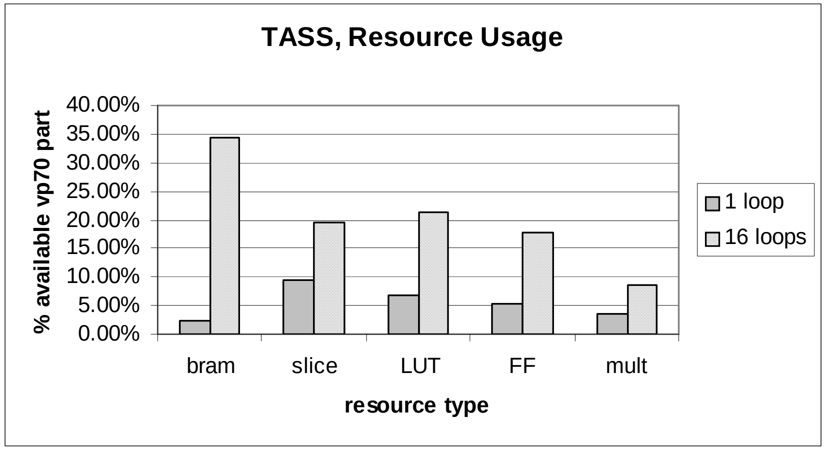


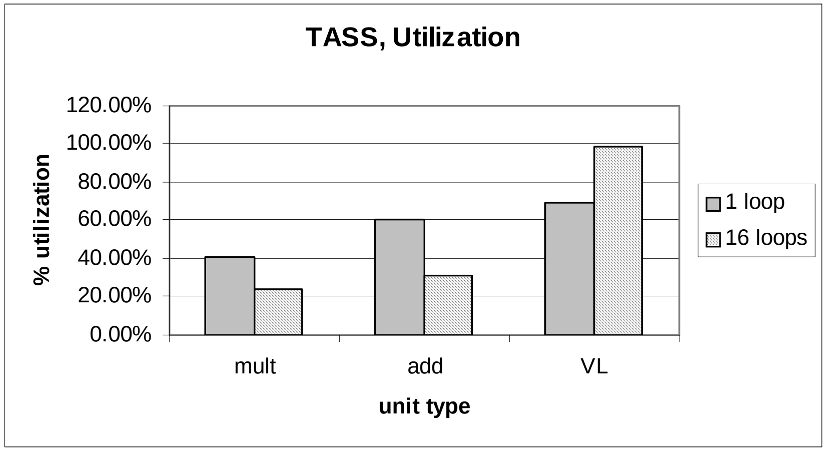




TASS loop. I,J,K= 64,1,1; 64,4,4







Summary

- Research produced a problem formulation and approach for determining an architecture to implement a given computation, while satisfying a performance metric and resource constraints
  - Solution space constrained to a vector processing computing paradigm
  - Approach components:
    - Parametric architectural framework for vector processing that can be implemented in reconfigurable logic
    - Scheduling/mapping algorithm
    - Hardware/microcode generators

Future Work

- Completion of development and test of experimental framework
- Examination of effective problem space/limits of approach
- Documentation of lessons learned and heuristics
- Identification of areas of improvement

Lessons Learned

- *Development tools, including debugging, should be tested in a relevant configuration as early as possible.*
- *Identify problems early in test hardware.*
- *Component development should be tested early on" real" problem sizes.*

Backup Slides

TASS Basic Loop Fortran Code

```
C
C##########################################################################
C CALCULATE U COMPONENT OF VELOCITY
C##########################################################################
#
C
C
C ADVANCE U TO NEXT TIME LEVEL
C
C
DO 11 K=1,KS
DO 11 J=1,JS
DO 11 I=2,IS
X1=U(I,J,K,2)
U(I,J,K,2)=(P(I,J,K,1)-P(I-1,J,K,1))*A(I,J,K)
E(I,J,K)=ALS*U(I,J,K,2)+BTS*X1+U(I,J,K,4)
11 CONTINUE
```

TASS Pseudo-code Example

```
vl,v1,mem(1),63,X1
vl,v2,mem(128),63,A
vl,v3,mem(64),63,P
vl,v4,mem(63),63,Pm1
vmult,v5,v2,v4,63,PA
vadd,v6,v5,v3,63,PS
vl,v7,mem(192),1,ALS
vl,v8,mem(191),1,BTS
vl,v9,mem(193),63,U4
vmults,v10,v7,v6,63,ALSU
vmults,v11,v8,v1,1,BTSX1
vadd,v12,v10,v11,63,ALSUBTSX1
vadd,v13,v12,v9,63,ALSUBTSX1U4
vs,mem(1),v6,63,U
vs,mem(195),v13,63,E
```