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

David K. Rutishauser

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Mark T. Jones, Chair
Peter Athanas
Tom L. Martin
Gary S. Brown
Fred H. Proctor
Richard Barnwell

December 16, 2010
Blacksburg, Virginia

Keywords: Reconfigurable Computing, Field-Programmable Gate Arrays, Vector Computing, Scientific Computing

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

David K. Rutishauser

(ABSTRACT)

Prior to the availability of massively parallel supercomputers, the implementation of choice for scientific computing problems such as large numerical physical simulations was typically a vector supercomputer. Legacy code still exists optimized for vector supercomputers. Rehosting legacy code often requires a complete re-write of the original code, which is a long and expensive effort. This work provides a framework and approach to utilize reconfigurable computing resources in place of a vector supercomputer towards the implementation of a legacy source code without a large re-hosting effort. The choice of a vector processing model constrains the solution space such that practical solutions to the underlying resource constrained scheduling problem are achieved. Reconfigurable computing resources that implement capabilities characteristic of the application's original target platform are examined. The framework includes the following components: (1) a template for a parameterized, configurable vector processing core, (2) a scheduling and allocation algorithm that employs lessons learned from the mature knowledge base of vector supercomputing, and (3) the design of the VectCore co-processor to provide a low-overhead interface and control method for instances of the architectural template. The implementation approach applies the framework to produce VectCore instances tailored for specific input problems that meet resource constraints. Experimental data shows the VectCore approach results in efficient implementations with favorable performance compared to both general purpose processing and fixed vector architecture alternatives for the majority of the benchmark cases. Half the benchmark cases scale nearly linearly under a fixed time scaling model. The fixed workload scaling is also linear for the same cases until becoming constant for a subset of the benchmarks due to resource contention in the VectCore implementation limiting the maximum achievable parallelism. The architectural template contributed by this work supports established vector performance enhancing techniques such as parallel and chained operations. As the hardware resources are scaled, the VectCore approach scales the amount of parallelism applied in a problem implementation. In end-to-end hardware experiments, the VectCore co-processor overhead is shown to be small (less than 4%) compared to the schedule length.

Contents

1 Introduction **1**

2 Background and Related Research **6**

2.1 Reconfigurable Processors for Matrix Operations 8

2.2 Reconfigurable Scientific Applications 9

2.3 Design Space Exploration 10

2.4 Scheduling 12

2.5 Configurable Vector Co-Processors 15

3 Problem Formulation **17**

3.1 Problem Definition 17

3.2 Complexity Analysis 20

4 Approach **23**

4.1 Overview 23

4.2 Approach Inputs 27

4.3 Minimization Algorithm 28

4.4 Architectural Template 30

4.4.1 Interconnect Topology Analysis 34

4.4.2 Memory Interface Analysis 38

4.4.3 Resource Usage Prediction 39

4.5 Scheduling and Allocation 43

4.5.1 Scheduler Algorithm and Heuristics . . . 43
4.5.2 Allocation . . . 47
4.6 VectCore Design . . . 50
4.6.1 Event Dispatch and Control . . . 51
4.6.2 Overhead Analysis . . . 55

5 Experiment Design **57**
5.1 Evaluation System . . . 58
5.2 Benchmark Input Cases . . . 58
5.2.1 Matrix Multiplication . . . 61
5.2.2 Triangular Solve . . . 67
5.2.3 TASS Case Study . . . 69
5.3 Vector Performance Parameters . . . 71

6 Experimental Results **74**
6.1 Test Cases and Nomenclature . . . 75
6.2 VectCore Performance Comparisons . . . 77
6.2.1 VectCore/General Purpose Processor Comparison . . . 77
6.2.2 VectCore/Fixed Vector Architecture Comparison . . . 80
6.2.3 Comparison of VectCore to High Performance Processing Alternatives 85
6.3 VectCore Solution Specificity . . . 90
6.4 Scalability . . . 90
6.4.1 Fixed-Load . . . 92
6.4.2 Scaled Load/Resources . . . 96
6.4.3 Fixed-Resources . . . 98
6.5 VectCore Overhead . . . 98
6.5.1 VectCore Approach . . . 100
6.5.2 VectCore Interface . . . 107
6.6 VectCore Cost Scaling . . . 108

6.6.1 VectCore Code Size Scaling . . . 109
6.6.2 VectCore Time to Solution . . . 109
6.7 Potential Improvements . . . 110
6.8 Custom Instruction Evaluation . . . 112

7 Concluding Remarks 117
7.1 Summary . . . 117
7.2 Future Work . . . 120

References 122

A Implementation Details 135
A.1 Scheduler Example . . . 135
A.2 VectCore Assembly Syntax Description . . . 137
A.3 S-PAK Word Format . . . 139
A.4 Constraint Function Development . . . 141
A.5 Manually Generated Architecture Specifications . . . 142
A.6 C Implementations of Benchmark Problems . . . 142
A.7 Global Clock Control . . . 145

B Verification 147
B.1 VectCore Allocation . . . 147
B.2 Scheduler Operation . . . 147

C Cray Performance Benchmarks 149
C.1 Cray Hardware Performance Monitor description . . . 149
C.2 Beltrami Benchmark . . . 151
C.3 UWP Subroutine Benchmark . . . 153
C.4 Typical Loop Benchmark . . . 156
C.5 CMNLOOP List File . . . 157

D Background on Vector Processing **159**
D.1 Performance Issues in Vector Computers 160
D.1.1 Startup Latency 160
D.1.2 Stripmining 161
D.1.3 Vector Stride 162
D.1.4 Vector Chaining 163
D.2 Vector Compilers 163

E Case Study Application **165**

List of Figures

1.1 A problem formulation and approach to solving the problem are contributed by this research. The approach accepts a target computation and a resource limit as inputs, and produces a VectCore processor instance tailored for the computation, with a schedule for executing the computation on the tailored processor. Specific research contributions are identified by numbered labels in the figure. 3

2.1 Overview of the list scheduling algorithm. For each time index, a set of all tasks ready for execution is formed. Using a heuristics-based priority function, the next task is selected. The current time step is assigned to the selected task, and resources are allocated for its execution. The time index is advanced, and the algorithm repeats until all tasks are scheduled. 14

3.1 Example of architecture specification vector. 18

3.2 Example of task graph for input computation $E = C(A+B)+D$. The vertices represent operations or tasks, and edges denote data dependencies. 19

4.1 Solution system components and functions. Tools on a development workstation generate a tailored architecture instance, and a mapping to the architecture, for sections of a software application targeted for acceleration. Implementation tools produce a file to configure the FPGA for an architecture instance. A general-purpose processor runs the original software portions of the application, and provides an interface for microcode programs that run the accelerated application sections on the FPGA. 25

4.2 Algorithmic approach overview. A minimization algorithm iterates on specifications for architecture instances to minimize the objective. Each architecture instance, X^i is the target for a mapping algorithm that determines a schedule for the operations in the input computation, G, on X^i. The approach outputs are the architecture specification, X^* that minimizes the objective, and the schedule for executing the computation G on the architecture specified by X^*. Implementation tools use these outputs to produce a VectCore co-processor instance that runs the input problem. 27

4.3 Example penalty function, p(X). 29

4.4 Parameterized architecture template for VectCore custom vector processing core. The memory modules are external to the FPGA. 31

4.5 Example VectCore architecture template. Specification includes $4L$, $5V$, $1A$, $1M$, $3B$, $1Y$, and $0I$. 32

4.6 Example of chaining on VectCore architecture template for operation $D = (A + B)C$. Allocation of template resources is shown in part (A), where superscripts on the resource labels indicate data allocation. Functional unit allocation is indicated by labels over the functional units. Part (B) shows the same allocated resources in a data flow representation of the chained operation. 35

4.7 VectCore scheduling heuristic. A list scheduling algorithm is used with a priority function that assigns weights to tasks in the ready set. The weighted ready set is sorted by decreasing weight to produce the pick set, θ_p. Resources are assigned to each pick set operation using a greedy heuristic. After assessing the need to spill a register, each operation is checked for potential chain formation with its inputs. Chain failures cause the inputs for an operation to be stored in registers. 45

4.8 VectCore allocation example. Part (A) is a task graph for an example computation, and part (B) shows the allocation of resources to the tasks for an architecture specification $1L, 3V, 2A, 1M, 3B, 0Y, 0I$ 49

4.9 VectCore approach hardware implementation. 51

4.10 VectCore interface and S-PAK dispatch scheme. 52

4.11 Global clock control operation description. A new schedule event, which includes a starting time and a resource configuration, is read from the event FIFO. If the global clock has reached the next event start time, the event bit-field is compared to the resource bit-field. Otherwise, the global clock is incremented. If all resources scheduled for the next start time are ready, the resource bit-field matches the event bit-field and the "go" signal is asserted. If the resources are not ready, the "hold" signal is asserted to stall all active processing resources until all resources executing on the next schedule event indicate ready. 54

5.1 VectCore evaluation system organization. 59

5.2 Three orderings for the matrix multiplication. 62

5.3 Example *SRAM* organization for *jik* matrix multiplication. The nomenclature $a_{1_}$, for example, denotes the first row of matrix a. It is assumed that the compiler for a legacy application targeted by the VectCore approach stores multi-dimensional arrays in column-major order, so the data for the columns of the operand or result matrices occupies contiguous addresses. A buffer for register spills is allocated in each *SRAM* bank. 63

5.4 Partial task graph for *jik* matrix multiplication, with o_m, o_n, o_p = 4, 4, 4. . . 64

5.5 Partial task graph for *kji* matrix multiplication, with o_m, o_n, o_p = 4, 4, 4. . . 65

5.6 Partial task graph for *jki* matrix multiplication, with o_m, o_n, o_p = 4, 4, 4. . . 66

5.7 Task graph for dimension 4 x 4 upper triangular solve, inner product operation. 68

5.8 Task graph for dimension 4 x 4 upper triangular solve, *SAXPY* operation. . 69

5.9 Partial task graph for TASS loop, with j, k, i = 2, 2, 4. 70

5.10 A portion of a *jki* schedule illustrating the potential for parallel independent vector chains. 73

6.1 FLOPS performance for VectCore and desktop implementations. Workload and architecture limits are scaled linearly (p1a1, p2a2, p3a3). 78

6.2 VectCore performance: (a) average number of parallel floating-point operations, (b) average number of vector chains, and (c) average ready set size. Workload and architecture limits are scaled linearly (p1a1, p2a2, p3a3). . . . 79

6.3 FLOPS performance delta (a) and floating-point percent utilization (b) for VectCore and fixed vector implementations. Workload and architecture limits are scaled linearly (p1a1, p2a2, p3a3). 81

6.4 VectCore to fixed vector performance comparison: (a) FLOPS performance, (b) average number of parallel floating-point operations, (c) average number of vector chains, and (d) number of spills. Workload and architecture limits are scaled linearly (p1a1, p2a2, p3a3). 83

6.5 (a) Comparison of measured VectCore FLOPS performance to predicted FLOPS performance, and (b), portion of schedule for *kji_saxpy* p3a3 problem. . . . 84

6.6 FLOPS, parallel vector, and chained vector performance across problem types for fixed workload size p2, and scaling resource limit. 93

6.7 VectCore fixed-time scaling performance: (a) FLOPS performance, (b) execution time in VectCore clock cycles, (c) VectCore register specification, and (d) number of spills. Workload and architecture limits are scaled linearly (p1a1, p2a2, p3a3). 97

6.8 VectCore fixed-architecture scaling performance: (a) FLOPS performance, (b) average number of parallel floating-point operations, (c) average number of vector chains, and (d) number of spills. Workload is scaled linearly (p1a1, p2a1, p3a3). 99

6.9 Scaling of dominant VectCore overhead components as a function of the number of floating-point units. 101

6.10 VectCore resource overhead components for the jik and jki matrix multiplication problems, with various problem size and architecture limits. 102

6.11 Number of S-PAKs, fixed-time and resource scaling. 109

6.12 Architecture parameter ratios for each problem type under fixed-time scaling. 112

6.13 VectCore FLOPS performance comparison to parameter ratio-derived implementations under fixed-time scaling. 113

6.14 Custom *tass* instruction performance comparison: (a) percent difference in FLOPS performance, (b) predicted and measured FLOPS performance for the *tass* and *ntass* implementations, (c) average parallel operations, and (d) average chained vector operations. 115

7.1 Potential VectCore approach applied to a large application. An optimization of the parameters of the VectCore processor considers N task graphs from an input application. Heuristic weights for each task graph are applied to control the affect a particular graph has on the outcome of the architecture optimization. 121

A.1 Annotated data flow graph for one *tass* loop iteration. 136

A.2 VectCore assembly syntax example. . . . 139

A.3 Update register syntax example. . . . 139

A.4 S-PAK word format, bits 63-32. . . . 140

A.5 S-PAK word format, bits 31-0. . . . 140

A.6 S-PAK 2nd word format, bits 31-0. . . . 141

A.7 S-PAK SRAM test word format, bits 63-32. . . . 141

A.8 Data used to compute constraint function constants C_q and C_B. . . . 142

A.9 Spreadsheets for design of (a) fixed and (b) ratio-scaled VectCore implementations. . . . 143

A.10 C-code listings for benchmark implementations. . . . 144

A.11 Global clock control finite state machine description. . . . 146

B.1 Test coverage of VectCore allocation capabilities and scheduler logic paths. . . 148

B.2 Test coverage for benchmark problem scheduler verification. . . . 148

C.1 HPM Group 0 report for Beltrami benchmark. . . . 151

C.2 HPM Group 1 report for Beltrami benchmark. . . . 151

C.3 HPM Group 2 report for Beltrami benchmark. . . . 152

C.4 HPM Group 3 report for Beltrami benchmark. . . . 152

C.5 Beltrami benchmark output file. . . . 153

C.6 List file example, *uwp* subroutine. . . . 154

C.7 Group 0 report for greater-than MVL *UWP* benchmark. . . . 154

C.8 HPM Group 0 report for less-than MVL *UWP* benchmark. . . . 155

C.9 HPM Group 0 report for multiple of MVL *UWP* benchmark. . . . 155

C.10 Typical loop benchmark, *UWP* subroutine. . . . 156

C.11 HPM Group 0 report for greater-than MVL loop benchmark. . . . 156

D.1 Pseudocode comparison of non-stripmined (a), and stripmined (b) vector code. 162

D.2 Vectorizing compiler components [1]. . . . 164

E.1 TASS grid scheme. . . . 167

List of Tables

4.1 VectCore vector unit number of pipeline stages. 33

4.2 Comparison of interconnect topologies. 37

4.3 VectCore per-unit resource cost in slice units. 40

4.4 VectCore resource estimates, using the Virtex II pro^{TM}vp100 part as the target for synthesis. 42

5.1 FPGA and EDK design configuration data. 59

6.1 Test case problems with nomenclature. 75

6.2 Test case workload size nomenclature. 76

6.3 Workload and architecture limit combinations for test cases. 76

6.4 Average benchmark problem vector lengths for the VectCore implementations, across problem sizes p1, p2, and p3. 80

6.5 Live value *kji_saxpy* problem requirements and register specifications for the VectCore and fixed vector implementations. 82

6.6 VectCore performance and price per performance comparison to implementation alternatives. 87

6.7 VectCore performance of each tailored problem implementation for each benchmark problem. Problem size and architecture limits are p1a1, p2a2, and p3a3. 91

6.8 VectCore load/store and functional unit specification-derived supported number of concurrent operations and measured average number of parallel operations for *tass* problem size p2 and scaled resource limit. 95

6.9 VectCore combined average parallel and chained operations as a percentage of the maximum DFG level size under fixed workload scaling. 95

6.10 VectCore average number of active load/store units as a percentage of the maximum DFG load/store level size under fixed workload scaling. 96

6.11 Average vector length for the VectCore triangular solve implementations, with measured and predicted percent FLOPS increase between problem sizes p1 and p2. 99

6.12 VectCore FLOPS performance as a percentage of the theoretical upper FLOPS bound supported in a given architecture limit. 104

6.13 Comparison of percent theoretical FLOPS performance limit for the VectCore and other implementation options. 105

6.14 MTASS hardware performance data for scaled workload and fixed resource limit (a1). 108

6.15 FLOPS utilization and performance changes for the *jik_ip* problem after manual adjustments to the architecture specifications. 111

6.16 Custom *tass* instruction architecture comparison. The parameters listed determine the amount of parallel and chained operations an architecture will support. 116

A.1 Scheduler operation example. 136

A.2 VectCore scheduler heuristic weights. 137

A.3 Pseudo-assembly operation mnemonics. 138

C.1 Cray SV1 system counters descriptions. [2] 150

D.1 Example vector assembly instruction. 160

E.1 TASS module descriptions. [3] 168

Glossary

A number of vector adder units in X.

B number of functional unit buses in X.

C_0 resource cost for the basic fixed portion of the VectCore design.

C_B resource cost per input for a VectCore B-input multiplexer.

C_q resource cost per input for a VectCore q-input multiplexer.

C constraint limit for thesis minimization problem.

D time index deadline for thesis minimization objective function mapping.

F objective function for thesis minimization problem.

G task graph for task set T.

I number of vector inner product units in X.

L_d latency between VectCore event read and resource start.

L_s latency between S-PAK writes from general-purpose processor to VectCore.

L number of vector load/store units in X.

M number of vector multiplier units in X.

O_c number of vector chains per vector sequence.

O_f number of floating-point operations per vector sequence.

O_p average number of parallel vector operations.

O_v number of vector operations per vector sequence.

P set of valid processing element mappings for T.

R matrix of quantities of resource types consumed by T.

$SAXPY$ sum of A times X plus Y.

T_c average vector chain length.

T_s pipeline startup latency for a vector operation.

T set of tasks in multiprocessing model for general scheduling problem.

U total number of vector floating-point units in X.

V number of vector register units in X.

W memory interface bandwidth.

X objective function input vector defining an architecture parameter specification.

Y number of vector $SAXPY$ units in X.

Γ allocation for the resources in X^* corresponding to a given schedule ϕ .

ω memory interface/ VectCore processor frequency.

ϕ_i starting time for task T_i of task graph G.

σ valid schedule function for input to the general scheduling problem.

τ set of durations for members of task set T.

θ_p scheduler ready set sorted by decreasing heuristic weight.

e_1 first event start time in a given schedule ϕ .

e_2 second event start time in a given schedule ϕ .

g constraint function for thesis minimization problem.

h_d number of descendants heuristic.

h_i loop iteration heuristic.

h_l in last ready set heuristic.

h_p path length heuristic.

h_s operation type heuristic.

l_p task graph path length of task i.

n_d task graph number of descendants of task i.

n number of tasks in task set T.

o_m for matrix multiplication $y = ax$, number of rows in y.

o_n for matrix multiplication $y = ax$, number of rows in x.

o_p for matrix multiplication $y = ax$, number of columns in y.

p penalty function for implementation of thesis minimization problem.

q number of inputs and outputs to an interconnect network.

v_l vector length.

w interconnect link data width in bits.

ALAP As Late As Possible.

API Application Programming Interface.

ASAP As Soon As Possible.

BLAS Basic Linear Algebra Subprograms.

BRAM Block Random Access Memory.

DFG Data Flow Graph.

DRAM Dynamic Random Access Memory.

EDK Embedded Development Kit.

FIFO First In First Out.

FLOPS Floating-Point Operations Per Second.

FPGA Field-Programmable Gate Array.

GPP General Purpose Processor.

GPU Graphics Processing Unit.

HDL High-Level Design Language.

HPC High Performance Computer.

ISA Instruction Set Architecture.

LUT Lookup Table.

PLB Processor Local Bus.

S-PAK Schedule-Packet.

SerDes Serializers/Deserializers.

SIMD Single Instruction Multiple Data.

SRAM Static Random Access Memory.

TASS Terminal Area Simulation System.

VLIW Very Long Instruction Word.

Chapter 1

Introduction

The motivation for this work comes from an observation that amidst the push for massively parallel solutions to high-end computing problems such as numerical physical simulations, large amounts of legacy code exist that are optimized for vector supercomputers. Re-hosting legacy code often requires a complete re-write of the original code. This work provides a framework and approach to utilize reconfigurable computing resources in place of a vector supercomputer towards the implementation of a legacy source code without a large re-hosting effort. Applying reconfigurable computing resources to implement capabilities of the application's original target platform results in a scalable implementation that utilizes available resources efficiently.

Reconfigurable computing has been an active area of research from the introduction of its concepts [4] to the present day when enabling advancements in configurable processing elements [5] have allowed for a wide variety of experimentation. At the center of reconfigurable 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 believed not to have tractable solutions [6]. To solve practical instances of such problems, constraints on the

problem scope and heuristics are 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, and many other problems suited to vector implementation. Lessons learned from years of vector computer development are leveraged in the context of the flexibility of a reconfigurable implementation. The thesis of this research is the following.

The use of reconfigurable computing resources to implement the vector processing capabilities of an application's target platform, tailored for the specific application, results in an efficient, scalable implementation.

The contributions of this research are summarized as follows, referring to Figure 1.1.

- A formulation of the thesis problem as a multivariable minimization problem with constraints (item (1) in Figure 1.1) is presented in Chapter 3. The inputs are a set of vector computations defining the problem, and a set of parameters that specify an instance of a parameterized processing architecture. The objective function for the minimization is a scheduling algorithm that provides a mapping of the input vector operations to the resources of a particular architecture specification. The minimization is subject to a resource limit that constrains the range of the parameters in an architecture specification. The problem formulation is suitable for incorporation into optimization algorithms. The formulation enables a tradeoff of cost versus performance that produces efficient architectures exhibiting high computational rates while remaining within given resource constraints. These architectures are tailored for the input computation.

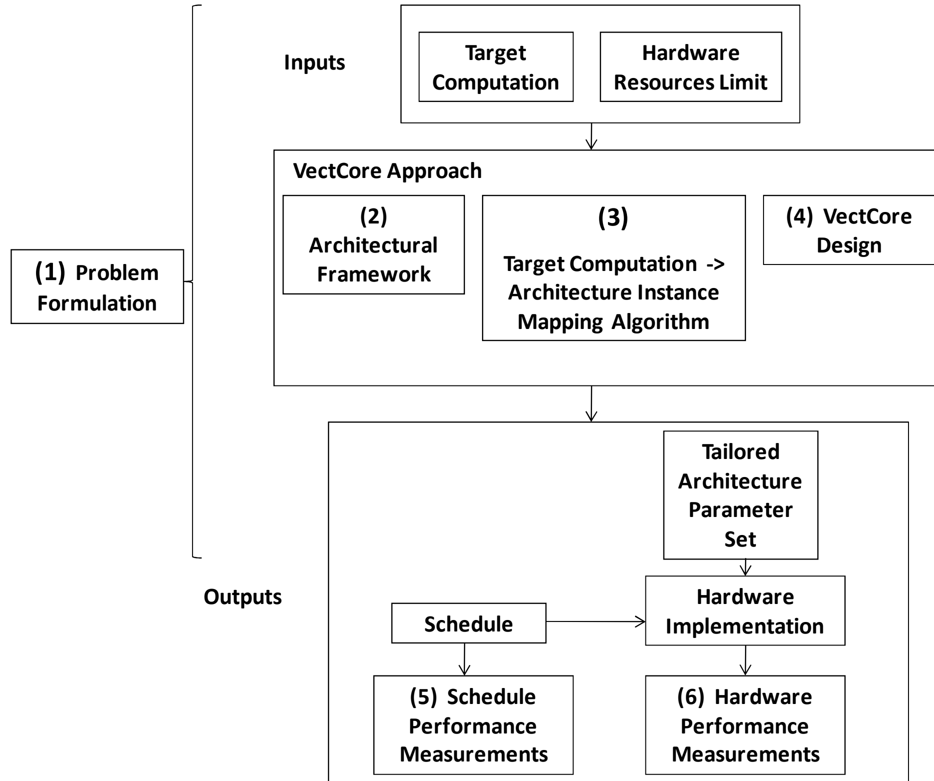

Figure 1.1: A problem formulation and approach to solving the problem are contributed by this research. The approach accepts a target computation and a resource limit as inputs, and produces a VectCore processor instance tailored for the computation, with a schedule for executing the computation on the tailored processor. Specific research contributions are identified by numbered labels in the figure.

- An algorithmic approach to solve the thesis problem is provided in this work. This research contribution includes the following components.
  - A parameterized architectural framework for vector processing provides the target for the objective function mapping algorithm (item (2), Figure 1.1). The framework is capable of executing parallel and/or chained pipelined vector operations and performing parallel memory accesses. The parameterized design facilitates tradeoffs between memory bandwidth, parallel floating-point operations, interconnect bandwidth, and temporary storage. The parameter set enables tailoring of a framework instance to the structural characteristics of an input problem. Section 4.4 discusses significant attributes of the architectural framework design.
  - A scheduling/mapping algorithm (item (3), Figure 1.1) is presented as the objective function for the thesis problem. The algorithm uses heuristics that leverage lessons learned from the large volume of research available on scheduling (Section 4.5.1). The algorithm effectively uses the flexibility of the architectural template to provide solutions that benefit from established performance-enhancing practices in vector computing. Details of the scheduling/mapping algorithm are provided in Section 4.5.
  - This research contributes the design of the VectCore co-processor (item (4), Figure 1.1). The VectCore provides an interface between the architectural framework instance and a general purpose microprocessor, and the control of the framework instance during problem execution. The design features a dispatch and control approach that ensures proper implementation of a scheduler solution while maintaining low overhead as compared to the schedule length (Section 4.6).
- This research contributes an assessment of the algorithmic approach using experimental data (items (5) and (6) in Figure 1.1) that includes data produced from an end-to-end

implementation in hardware. Chapter 5 describes the experimental design. Chapter 6 provides results that substantiate the research contributions, including the following assertions.

- The approach produces implementations competitive with the performance and efficiency of alternative techniques.
- Implementations produced using the research approach perform well under scaling of workload and resource limits.
- The approach is practical in terms of code size and time to solution.

As an application case study, a large finite-difference weather simulation (the Terminal Area Simulation System (TASS)) optimized for execution on a Cray vector supercomputer is used for experimentation [7]. NASA removed the Cray computers from its supercomputing inventory, opting for massively parallel systems such as those produced by Silicon Graphics. Re-writing the TASS code to run on the new machines has taken a highly qualified scientist one full man-year to reach a version of the code with limited capabilities. TASS is just one example of many other simulator codes optimized for vector architectures in existence.

In summary, the report is organized as follows. Chapter 2 includes background on reconfigurable computing and research related to the thesis of this work. The thesis problem is formulated in Chapter 3. Chapter 4 provides details of the algorithmic approach contributed by this research for implementation of the thesis problem. The design of the experiments to evaluate the algorithmic approach is provided in Chapter 5. Chapter 6 presents the results of the experiments, and Chapter 7 discusses conclusions and future work. The Appendices include design details for the algorithmic approach (Appendix A), verification test data (Appendix B), Cray performance data for the TASS code (Appendix C), general background on vector processing (Appendix D), and background on the TASS application (Appendix E).

Chapter 2

Background and Related Research

General purpose computers are designed to balance price versus performance over a range of applications [8], but higher performance requirements must be met with machines with architecture enhancements tailored to the desired application. The tailored computers tend to perform poorly on other applications [9]. Field-Programmable Gate Array (FPGA) technologies [5],[10] enable reconfigurable machines that can be tailored for many problems. In some cases, reconfiguration can occur as the machine is running. Thus, a machine architecture optimized for a particular problem can be designed, and as long as the overhead of configuring and using the hardware is less than running the same problem on a general purpose processor, performance gains will be realized. Examples cited in this chapter report orders of magnitude performance gains over general purpose processors.

Research concerned with the efficient use of configurable hardware to create custom accelerated applications spans nearly two decades. The applications are usually written in high-level software languages such as FORTRAN, C, and C++, and do not map directly or easily to a custom hardware implementation. Early examples of systems that provide this mapping are the configuration compiler for the PRISM system [9] and the NAPA-C language [11]. A recent example is the Riverside Optimizing Compiler for Configurable Circuits (ROCCC) [12].

Software-to-hardware compilation systems are also available commercially [13]. These systems, like the VectCore approach of this research, assume implementation on a hybrid general purpose, configurable architecture. These architectures feature a configurable fabric interfaced with a traditional processor, where the fabric generally acts as a co-processor or allows for expansions to the processor instruction set. The general purpose processor provides an interface to the application targeted for acceleration.

The viability of supercomputing on FPGA systems has been assessed [14]. The potential for performance gains has resulted in numerous studies ranging from implementations of basic building-block computations, such as matrix operations (discussed in Section 2.1) on research systems, to the porting of a full scientific application to a production supercomputing system with configurable hardware capabilities [15]. Hybrid general purpose, configurable High Performance Computer (HPC) systems have also been developed as commercial [16],[17] and research [18] systems. Several other examples of custom configurable implementations of scientific simulations are provided in Section 2.2.

Determining an appropriate architecture to use as a mapping target for an application requires a means to manage limits on the quantities of processing resources and design space exploration. The Dynamic Instruction Set Computer (DISC) [19] and the Piperench system [20] are early examples from the literature of the use of run-time reconfiguration as a means to address size limitations of the reconfigurable fabric. DISC uses a hardware instruction cache analogous to software memory caches. Piperench "virtualizes" the hardware using an architectural template composed of arrays of processing elements called stripes, and making a larger amount of "logical" stripes than physical stripes available for computation by time-sharing the actual available hardware. Recent efforts include an investigation of reconfigurable resource sharing on a HPC [21], and interleaving reconfiguration operations with processing operations to hide reconfiguration latency in [22].

The VectCore does not attempt to manage resources using run-time reconfiguration, but attempts to find the best configuration for a given problem and resource limit. The design space is limited to coarse-grained configurable architectures where the configurable parameters are quantities of processing components, interconnect components, and local storage. Related work in design space exploration and in tuning of coarse-grained configurable architectures for applications is described in Section 2.3.

Operations in an application must be scheduled once an architecture is defined. The VectCore approach specifically addresses scheduling under resource constraints [23], [24]. Research relevant to the VectCore scheduling algorithm is provided in Section 2.4.

Vector processing remains a relevant processing paradigm in the domain of scientific computation [25]. Several examples can be found in the literature of using configurable hardware to implement vector computations. For example, the Convey hybrid HPC [26] includes the "SPVector" single-precision vector processing functionality as an example of its application-specific "personalities". The Vector Instruction Set Architecture (ISA) Processors for Embedded Reconfigurable Systems (VIPERS) [27] is a "soft" vector processor approach that provides a configurable feature set that can be tailored to an application. The VectCore uses a parameterized vector processing template to allow tailoring of the implementation to the input problem. Additional relevant work on configurable vector co-processors is discussed in Section 2.5.

2.1 Reconfigurable Processors for Matrix Operations

Several examples of custom implementations for matrix operations, which are prevalent in science and engineering computations, are described in this section. An FPGA-based sparse matrix-vector multiplier is described in [28] motivated by the need to solve large systems of

sparse linear equations in Finite Element Method applications. The authors manage limited hardware resources by using a "striping" method to cover the elements of a sparse matrix input. The method is used to reduce the required number of processing elements in a parallel processing array.

Another linear parallel processing array architecture [29] is used for the LU decomposition method for solving a set of linear equations [30]. The authors develop a fine-grained pipelined implementation of the LU decomposition algorithm to expose parallelism that maps efficiently to a custom, scalable processing array. A blocking approach to the same problem is reported in [31], where the blocking partitions large input matrices into smaller data working sets to increase the utilization of limited local memory on the FPGA.

The matrix multiplication and the back-substitution step of LU decomposition problems are also used for experimental evaluation of the VectCore approach (see Chapter 5). The projects cited in this section are examples of focused design efforts that produce a highly specific hardware implementation of a particular problem.

2.2 Reconfigurable Scientific Applications

A computational fluid dynamics implementation using a systolic array of simple processing elements is described in [32]. A two-dimensional square driven cavity flow problem is used as a benchmark for performance tests. At a clock frequency of 60 MHz, the custom processor achieves a speedup factor of approximately seven times the speed of a general purpose processor running at 3.2 GHz.

A custom hardware simulator for galactic gas clouds is reported in [33]. The core computation is a method in which the cloud is modeled as a many-body particle problem, and the particles interact according to hydrodynamic governing equations. The authors describe their own

custom FPGA board design, design description language, and compilation tools to implement the simulation. The custom system performance is 40 times faster than a single general purpose processor.

The VectCore approach balances the performance of the custom designed implementations presented in this section with the flexibility of providing increased performance over a wider range of input problems suitable for vector implementation. A coarse-grained, parameterized processing target can be tailored to a particular input problem while allowing more design reuse than the result of a dedicated design effort.

2.3 Design Space Exploration

In [34], the authors describe several algorithms for determining allocations of functional units that meet performance and resource constraints for a given domain of applications. Two of the algorithms enable an automated examination of functional-unit quantities that maximize performance for a set of input applications while meeting a resource constraint. The authors use a cost and penalty function approach similar to this work. The performance and area predictions are only dependent on the number of functional units. There is no exploration of local storage and interconnect resources, or scheduling of the operations on an actual architectural instance.

Another design space exploration approach is described in [35], where a parameterized architecture model is used as a mapping target for a set of applications. The model contains reconfigurable processing elements, local storage, and interconnect parameters. The authors used the number of processing elements and number of possible contexts for their objective criteria. The context parameter supports run-time reconfiguration. For the approach in [35], simplifying assumptions such as unlimited register availability and interconnect re-

sources are made. The VectCore approach is tested in an actual hardware implementation, so all resources required for a functioning implementation are considered.

An approach that is demonstrated in a hardware implementation is presented in [36]. This work focuses on linear algebraic operations. Coarse-grained parameterized architectures are designed by hand specifically for each member of a set of input problems. With an architecture and parameter set, the range of each parameter is limited based on a set of chosen hardware constraints. The authors vary the parameters of the architecture instances to examine area versus performance tradeoffs. The VectCore approach does not rely on hand design or analysis, but rather assumes the vector processing model is appropriate for the input domain of interest. It then provides a tailored parameterized architecture to implement the input problems. An automated minimization scheme performs an area versus performance tradeoff for the tailoring.

In [37], a meta-level processor is defined in a proposed optimization framework called the YAWARA system. This processor obtains the execution profile for a given problem of a “base-level” (e.g. General Purpose Processor (GPP) processor) and determines an optimum configuration of the base-level processor. The target for this optimization is a custom fabric of uniform processing elements called thread engines. The YAWARA system is similar to the VectCore in its definition of a processing template and automatic “tuning” of the template for an application. The YAWARA system does not include vector processing capabilities.

The Unified Pattern Based Synthesis Kernel (UPak) [38] is a framework for identifying and selecting computation patterns specific to an application that can be implemented as custom instruction extensions on a configurable processor. The ASIP processor is the architectural model targeted in this work. The ASIP has configurable cells and registers connected to a GPP’s datapath through an interconnection structure. The number of registers and interconnection structure is tailored to an application, similar to elements of the VectCore

approach.

A layered approach for implementing linear algebra problems on distributed memory multiprocessor architectures is described in [39]. Components of the approach include a domain-specific Application Programming Interface (API) and runtime system. The runtime system is designed to target a multi-accelerator platform consisting of the GPP workstation connected to multiple hardware accelerators (e.g. GPP, Graphics Processing Unit (GPU), FPGA) [40]. The runtime system extracts parallelism at a high level from the input problem and schedules operations to the target. These operations can be calls to vendor libraries implementing Basic Linear Algebra Subprograms (BLAS) [41], thus enabling multiple levels of parallelism. The concept of a domain-specific layered approach is similar to the VectCore approach, as well as the chosen application domain. VectCore currently targets a single processor with general vector processing capabilities, using primarily a compilation-phase system for task scheduling and allocation.

2.4 Scheduling

Scheduling involves assigning a starting time to each member of a task set, typically in a manner that minimizes the total execution time for the set. Allocation, which assigns resources to tasks, is closely related. Scheduling algorithms can be classified as transformational or iterative/constructive [24]. Transformational algorithms start with an extreme schedule, such as completely serial operations, and apply transformations to reduce the overall schedule length. Iterative/constructive algorithms build a schedule one task at a time until all operations in the task set are scheduled. Several examples of these two algorithm classes are cited in [24].

List scheduling is a type of iterative/constructive algorithm, and is the approach chosen for

the VectCore (design decisions are discussed in Chapter 4). An overview of the algorithm is shown in Figure 2.1. For each schedule time index, the operations whose data dependencies are satisfied form a ready set. Operations are scheduled from the ready set while remaining within resource constraints. List scheduling algorithms differ in how the next operation is selected. The selection criteria, or heuristics, constitute a priority function.

An example heuristic is As Soon As Possible (ASAP) scheduling, also referred to as demand list scheduling [42]. For this approach, operations from the ready set are scheduled as soon as resources are ready to support them. ASAP is therefore a local (greedy) heuristic, because resources are applied to all the ready operations until exhausted. No consideration is given to other operations as each ready set member is scheduled. Conversely, in As Late As Possible (ALAP) scheduling, operations are deferred to the latest time index that will still allow data dependencies to be satisfied.

Kohler [42] shows ASAP scheduling will not consistently produce a minimum length schedule, and describes a more global heuristic of critical path length. This heuristic can be implemented by computing the length from the operation to a terminating node of the data flow graph. For example, a “STORE” operation. The ready list is sorted by decreasing path length. Experimental data presented in [42] shows the performance of this heuristic to be near-optimal in most cases.

The critical path heuristic is closely related to the freedom of an operation. The freedom is the number of time indices an operation can be assigned to while satisfying all data dependencies. This heuristic was used in the SLICER state synthesizer [43] and in the MAHA system [44]. Ties in ready set priority must be broken when path lengths are the same, or the worst schedule could result [45]. The Critical Path, Most Immediate Successors First algorithm [45] asserts the number of immediate successors as a effective heuristic for

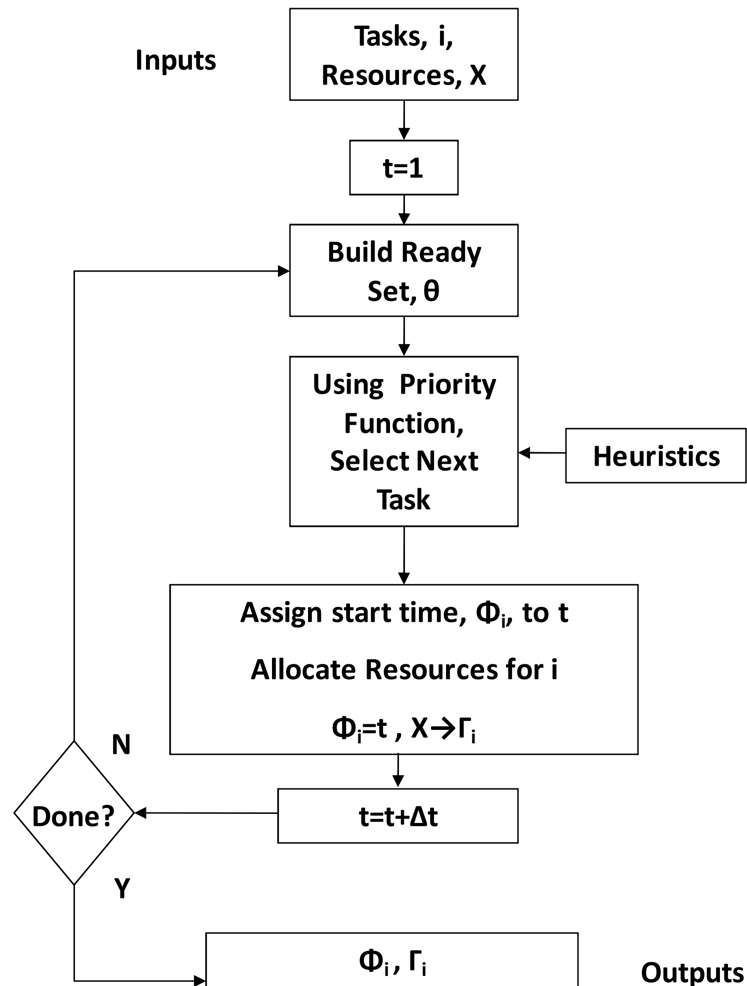

Figure 2.1: Overview of the list scheduling algorithm. For each time index, a set of all tasks ready for execution is formed. Using a heuristics-based priority function, the next task is selected. The current time step is assigned to the selected task, and resources are allocated for its execution. The time index is advanced, and the algorithm repeats until all tasks are scheduled.

breaking ties in the path length. The VectCore uses the path length and number of successors heuristics.

The VectCore scheduling heuristics are also designed with consideration of those used in Cray compiler systems. Lee [46] describes the scheduler used in the 1990 version of the Cray FORTRAN compiler (cft77). This list scheduling algorithm prioritizes the ready set by decreasing path length and breaking ties with the number of descendants as in the SLICER system. In a second scheduling pass, vector register loads are re-scheduled ALAP to minimize live register requirements.

2.5 Configurable Vector Co-Processors

Similar to the VectCore, the Vector-Extended Soft Processor Architecture (VESPA) [47] features a vector co-processor with a configurable architecture that can be tailored to input problem requirements. The VESPA configurable parameters are the number of vector lanes, lane width, maximum vector length, memory crossbar lanes, and selective enabling of vector instructions. Multiple vector lanes allow faster processing of a single vector instruction. The vector is partitioned into equal parts and processed in parallel, where the number of parts is equal to the number of vector pipeline lanes. Thus, the amount of parallelism the VESPA approach enables is dependent on the vector length of the input problem and the maximum vector length of the architecture. The VectCore architecture (Chapter 4.4) allows parallelism at the vector instruction level that is not dependent on the vector length. The VESPA architecture parameters are hand-tailored with the goal of reducing the required amount of FPGA resources. An automated performance optimization is not performed.

The AutoTIE [48] system performs a tradeoff of cost versus performance to determine a set of ISA enhancements to the basic ISA of Tensilica's Xtensa configurable processor. The

enhancements improve the processor's performance for a particular application and remain within hardware constraints. Specifically, Very Long Instruction Word (VLIW), vector, and fused (e.g. $SAXPY$) operations comprise the solution space for the enhancements. The AutoTIE system does not support chained vector operations. The cost/performance tradeoff is executed in AutoTIE's configuration generation flow. For each loop in the input code, and over a range of vector lengths, a basic configuration to execute the loop is determined. The basic configurations are expanded by adding critical architecture resources until a physical limit is reached. An exhaustive enumeration of resource and operation slot assignments in the instruction issue logic is performed to determine assignments resulting in the minimum execution time. This process does not consider data dependencies in the input problem. The VectCore approach does not utilize complex instruction issue logic. Further, data dependencies are considered because an actual schedule and allocation is determined for each architecture specification.

Chapter 3

Problem Formulation

In this chapter, the thesis problem of determining a set of architecture parameters that minimize the execution time for a set of vector operations, subject to hardware constraints, is formulated as a multivariable constrained minimization. The complexity of this problem is assessed with a comparison to the general scheduling problem[49].

3.1 Problem Definition

Let X represent a space of possible parameter specifications for a parameterized processing architecture. X is an integer vector of length r, where $X(j)$ $(1 \leq j \leq r)$ is the quantity of architecture resource type j, and the first k $(k \leq r)$ members of X represent the number of architecture nodes capable of processing tasks (processors), while the remaining $(r - k)$ members represent the number of supporting architecture resources such as interconnect buses and registers that do not directly process tasks. The elements of X are constrained to be a non-negative numbers. An example instance of X is shown in Figure 3.1.

Additional variables describing the target computation are as follows. Let G be a task graph with n tasks represented as vertices, and an edge G_{ij} denotes task j occurs only after task i completes. An example task graph is shown in Figure 3.2. Let the vector τ_i of length n

X(2, 1, 3, 4)

r=4, k=2

X(1) = number of adders, A
X(2) = number of multipliers, M
X(3) = number of interconnect busses, B
X(4) = number of registers, V

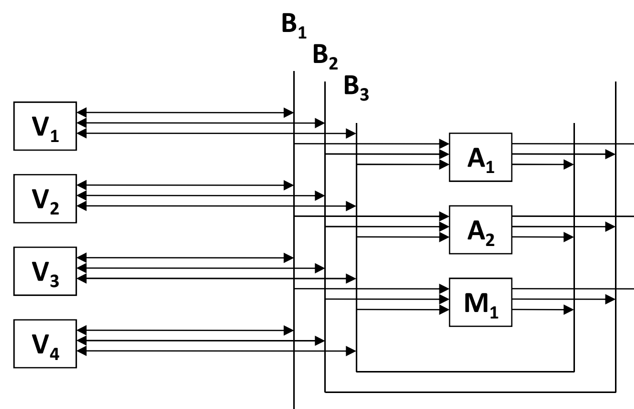

Figure 3.1: Example of architecture specification vector.

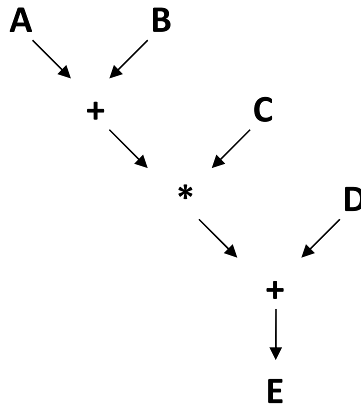

Figure 3.2: Example of task graph for input computation $E = C(A + B) + D$. The vertices represent operations or tasks, and edges denote data dependencies.

indicate the time required for task i to execute. Define the positive vector ϕ_i to indicate the starting time for task i. Assume the processing elements of X perform unique operations only suitable for specific tasks, and let P be defined as an integer-valued vector of length n where $1 \leq P(i) \leq k$ indicates the index of processor resource task i must use[1]. Also define R as an n by $(r - k)$ integer-valued matrix, where R_{ij} indicates the amount of resource type $(j + k)$ that is consumed by task i.

The minimization problem for the objective function $F(X)$ is then stated as follows. Find $X = X^*$ that satisfies

$$\min_{X \geq 0}(F(X)) = \min_{\phi \geq 0}(f_X(\phi)) = \min \left[\max_{1 \leq i \leq n} (\phi_i + \tau_i) \right] \tag{3.1}$$

subject to

$$g(X) \leq C \tag{3.2}$$

where

$$1 \leq j \leq k, \quad |\{\{i : \phi_i \leq t \leq \phi_i + \tau_i\} : P_i = j\}| \leq X(j), \tag{3.3}$$

[1]In an actual processing architecture, tasks will be computational operations, such as additions and multiplications. Although processing elements could be designed to perform multiple operations, it is assumed that an implementation uses elements specifically designed for one operation, as may be desired to conserve hardware resources. This assumption is conservative for this complexity analysis.

$$(k+1) \leq j \leq r, \sum_{\{i:\phi_i \leq t \leq \phi_i+\tau_i\}} R_{ij} \leq X(j), \tag{3.4}$$

and

$$G_{ij} \in G \Longrightarrow \phi_i + \tau_i \leq \phi_j \tag{3.5}$$

In Equation 3.2, g represents the area cost of resources in X and C is the maximum available area. The details of g are provided in Chapter 4. Equation 3.1 specifies a minimization problem within another minimization problem:

1. Given a particular architecture X^i, what ϕ minimizes the deadline D such that $f_{X^i}(\phi) \leq D$?

2. Given the objective function F, what specific architecture X^*, minimizes $F(X)$ while satisfying resource constraints g?

The approach used to solve these problems is discussed in Chapter 4. The next section provides an analysis of the complexity of the problems.

3.2 Complexity Analysis

Solving Equation 3.1 is at least as difficult as solving the nested problem of finding the ϕ that minimizes D. A formulation for the general scheduling problem is described using a multiprocessing model in [49], and restated here for the purpose of comparison to the thesis problem[2]. As defined in [49], the model consists of three distinct finite sets: processors, resources, and tasks. There is only one type of processor, and each processor is capable of executing one task of finite duration τ at a time. The members of the set of tasks T are subject to a partial order $\prec$ such that if $T_i \prec T_j$, T_j can only begin execution after the

[2]This model may not be a sufficiently complete representation for real systems, but is sufficient to identify a lower bound on the complexity of the thesis problem.

execution of T_i is complete. For each resource R_j and task T_i, there is a resource requirement $R_j(T_i) \leq Z_j$ representing the amount of resource R_j required by T_i during the task execution time τ_i, where Z_j is a bound on the available amount of resource R_j at any particular time. A task T_i requires one processor and may require more than one resource type to execute.

The input for the general scheduling problem using this model is a number m of processors, a set $R = R_1, R_2, ..., R_r$ of resources with an associated bound Z_j for each R_j, a set $T = T_1, T_2, ..., T_n$ of partially ordered tasks with resource requirements $R_1(T_i)...R_r(T_i)$ for task T_i, and finally a deadline D.

In [49], a valid schedule for an input to the general scheduling problem is defined as a function $\sigma : T \rightarrow \{0, 1, 2, ..., D-1\}$ satisfying the following four conditions:

1. for each $T_i \in T, \sigma(T_i) + \tau_i \leq D$,
2. for each $T_i, T_j \in T$, if $T_i \prec T_j$, then $\sigma(T_i) + \tau_i \leq \sigma(T_j)$,
3. for each integer $t, 0 \leq t < D$ the set $E_\sigma(t) = \{T_i \in T : \sigma(T_i) \leq t < \sigma(T_i) + \tau_i\}$ satisfies $|E_\sigma(t)| \leq m$ where $E_\sigma(t)$ is the the set of tasks being executed at time t under schedule σ, and
4. for each integer $t, 0 \leq t < D$, and each $j, 1 \leq j \leq r, \sum_{T_i \in E_\sigma(t)} R_j(T_i) \leq Z_j$.

Comparing the thesis formulation to that of the general scheduling problem shows the latter to be a special case of the thesis problem, where $k = 1$. The thesis problem allows for a heterogenous set of processing elements, $X(1), X(2), \ldots, X(k)$ and a constraint P on which element can be allocated for a particular task. The general scheduling problem is defined with only one processor type of quantity m for a given input. The remaining elements of the general scheduling problem map directly to the elements of the thesis problem. For example, the members of $X : X(k+1), X(k+2), \ldots, X(r)$ map to the resource bounds

$Z = Z_1, Z_2, ..., Z_r$ in the general scheduling problem, and the partially ordered task set T can also be represented as a graph G. Therefore the general scheduling problem is a more limited instance of the thesis problem, and solving the thesis problem is at least as difficult as solving the general scheduling problem.

Verifying that any guess for an n-task schedule meets the deadline D clearly is a decision problem that can be verified in polynomial time, classifying the thesis problem as class NP [6]. In [49], the authors show the general scheduling subproblem specified as $m \geq 2$, $r \geq 1$, and the tasks of T subject to the partial order $\prec$ of a "forest" is NP-complete. The forest partial order is defined in [49] as given the order $T_i \prec T_k$ and $T_j \prec T_k$, either $T_i \prec T_j$ or $T_j \prec T_i$. An equivalent instance of the thesis problem could easily be encountered, implying that instances of the thesis problem are also NP-complete.

Chapter 4

Approach

This chapter describes the algorithmic approach designed to find solutions to the problem formulated in Chapter 3. Details of the parameterized vector processing template, the vector scheduling and allocation algorithm, and the VectCore co-processor design are presented. Characteristics of these components are described that enable efficient and scalable implementations. These implementations use techniques for high performance vector processing while leveraging the added flexibility of a reconfigurable implementation.

4.1 Overview

The motivation of this research, as stated in Chapter 1, is to investigate alternatives to re-hosting large, high-performance applications on a new computer system. This research focuses on the use of custom vector processing resources to achieve a performance gain as compared to a single scalar processor. A legacy code application tailored for execution on a vector computer is the assumed target for acceleration. Custom configurable processing resources enable a wider range of vector processing architectures than found in traditional vector computers. Additionally, configurable computing resources allow a vector architecture to be tailored to a specific application, instead of the traditional approach of writing the

application to be tailored to the computer. The problem of finding the best tailoring of flexible but limited resources to a problem is complex, as discussed in Chapter 3, and is an underlying problem in the field of configurable computing. This work limits the scope of the solution space to vector processing, a mature computation paradigm. The paradigm offers proven performance enhancing techniques and lessons learned used in this research approach.

Scientific codes are often large applications, and hardware resource limitations constrain the number of code segments that can be accelerated. A hybrid microprocessor/FPGA system on which code segments not implemented in hardware execute on the microprocessor, and accelerated code segments run on the FPGA configured as a co-processor, is a suitable implementation for this problem. Alternatively, an FPGA could be configured for direct implementation of a segment of code, but there could be many of these candidates identified for acceleration. A custom implementation of each would not generally be practical given limited resources. A more coarse-grained co-processor approach is a way to remain within resource constraints. Custom configurable hardware allows for a tailoring of a co-processor to its input, providing a means to balance the performance of a highly specific implementation with the flexibility of architecture reuse that manages resource usage and reduces the time to a solution.

The overall system assumed by the solution approach is shown in Figure 4.1. Software application code targeted for acceleration runs on a hybrid system consisting of an FPGA tightly-coupled with a general-purpose processor. Computations in sections of the application code identified as candidates for FPGA implementation are input to tools that determine an architecture tailored to each input problem. A mapping of each input computation to its tailored architecture is also produced. Custom tools produce the High-Level Design Language (HDL) representation of the tailored architecture, and a microcode program to run

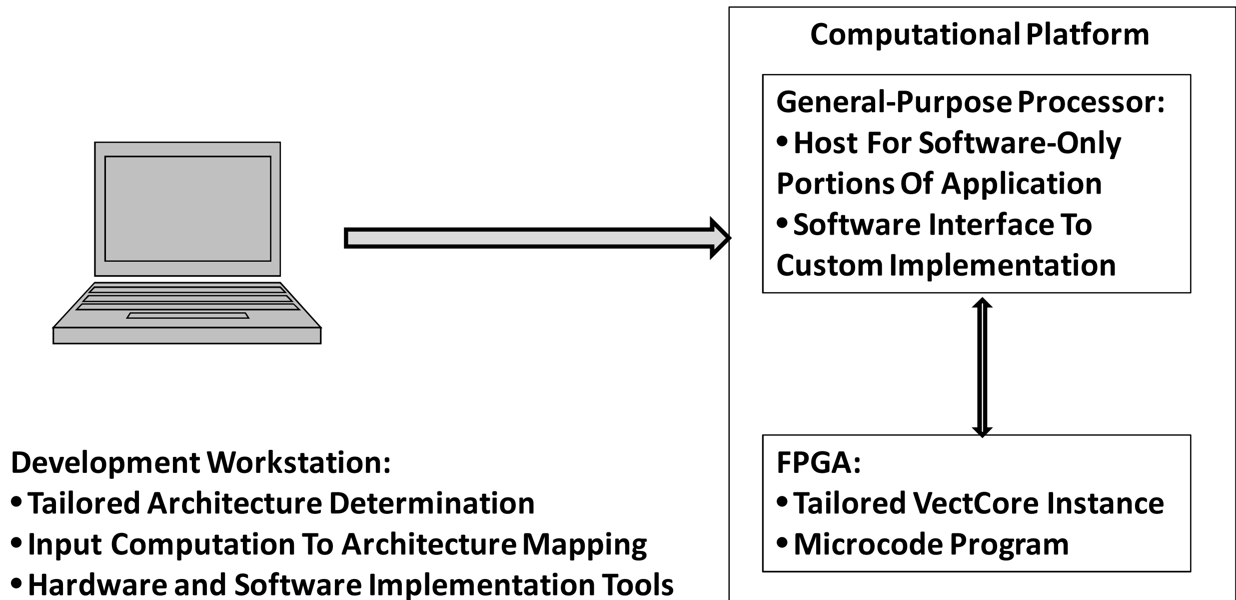

Figure 4.1: Solution system components and functions. Tools on a development workstation generate a tailored architecture instance, and a mapping to the architecture, for sections of a software application targeted for acceleration. Implementation tools produce a file to configure the FPGA for an architecture instance. A general-purpose processor runs the original software portions of the application, and provides an interface for microcode programs that run the accelerated application sections on the FPGA.

on the architecture. Vendor tools produce the FPGA configuration file. All the custom and vendor tools run on a development workstation. A software program consisting of the original application augmented with code that provides an interface between the software application and the FPGA runs on the general-purpose processor.

The algorithmic approach consists of the following development process and associated components, referring to Figure 4.2.

- The approach inputs (labeled (1) in Figure 4.2) are a representation, G, of a set of computations targeted for acceleration, and an overall resource constraint, C, for the FPGA. Section 4.2 describes the implementation details for these inputs.
- An integer minimization algorithm (labeled (2) in Figure 4.2) is executed for the ob-

jective function F. Implementation details for the algorithm are in Section 4.3. The minimization algorithm consists of the following subcomponents.

- An architectural framework (labeled (3) in Figure 4.2) is the target for mapping the input computation to processors and supporting resources. The template is parameterized allowing different specifications for the quantity and type of processing resources. A vector X specifies the parameters for a given architecture. Section 4.4 describes details of the chosen implementation for the architectural template.
- A scheduling and allocation algorithm (labeled (4) in Figure 4.2) maps operations in G to a set of starting times, ϕ, and resources in a particular instance, X^i of the architectural template. Section 4.5 provides details of the scheduling and allocation algorithm.
- A penalty function, p, is used for minimizing the objective subject to the constraint limit, C. Section 4.3 describes the implementation of the penalty function.

- The outputs (labeled (5) in Figure 4.2) of the minimization algorithm are the architecture specification, X^*, that minimizes the scheduled execution time of G without exceeding the resource constraints, and the associated schedule, ϕ.

- The VectCore co-processor design (labeled (6) in Figure 4.2) integrates the tailored instance of the architectural template with the overall system. The VectCore provides an interface to the general-purpose processor and controls the execution of the resources specified by X^*. The VectCore design is described in detail in Section 4.6.

- An integrated hardware implementation (labeled (7) in Figure 4.2) completes the solution approach. The implementation consists of the tailored VectCore instance and a

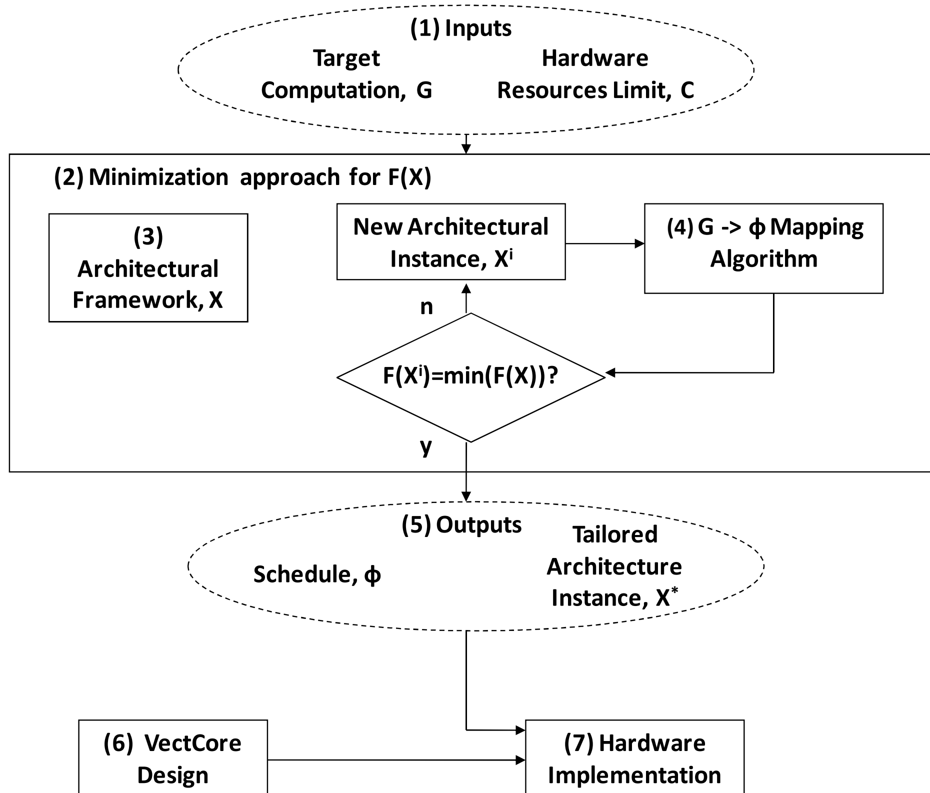

Figure 4.2: Algorithmic approach overview. A minimization algorithm iterates on specifications for architecture instances to minimize the objective. Each architecture instance, X^i is the target for a mapping algorithm that determines a schedule for the operations in the input computation, G, on X^i. The approach outputs are the architecture specification, X^* that minimizes the objective, and the schedule for executing the computation G on the architecture specified by X^*. Implementation tools use these outputs to produce a VectCore co-processor instance that runs the input problem.

microcode program to run the input computation on the VectCore. Hardware implementation details chosen for this research are described in Section 5.1.

4.2 Approach Inputs

The research approach assumes the input computation targeted for acceleration is a high-level legacy code segment optimized for vector implementation. The implementation choice for the

input problem is a custom VectCore vector assembly code developed for this research. The assembly code approach provides a traditional interface for a vector compiler. A description of the VectCore assembly code syntax is provided in Appendix A.2. This code is assembled to a data flow graph representation for input to the minimization algorithm.

The second input is the resource limit, C, used in the constraints for the minimization algorithm. This limit represents the available physical resources of an FPGA, and are part-specific. The implementation details of the constraint function are discussed in Section 4.4.3.

4.3 Minimization Algorithm

The problem formulation discussed in Chapter 3 is suitable for a multi-variable integer optimization approach to determine the solution that minimizes the objective function F. An adaptive simulated annealing algorithm [50] is used for the minimization, but other methods appropriate to minimization with discrete input variables could be used. The algorithm has many configuration options but only the default configuration is used.

A penalty function p, is used in the implementation of the minimization problem to incorporate the constraint $g(X)$ into the objective function. Many combinations of values for the members of X can exceed C, so the constraints in the problem cannot be expressed as simple ranges for members of X. Also, the mapping function for tasks to starting times is non-linear and does not allow for an analytic approach such as using Lagrange multipliers [51].

The penalty function is designed to add a large value to the objective function for input combinations that exceed C, and to have as little influence as possible ($p(g(X)) \Rightarrow 0$) for all X that satisfy the constraint. The second characteristic prevents "rewarding" the objective function for satisfying the constraint, because this influences the optimization with input combinations that may have little or no connection to the combinations that minimize the

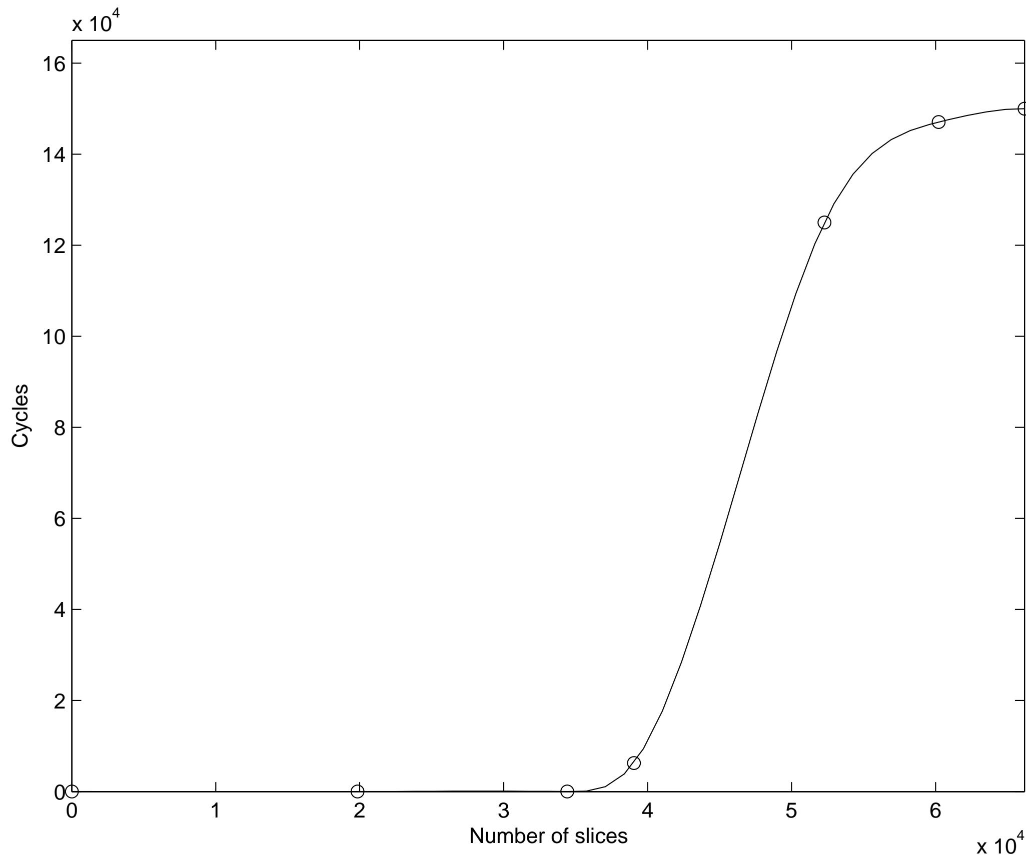

Figure 4.3: Example penalty function, p(X).

objective F. An additional requirement on p is that it is differentiable to facilitate its use in optimization algorithms [51].

The penalty function p is empirically designed using the MATLAB® "spline" function. The function is close to zero until approximately $g(X^*) = 0.6C$, and then rapidly increases up to a problem-dependent penalty for larger $g(X^*)$. An example of $p(g(X^*))$ is shown in Figure 4.3. In the figure, $g(X^*)$ computes a resource usage estimate using data collected from synthesis tools. The process for collecting this data is described in Section 4.4.3.

4.4 Architectural Template

The architectural template provides a design framework for a configurable vector processing core. As defined in Chapter 3, the vector X specifies the parameter set that defines an instance of the template. The members of X are quantities of processing components, and the following notation is used for the components included in the VectCore implementation designed for this research.

- L = vector load/store units
- V = vector registers
- A = vector adder units
- M = vector multiplier units
- B = functional unit buses
- Y = vector $SAXPY$ units
- I = vector inner product units

Other floating-point functional units could be defined for an implementation of the approach. Figure 4.4 is a conceptual diagram of the template. Many other templates could be used. It is shown in the following sections that the interconnect and memory bandwidth performance of the VectCore template scales well compared to alternatives. In addition, this section and Section 4.5.2 describe the flexibility of the template to support performance enhancing vector computing techniques. Figure 4.5 shows a specific example of a template instance for the specification $4L$, $5V$, $1A$, $1M$, $3B$, $1Y$, $0I$.

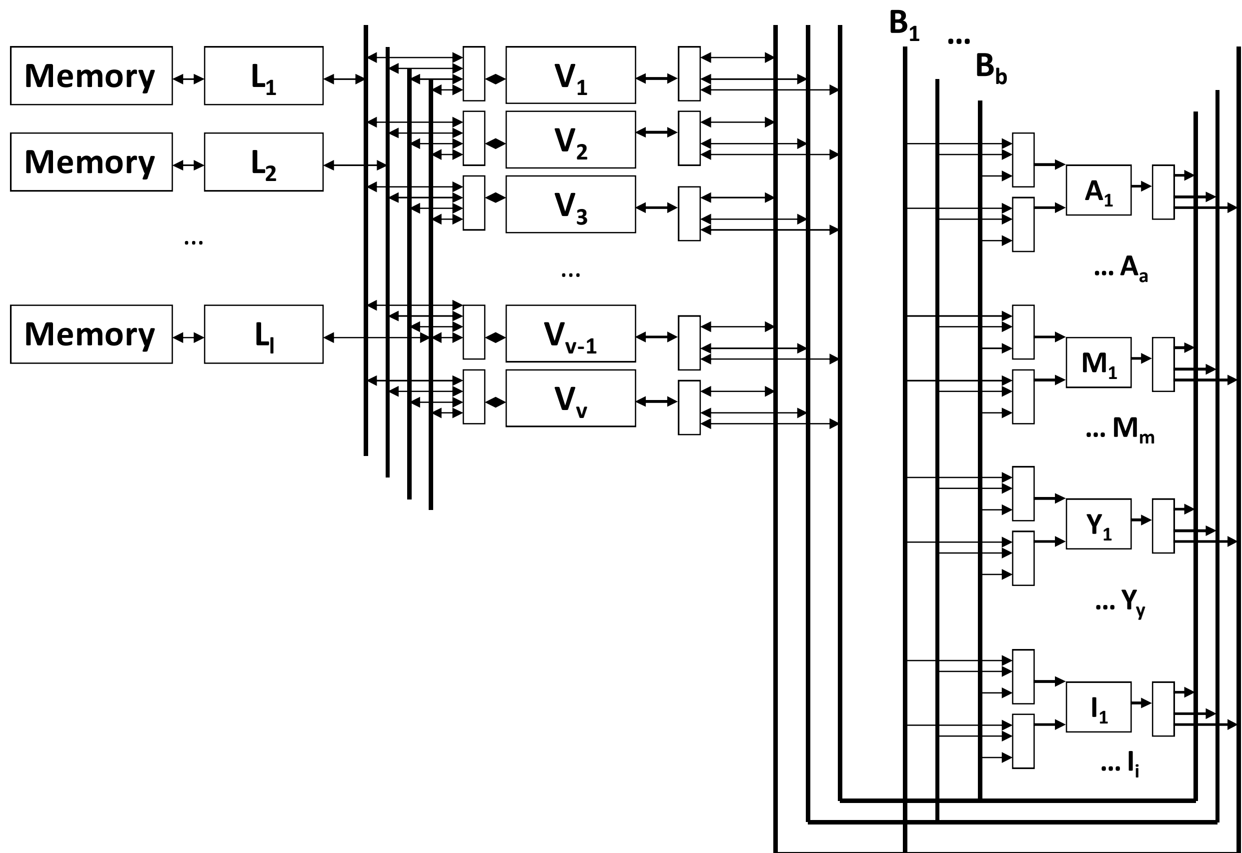

Figure 4.4: Parameterized architecture template for VectCore custom vector processing core. The memory modules are external to the FPGA.

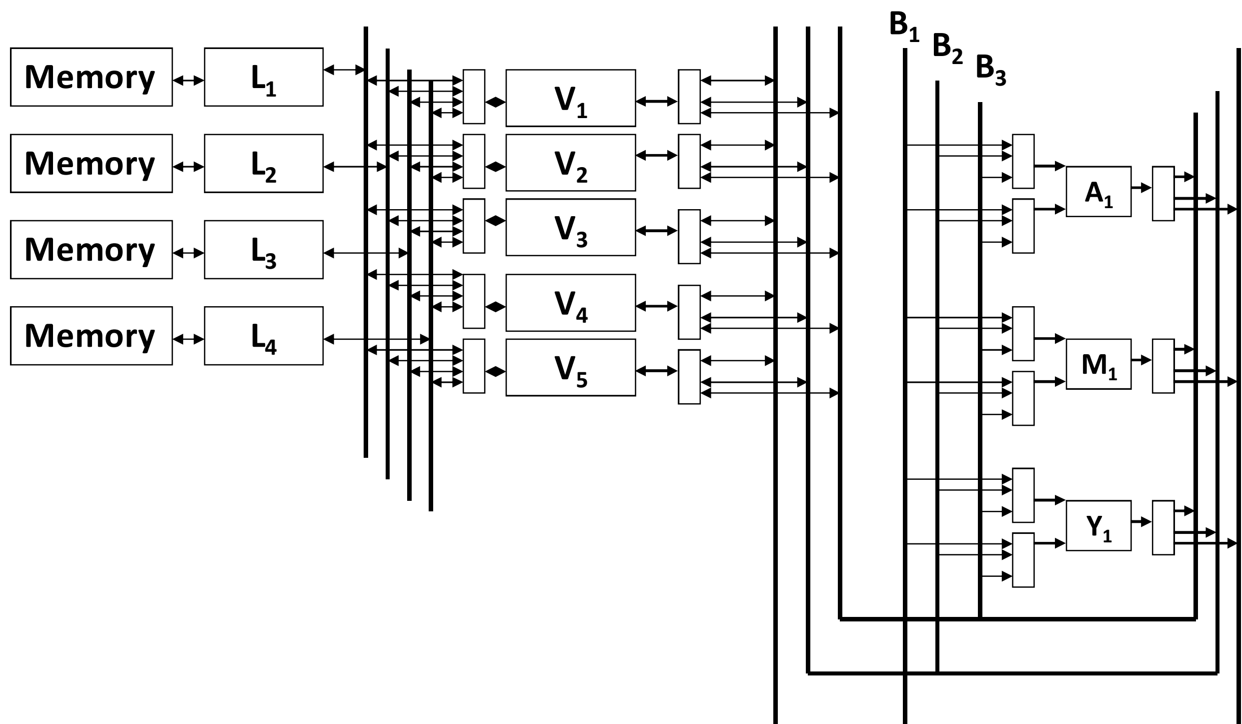

Figure 4.5: Example VectCore architecture template. Specification includes $4L$, $5V$, $1A$, $1M$, $3B$, $1Y$, and $0I$.

Table 4.1: VectCore vector unit number of pipeline stages.

Operation	**Pipeline Stages**
A	18
M	18
L	8

For the VectCore implementation tested in this work, the memory modules shown in Figure 4.4 are external to the FPGA, where the remaining template components are implemented. In general, the memory could be internal or external to the FPGA. The vector load/store units provide a pipelined, dedicated interface to each of these memory modules. Functional unit bus interconnect resources, B, can be configured to handle functional unit operands and results.

The functional unit implementations follow the approach in [52] for a similar format to the IEEE 754 single-precision standard. For the purposes of this research, not all the requirements of the standard are implemented[1].

Additional pipeline stages are added to the designs in [52] as required to operate at the frequency chosen for the VectCore implementation tested in this research. Table 4.1 summarizes the pipeline latencies for the VectCore operations implemented in hardware.

The architectural template supports efficient flexible vector chaining. The concept of vector chaining is introduced in Section D.1.4. Chaining enhances performance by connecting the output of one vector operation to the input of another. The combined vector pipelines act as one vector operation of the same length as either single operation, with the pipeline latency of the two chained operations added together. Depending on the implementation, vector

[1]Rounding, exception handling, special values such as NaN and inf, and denormal numbers are not implemented. The effect of this simplification is to reduce the hardware resources required for the functional units. The reduced hardware requirement could cause estimations of the performance of the VectCore under a particular architecture limit to be higher than those for an IEEE 754 fully-compliant VectCore implementation, due to the lower per-unit resource cost potentially allowing more units in the specification. These impacts are assumed small and not considered in this research.

chains can be formed with or without a storage element between the operations. A storage element enables a flexible chain, in which the second operation can begin later than the data ready time of the first operation. A direct chain with no storage requires the second operation to start immediately when data is ready from the first operation.

The VectCore template assumes a dual-port on-chip memory for vector register implementation. This allows flexible vector chaining between loads/stores and floating-point operations. Floating-point vector operations are chained by connecting the functional units directly, so an interconnect bus is shared between the output and input of two chained operations. Sharing an interconnect bus and not using a register between the functional unit chains saves these resources at the cost of missing some flexible chaining opportunities. To increase potential chaining opportunities, a selectable output delay is included in each functional unit pipeline to adjust its data ready time (see Section 4.5.2).

Figure 4.6 shows an example of chaining on the VectCore template for a VectCore instance specification of $4L$, $5V$, $1A$, $1M$, $5B$, $0Y$, and $0I$. The operation is $D = (A + B)C$, where A, B, C and D are vectors of length v_l. In the figure, flexible chains occur between the load operations for operands A and B, and the addition $A + B$. A direct chain is formed between the multiply and add operations. Flexible chains through register resources are formed between the load of operand C and the multiplication, and between the multiplication and the store of the result D. The entire set of operations can therefore be completed in v_l clock cycles plus the combined pipeline latency of a load, add, multiply, and store operation.

4.4.1 Interconnect Topology Analysis

The design of the interconnect architecture between the vector registers and functional units is compared to three other commonly used interconnect topologies in this section. The

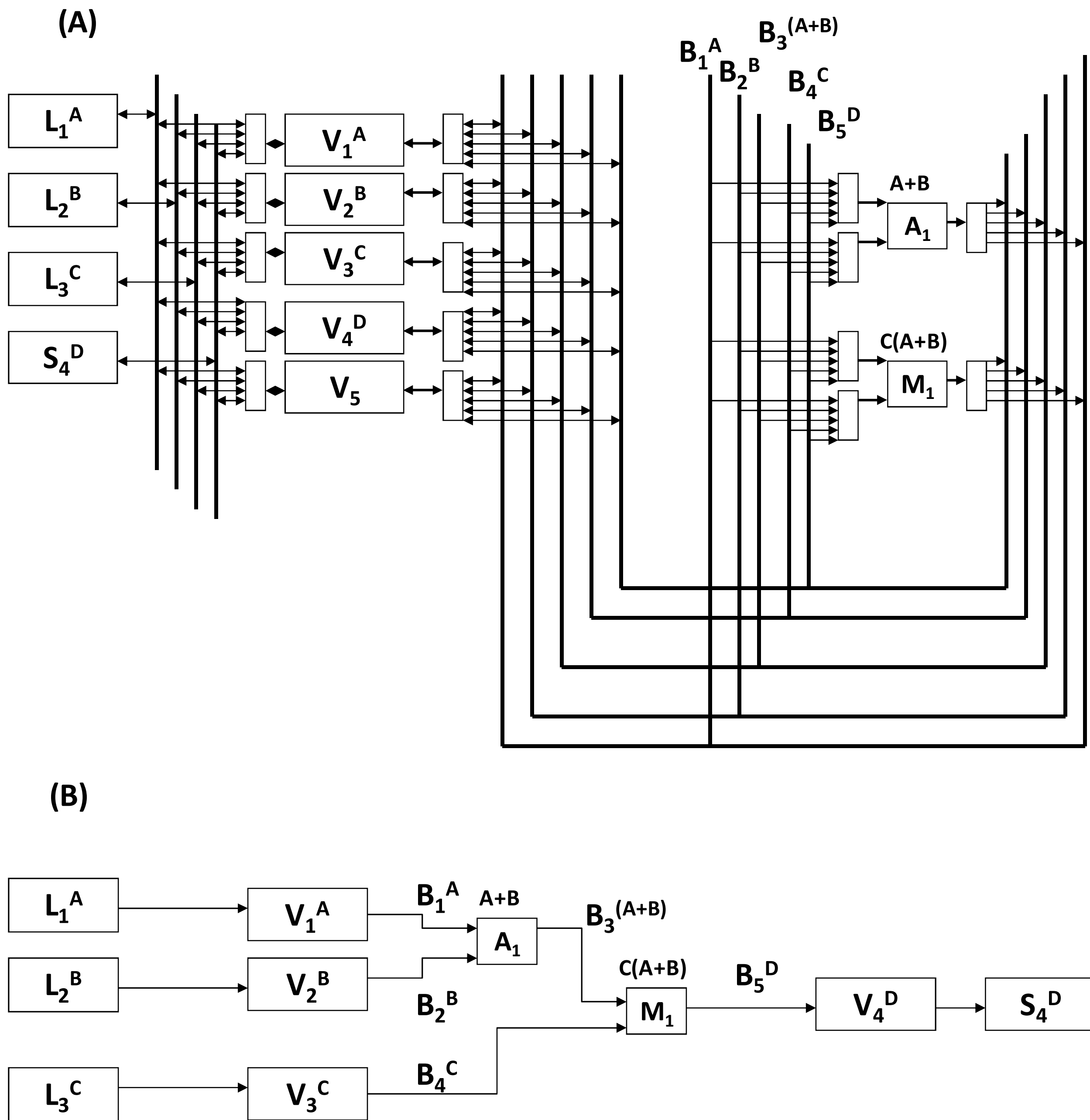

Figure 4.6: Example of chaining on VectCore architecture template for operation $D = (A + B)C$. Allocation of template resources is shown in part (A), where superscripts on the resource labels indicate data allocation. Functional unit allocation is indicated by labels over the functional units. Part (B) shows the same allocated resources in a data flow representation of the chained operation.

approach used for this research is shown to be a practical choice with advantages over these other topologies.

Typical characteristics [53] used to compare interconnect topologies are shown as the column headings in Table 4.2. The first three characteristics relate to the performance of the network, and the remaining characteristic relates to the implementation cost. The VectCore interconnect topology is compared to a bus, crossbar, and Omega network [53]. In Table 4.2, the number of inputs = the number of outputs = q for each network topology. The data width of each connection, or link in the network, is denoted as w.

Serializers/Deserializers (SerDes) communication links are a means of reducing wiring complexity by reducing a bus to a fast serial link with a transceiver pair [54],[55]. Multi-gigabit transceivers available in contemporary FPGAs can serialize a 32-bit bus at rates comparable with typical maximum FPGA design clock rates (in the hundreds of MHz) [56]. For the comparison in Table 4.2, the wiring complexity of a topology employing SerDes links is reduced by a factor of the data width, and shown after each complexity for a parallel interconnect implementation, in parenthesis.

In the VectCore topology, U is the total number of vector functional units and B denotes the number of independent links between input/output pairs in the network. For the VectCore network, $q = V + U$. There are B q-input multiplexers that select a functional unit or register as input for each functional unit bus. There are also V B-input multiplexers that select a functional unit bus as input for each register. Finally, there are approximately[2] $2U$ B-input multiplexers that select a functional unit bus as input for each of the two inputs of each functional unit. Therefore, the number of connections required for the VectCore network is $B[q + (V + 2U)]$. The wiring complexity for the VectCore can be considered proportional to

[2]Functional units with more than two inputs, such as a $SAXPY$, do not significantly affect this order of magnitude complexity analysis and are therefore omitted.

Table 4.2: Comparison of interconnect topologies.

Topology	**Min. Latency per Datum**	**Maximum Bandwidth**	**Connectivity**	**Wiring Complexity**
Bus	Constant	$O(w)$	$q!$ permutations of one input to one output, one at a time	$O(w)$, $(O(1))$
Crossbar	Constant	$O(qw)$	$q!$ permutations of one input to one output, q at a time	$O(q^2w)$, $(O(q^2))$
Omega	$O(log_2q)$	$O(qw)$	$q^{\frac{q}{2}}$ permutations of one input to one output without blocking, q at a time	$O(((\frac{q}{2})log_2q)w)$, $(O(((\frac{q}{2})log_2q)))$
VectCore	Constant	$O(Bw)$	$q!$ permutations of one input to one output, B at a time	$O(Bqw)$, $(O(Bq))$

Bq, or $O(Bqw)$.

Comparing the topologies summarized in Table 4.2 to the VectCore interconnect, when $B = 1$, the VectCore has the same performance and cost characteristics of a bus. Namely, one link is shared between all input/output pairs in the network, and only one input/output pair may be connected at a time. When $B = q$, the VectCore maximum bandwidth is proportional to qw, the same as the crossbar and Omega network. The corresponding VectCore cost in terms of wiring complexity is $O(q^2w)$. Therefore, the VectCore interconnect network performance and cost can be approximately matched to topologies ranging from a bus to a crossbar depending on the selection of the parameter B. This parameter tailoring allows partial customization of the VectCore interconnect network to a particular application.

The Omega can only guarantee $q^{\frac{q}{2}}$ permutations of input to output connections can be realized at a particular time without blocking, and up to $log_2 q$ passes could be required to accomplish blocked permutations [53]. Latencies associated with blocking networks potentially reduce opportunities for chaining. Chaining requires specific timing relationships between the data producers and consumers. Long vectors increase the data occupancy during a particular network transaction, potentially increasing blocking latencies and reducing the number of chaining opportunities. VectCore configurations where $B < q$ produce blocking networks as well, but the VectCore approach is designed to automatically perform a performance versus cost tradeoff when determining a configuration. The effects of network blocking on chaining, and ultimately schedule length, are reflected in each potential configuration evaluated by the minimization algorithm.

4.4.2 Memory Interface Analysis

The memory interface for the VectCore leverages the approach used in vector supercomputers to have multiple independent memory banks [57]. Each off-chip memory component is

associated with a particular vector load/store unit in the current implementation. The number of these units is a parameter of the architecture. This direct memory interface reduces the bandwidth requirements on the general purpose processor to core interface by keeping the data bandwidth independent from the instruction bandwidth. The bandwidth W scales linearly with the number of load/store units, L, as $W = Lw\omega$, where ω is the memory operating frequency.

4.4.3 Resource Usage Prediction

As described in Section 4.1, the penalty function used for the minimization problem requires an estimate of the physical resources used by an architecture specification. The constraint function introduced in Equation 3.2 computes the resource usage for an architecture specification X. The constraint function design is provided in this section, and its accuracy is evaluated using empirical measurements from synthesis tools.

The main resources of interest for the Xilinx®Virtex II pro™FPGA used for evaluating this research are slices, Block Random Access Memory (BRAM), and 18X18 multipliers [58]. The slice resource is limiting in the VectCore approach, and is considered to be an adequate measure of general resource usage. Devices from other vendors use different resource units, and these units could be used in the VectCore approach as easily as Xilinx®slices.

Each member of X has an associated physical resource cost per unit for resource type $X(j)$, defined as c_j. Per-unit slice costs for the members of X are derived from place and route statistics on VectCore instances and summarized in Table 4.3. The values are determined by varying each architecture parameter, running a VectCore synthesis, and obtaining the slice difference for each parameter change from the place and route output[3]. The parameter

[3]Slice usage for the Y and I operations are estimated by adding the slices for an adder and a multiplier. The adder and multiplier are the main components of a $SAXPY$ or inner product operation. These operations are evaluated in the scheduler, but not implemented in the current VectCore hardware.

Table 4.3: VectCore per-unit resource cost in slice units.

c_j	**Slices**
L	401
V	323
A	956
M	1133
B	442
Y	2531
I	2531
C_0	6553

values used to produce the data in Table 4.3 are small, so the data in the table represents basic slice resource costs for each type of resource. At higher parameter values, the wiring complexity described in Section 4.4.1 has a significant affect on the slice usage.

The slice usage attributed to the wiring complexity is modeled as $B[qC_q + (V + 2U)C_B]$, where C_q is the resource cost per input for a VectCore q-input multiplexer, and C_B is the resource cost per input for a VectCore B-input multiplexer. Adding the slice usage due to wiring complexity to the per-unit resource and basic configuration slice usage terms yields Equation 4.1.

$$g(X, B, V, U, q) = \sum_{j=1}^{r} X(j) \cdot c_j + B[qC_q + (V + 2U)C_B] + C_0 \tag{4.1}$$

The variables B and V are particular indices of X: $B = X(j_B)$, $V = X(j_V)$. Variable U is the sum of all X indices representing a functional unit, or $U = \sum X(j_U)$, where $j_U = \{j_A, j_M, j_I, j_Y, ...\}$. Finally, $q = U + V$, or $\sum X(j_U) + X(j_V)$. The constraint function, g, is defined in Equation 4.2.

$$g(X) = \sum_{j=1}^{r} X(j) \cdot c_j + X(j_B) \left[C_q \left(\sum X(j_U) + X(j_V) \right) + \left(X(j_V) + 2 \sum X(j_U) \right) C_B \right] + C_0 \tag{4.2}$$

Appendix A.4 provides the data used for a least squares estimate of C_q and C_B from the measured synthesis data. Slice usage estimates computed with Equation 4.2 are compared to measured slice usage in Table 4.4. In approximately half the cases, the predictions overestimate the usage. Overestimating the slice usage is conservative when evaluating the performance of the VectCore. The average error in the predications is approximately 7% over resource usage levels ranging from about 25% to 140% of the part limit used as the synthesis target.

Table 4.4: VectCore resource estimates, using the Virtex II pro^{TM}vp100 part as the target for synthesis.

q	L	V	U	B	**Act. Slices**	**Est. Slices**	**% ERROR**	**Slice LIMIT**	**% LIMIT**
5	1	3	2	4	10538	11917	-13	44096	24
6	1	4	2	4	10718	12257	-14	44096	24
6	1	3	3	3	12176	12450	-2	44096	28
11	1	7	4	6	15453	16659	-8	44096	35
10	1	6	4	7	15777	16908	-7	44096	36
11	1	6	5	6	16345	17461	-7	44096	37
21	1	13	8	12	27308	27924	-2	44096	62
20	1	12	8	13	27760	28305	-2	44096	63
21	1	12	9	12	29349	28897	2	44096	67
20	1	12	8	24	33660	36620	-9	44096	76
32	1	24	8	12	34109	32056	6	44096	77
40	1	32	8	12	39044	35060	10	44096	89
32	1	24	8	24	43526	41758	4	44096	99
36	1	24	12	12	44094	37448	15	44096	99
40	1	32	8	24	44706	45183	-1	44096	101
36	1	24	12	24	49925	48718	2	44096	113
44	1	24	20	12	53343	48232	10	44096	121
41	1	25	16	24	59684	56106	6	44096	135
40	1	24	16	25	60807	56748	7	44096	138

4.5 Scheduling and Allocation

This section describes characteristics of the scheduling and allocation algorithm included in the objective function of the research problem. The scheduling and allocation problems are defined in the minimization problem formulation in Chapter 3. The heuristics used in the scheduling algorithm are shown in Section 4.5.1 to use lessons learned from vectorizing compilers for supercomputer implementations. The allocation logic described in Section 4.5.2 complements the heuristics, exploiting the capabilities of the architectural template to realize efficient, high-performance vector computations.

4.5.1 Scheduler Algorithm and Heuristics

The scheduling approach in this research uses a greedy heuristic for assigning tasks to available resources. A list scheduling implementation is used, with a priority function composed of a set of weighted heuristics. This approach is used because it builds the schedule in one pass through the input data flow graph. Other types of algorithms such as transformational may involve the enumeration of many potential schedules to arrive at a solution for a given architecture specification [24]. The VectCore minimization algorithm enumerates many schedules for many architecture specifications, so a single pass is preferred to reduce execution time. Figure 4.7 shows an overview of the scheduling heuristic. As shown, the inputs to the algorithm are the computation task graph, G, and an architecture specification, X^i. The outputs are a set of starting times, ϕ, for the tasks in G, and an allocation of the tasks to the resources of X^i, represented as Γ. A set of heuristic weights used by a priority function are parameters of the algorithm. The algorithm functions as a typical list scheduler, building a ready set of tasks whose inputs are ready at the current time step, t. The priority function assigns weights to the ready set tasks and the set is sorted by decreasing weight to

produce a pick set, θ_p.

For each task in the pick set, the first available time for a set of resources required to support the task is determined. If all vector registers are in use, a register is spilled. A register spill adds a store and a load operation to the input operation set. A simple least recently used scheme is used for spill register selection. The scheduling approach uses a greedy heuristic to form vector operation chains. The resource ready time is compared to the data ready time of the inputs for the current task. If the current operation can be chained either directly or by using the output delay stage capability of the functional units described in Section 4.5.2, then the current operation is chained with its inputs. Once a chain is formed, some interconnect and register resources are not needed and can be deallocated. If a chain is not possible, then the inputs must be stored in registers, the ready time for the current task must be updated, and the schedule time index must be advanced. If a chain was successful or the inputs are stored in registers from a previous scheduler iteration, the task time ϕ_i is assigned to the resource available time, the resources are assigned to Γ_i, and the schedule time index is advanced.

The scheduler is designed to only schedule an operation if a register is available to hold the result. In this way, the scheduler can always make progress and no backtracking or multiple passes are required. In the cft77 algorithm register assignment occurs after scheduling [59]. This approach is not used in the VectCore scheduler because a dedicated pass to schedule registers increases the overall execution time of the minimization algorithm. Appendix A.1 provides a specific example of the scheduler operation.

The heuristics used in the priority function are listed with descriptions and design rationale as follows.

- The operation type, h_s, heuristic is included to prioritize STORE operations, because

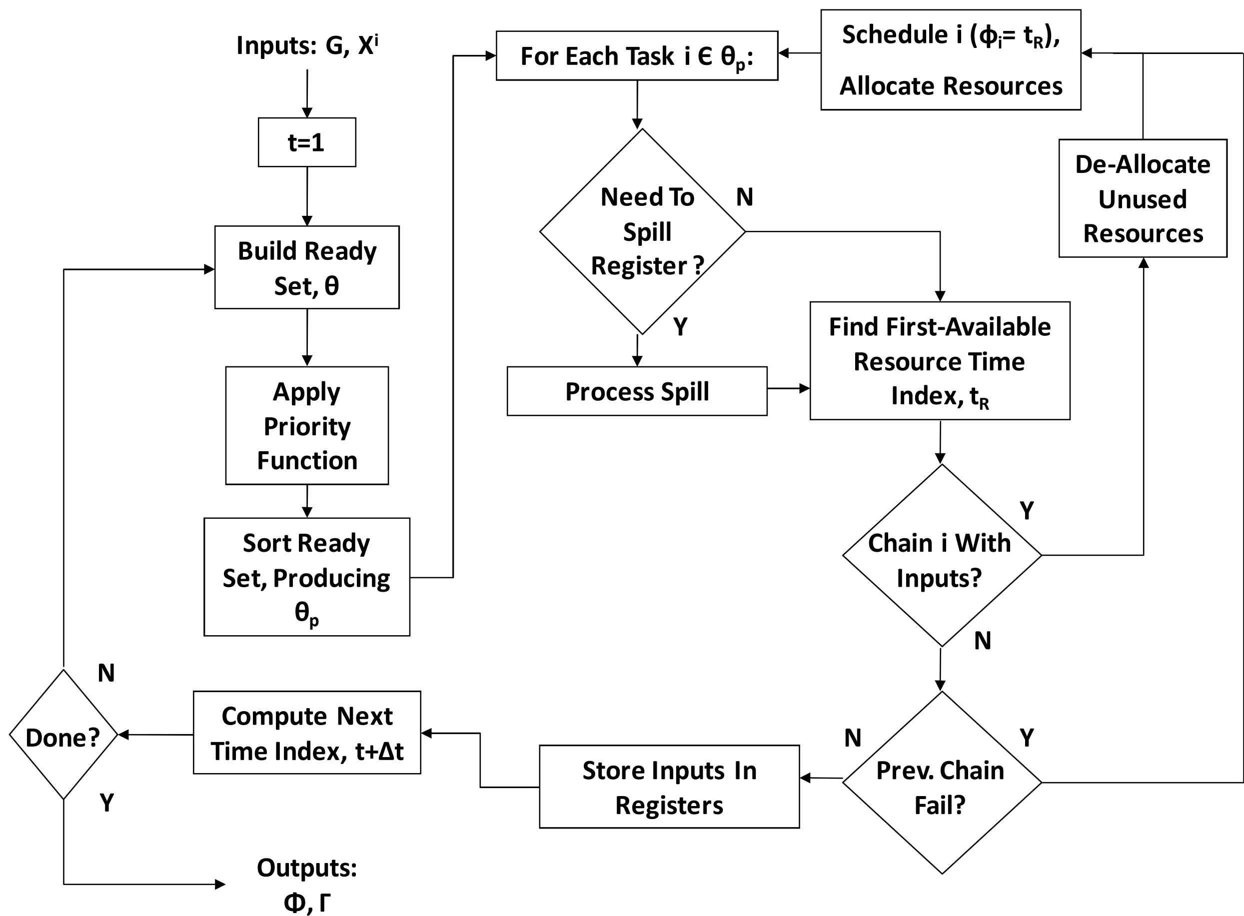

Figure 4.7: VectCore scheduling heuristic. A list scheduling algorithm is used with a priority function that assigns weights to tasks in the ready set. The weighted ready set is sorted by decreasing weight to produce the pick set, θ_p. Resources are assigned to each pick set operation using a greedy heuristic. After assessing the need to spill a register, each operation is checked for potential chain formation with its inputs. Chain failures cause the inputs for an operation to be stored in registers.

a STORE to memory completes an operation chain and allows multiple resources to be released.

- The path length, h_p, is described in Section 2.4 to be more effective than a simple ASAP heuristic, and is used in the Cray compiler.

- The number of descendants, h_d, is also shown to be effective in the literature, when used in combination with the path length heuristic to prioritize operations with equal path length.

- The loop iteration number heuristic, h_i, provides a means to control the amount of inter-loop iteration parallel operation scheduling performed. The cft77 compiler was designed for Simple Vector Scheduling [60], in which all operations within one loop iteration are scheduled before any subsequent iterations. Polycyclic Vector Scheduling [60], [61] introduces inter-loop iteration optimization to improve the performance for an entire loop of operations. The loop iterations are unrolled in the input VectCore assembly code, so operations from multiple iterations are included in a given ready set. Architecture resources needed to chain vector operations compete with those required to execute parallel operations. For independent loops, only operations in the same loop iteration can be chained. The h_i heuristic weighs operations in the current loop iteration higher, prioritizing chaining over parallel operations.

- The last ready set heuristic, h_l, is included to prioritize vector chaining opportunities. Operations not present in the last schedule iteration's ready set are immediate successors of operations just scheduled. Prioritizing these operations facilitates chaining and reduces resource busy times.

The priority function is shown in Equation 4.3.

$$H = h_s \cdot f_s + h_l \cdot f_l + h_p \cdot l_p + h_d \cdot n_d + h_i \tag{4.3}$$

where

$$f_s = \begin{cases} 1 & : \quad i = STORE \\ 0 & : \quad otherwise \end{cases}, \tag{4.4}$$

$$f_l = \begin{cases} 1 & : \quad i \in \theta_{k-1} \\ 0 & : \quad otherwise \end{cases}, \tag{4.5}$$

l_p is the path length of task i, n_d is the unscheduled number of descendants of task i, and θ_{k-1} is the ready set for the last schedule iteration. Values for the heuristic weights are provided in Section A.1.

The VectCore scheduling approach is to leverage the amount and maturity of practical experience in scheduling algorithms and heuristics in the vector processing domain. Section 4.5.2 describes the allocation logic that uses the capabilities of the hardware architecture to facilitate chained and parallel vector operations.

4.5.2 Allocation

The flexibility of the VectCore architectural template allows for allocation options not found in traditional vector computers. These options facilitate chained and parallel vector operations used widely in high performance vector computers. The VectCore architecture allows multiple chains to fanout from a single producer operation. The output of an operation can also be stored in a register while simultaneously forming a chain with a subsequent operation. Additionally, the VectCore approach avoids complex instruction issue logic by placing the responsibility of identifying chaining opportunities and avoiding resource conflicts on the scheduler. By comparison, a traditional vector architecture such as the Cray-1 has a register-to-register datapath. All functional unit operands are read from registers and all results are stored to registers. [57]. Therefore, chaining on the Cray-1 must occur through the registers,

and the number of register ports limits the amount and variety of chain formation. Direct chaining between functional units in the VectCore also reduces live register requirements.

Unlike the VectCore, chaining on the Cray-1 is accomplished by the instruction issue logic as a hardware operation, and not handled explicitly by the scheduler. During instruction issue, the logic performs conflict checks and holds an instruction if conflicts exist. This functionality introduces complexity in the instruction issue logic. Vector To Scalar Transmission Instructions can be inserted by the Cray-1 compiler to align operations for chaining [62]. A capability similar to this is included in the VectCore scheduler, implemented as selectable delay stages for the output of the vector functional units. This capability provides a means to increase direct chaining opportunities.

An example of VectCore flexible chaining scenarios is shown in Figure 4.8. In the figure, A_i^k denotes task i is an add operation allocated to adder resource index k. S represents a store operation. Part (A) shows a task graph for a set of operations, and part (B) shows the task allocation to resources, for an architecture specification $1L, 3V, 2A, 1M, 3B, 0Y, 0I$. The operation M_1 forms a chain with operations M_2 and A_3, but adder resources do not permit an immediate chain with operations A_4 or A_5. A_4 and A_5 are not ready to consume the output of M_1, so the output must be stored in a register. Multiplexers on the inputs of the functional units and the inputs of the registers are independent, so M_2 and A_3 can consume the output of M_1 while it is simultaneously being stored in V_1.

The store operation, S_6, requires its data to come from a register. S_6 uses a dedicated port of the dual-port register, allowing the output of M_1 to be written to V_1 while S_6 reads V_1. S_6 can begin as long as the same address is not asserted on the two ports at the same time. A_4 must use the same port to read V_1 as M_1 uses to write, so A_4 cannot be configured to read V_1 until the M_1 write is complete. Similarly, A_5 shares the same port, so it must execute serially with A_4.

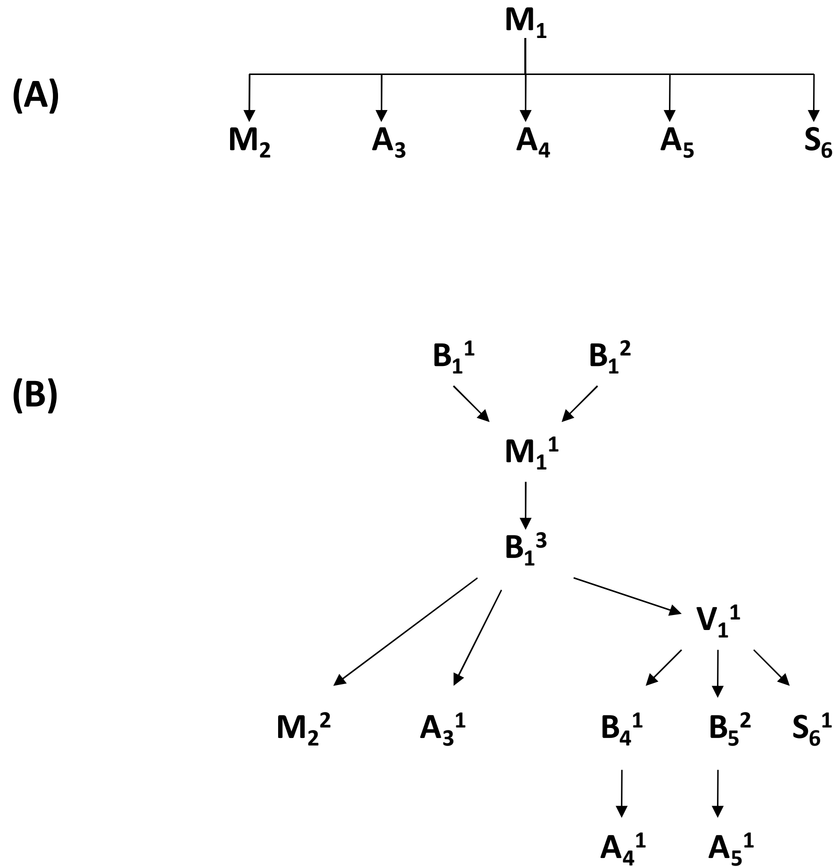

Figure 4.8: VectCore allocation example. Part (A) is a task graph for an example computation, and part (B) shows the allocation of resources to the tasks for an architecture specification $1L, 3V, 2A, 1M, 3B, 0Y, 0I$

Two direct and one flexible chain among four operations is demonstrated in this example. The resources used are three functional unit buses and only one register. All the basic capabilities of the architecture like those shown in this example are verified with hardware testing. The test matrix used for the verification is provided in Appendix B.

4.6 VectCore Design

As described in Section 4, the VectCore approach is to tailor an architectural template for an input computation, and run the computation on the architecture via a microcode program. The microcode runs on the FPGA, but originally resides on the general-purpose processor (see Figure 4.1). Software running on the general-purpose processor interfaces with the VectCore design to transfer the microcode between the processors. Figure 4.9 shows the hardware implementation step of the VectCore approach. As shown, the microcode is a representation of the schedule, ϕ, and allocation, Γ, determined by the minimization algorithm. The schedule and allocation are composed of event (operation) starting times and resource configurations, respectively. These event times and configurations must be provided to the processing resources that will execute the event operations. The method to deliver this data to the VectCore resources must provide the events in the proper order and with the proper timing to ensure correct execution. Additionally, the performance overhead and resource cost of the interface must be low to maximize the effectiveness of the VectCore approach.

The format for the event times and configurations is the VectCore Schedule-Packet (S-PAK). An S-PAK is a low-level microcode word containing a start time and resource configuration data for a schedule event. The size of each S-PAK word is fixed for a given maximum number of resources, and number of resource types. Details of the S-PAK format are provided in Section A.3.

Section 4.6.1 describes a low overhead, low resource cost, interface and dispatch scheme that transfers S-PAKs from the general-purpose processor to the VectCore resources. The associated control approach maintains proper schedule execution order and relative timing of schedule events in the presence of uncertainties in resource ready times, while avoiding

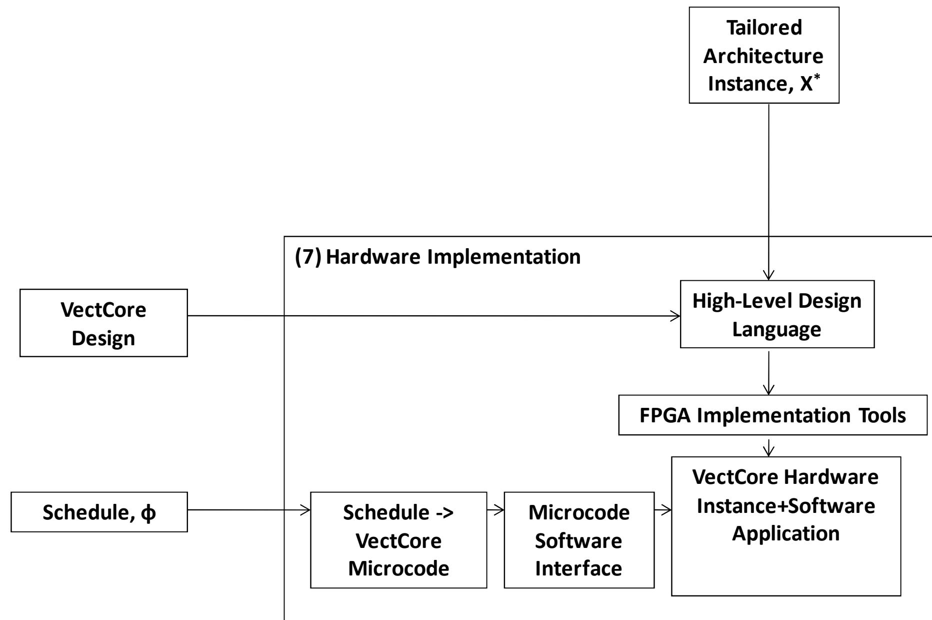

Figure 4.9: VectCore approach hardware implementation.

the need for complex instruction issue logic as found in a superscalar processor [8].

Section 4.6.2 provides an analysis of the overhead of the VectCore interface, dispatch, and control scheme. The analysis shows that the VectCore overhead increases slowly with available resources. Mitigation strategies for this characteristic of the overhead are discussed.

4.6.1 Event Dispatch and Control

Low overhead schedule execution requires either a high bandwidth interface between the general purpose processor and custom core, or low bandwidth requirements for this interface. The bandwidth is implementation-dependent, and tightly-coupled interfaces are available (see experimental platform details in Chapter 5.1). Minimizing the amount of information transmitted through the interface and hiding the latency of the interface with pipelined operations reduces the overhead. Both approaches are applied in the VectCore design. The number of S-PAK bits is reduced by not including complex routing information or unique bit

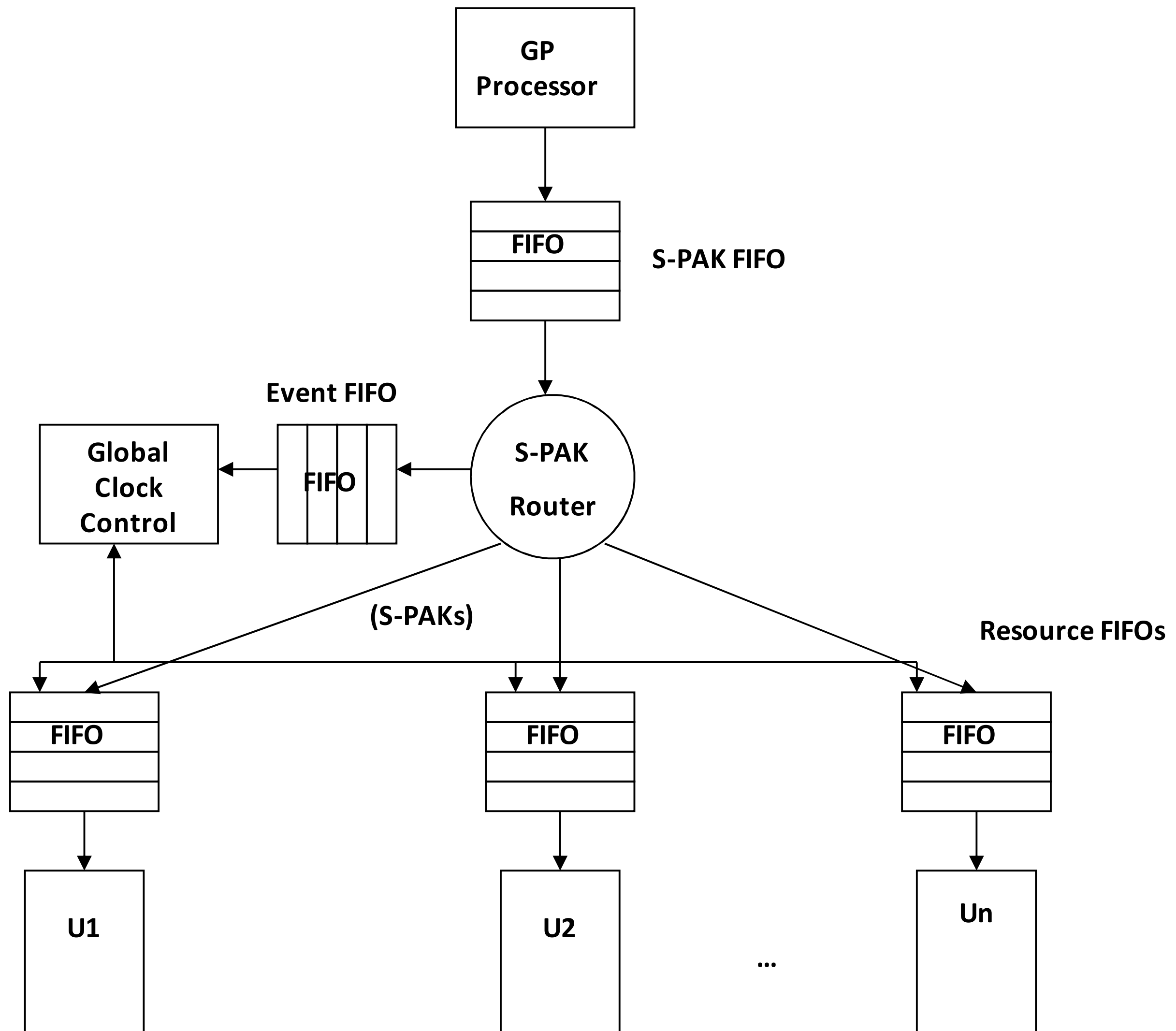

Figure 4.10: VectCore interface and S-PAK dispatch scheme.

fields for each resource as found in a VLIW approach [8]. Figure 4.10 shows a block diagram of the interface and S-PAK dispatch architecture.

Starting at the top of the figure, S-PAKs are written by the general purpose processor to the S-PAK First In First Out (FIFO) buffer interface of the VectCore. The S-PAK router forwards each S-PAK to resource FIFO buffers for each appropriate computing resource. Independent state machines for these resources enable parallel independent execution of

each resource. When all the S-PAK configurations for a given schedule event have been read from the S-PAK FIFO, a control word is written to the event FIFO for consumption by the global clock control logic. The design approach of using independent FIFOs in the communication path to each system resource allows the resources to execute at a maximum rate for the duration data is available in the resource FIFOs. The FIFOs of slower resources will tend to fill, ensuring that data is available for these resources to execute at their full rate.

VectCore execution control avoids complex instruction issue hardware by leaving all conflict checking and construction of parallel operations to the scheduler and by using a global clock control approach. The global clock is an index that directly correlates to the event start times output by the scheduler. The relative start times between events must be executed exactly as scheduled for the result of the input problem to be computed correctly. An overview of the global clock control logic is shown in Figure 4.11.

Proper execution timing is ensured by including a capability for all the resource pipelines to be stalled until all necessary resources are ready to support a given global clock cycle. The control logic maintains the global clock count, and the next schedule event time. If the global clock count reaches the next event time before all the required resources are ready, the logic asserts a hold signal. The hold signal stalls all the data pipelines in the VectCore. The S-PAK router continues operation during a hold condition, and state machines that control each resource only stall if the resource is operating on data. These two design characteristics allow unscheduled resources to read configurations and progress to a state of readiness for the next event. When all resources are ready, the hold signal is de-asserted, allowing the pipelines and the global clock to resume. More details of the global clock logic are in Appendix A.7.

The results in Section 6.4.3 show the VectCore dispatch and control architecture tends to

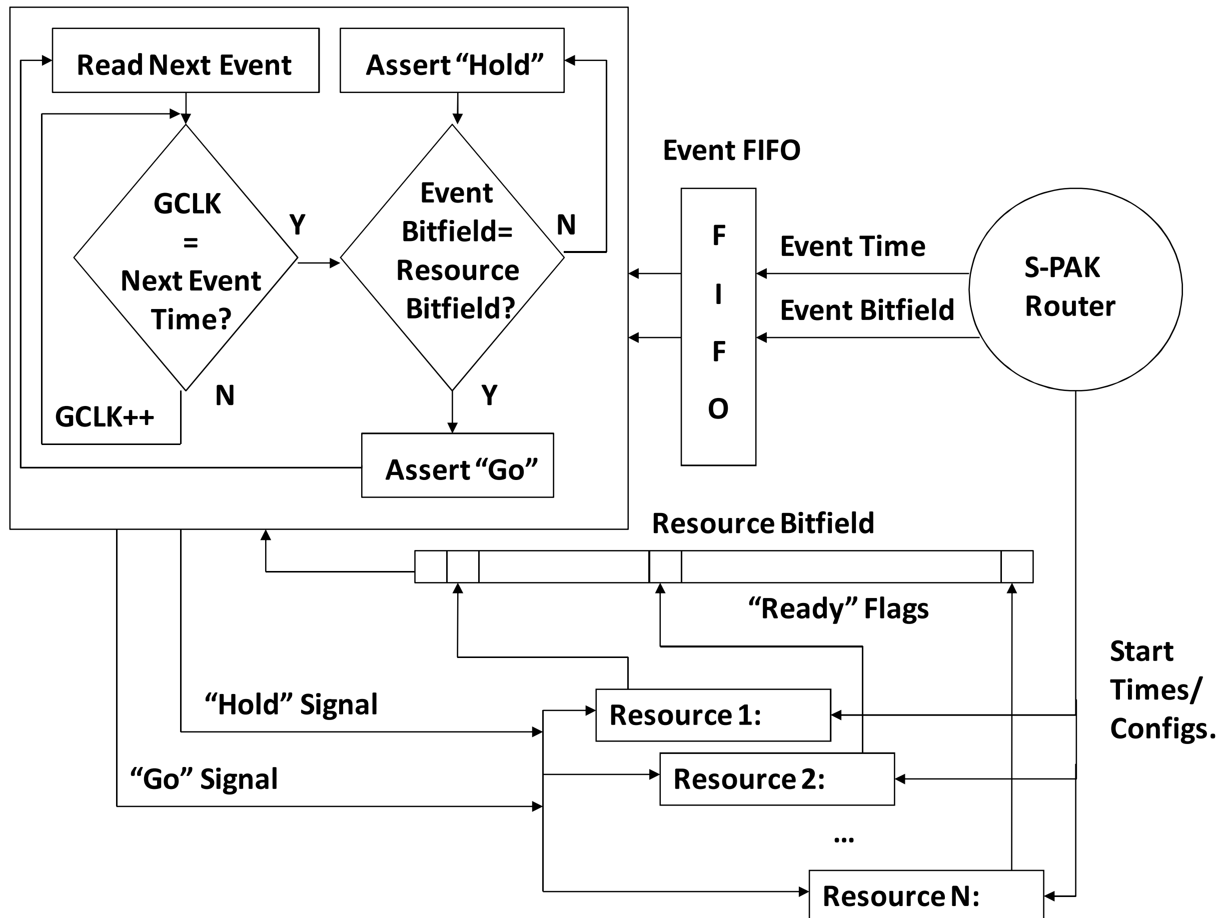

Figure 4.11: Global clock control operation description. A new schedule event, which includes a starting time and a resource configuration, is read from the event FIFO. If the global clock has reached the next event start time, the event bit-field is compared to the resource bit-field. Otherwise, the global clock is incremented. If all resources scheduled for the next start time are ready, the resource bit-field matches the event bit-field and the "go" signal is asserted. If the resources are not ready, the "hold" signal is asserted to stall all active processing resources until all resources executing on the next schedule event indicate ready.

hide the latency of the processor to VectCore interface. As long as the execution time of a schedule event is large compared to the latency of each S-PAK write, the event FIFO will become full waiting for schedule events to complete. When the event FIFO is full, the S-PAK router stalls until the next event can be written, allowing the S-PAK FIFO to fill. As long the S-PAK FIFO is not empty, the S-PAK router is not affected by the processor to VectCore interface latency. For long vector lengths, which are characteristic of the types of problems considered in this research, long schedule event execution times are prevalent.

4.6.2 Overhead Analysis

The overhead of the VectCore interface and control is the sum of the number of VectCore hardware cycles the hold signal is asserted, and the VectCore startup latency. The startup latency is the number of VectCore cycles from the first S-PAK write to the execution of the first schedule event. The global clock control logic cannot issue the signal to execute the current schedule event before reading the next schedule event from the event queue. If the global clock starts without knowledge of the next event time, the clock could reach that time before the next event is executed and a required hold condition will be missed.

Multiple S-PAKs may share the same schedule start time for operations scheduled to execute in parallel. The latency is computed using the number of events that execute on the first two schedule event times ($e_1 = \min_{1 \le i \le n}(\{\phi_i\}\ )$ and $e_2 = \min_{1 \le i \le n}(\{\{\phi_i\} - e_1\})$), the number of cycles between S-PAK writes, L_s, and the latency between event read and resource start, L_d, as shown in Equation 4.6.

$$latency = (|\{\phi_i : \phi_i = e_1\}| + |\{\phi_i : \phi_i = e_2\}|)L_s + L_d \tag{4.6}$$

The latencies are design and implementation dependent. For the VectCore implementation tested in Chapter 5, L_s and L_d are 20 and 10 core cycles respectively. These low values, applied as shown in Equation 4.6, ensure startup latency is not the dominant overhead factor. The number of core hold cycles is not as straightforward to predict, so empirical measurements from hardware experiments are provided in Section 6.4.3 to characterize this overhead factor.

As described in Section 4.6.1, long schedule event times allow the FIFOs in the dispatch architecture to fill, and this condition can hide the latency L_s. Adjustments to the FIFO depths in the design could maximize this benefit. More pipelining of the schedule events in the global clock control logic could reduce the latency L_d. These mitigation approaches are not tested in this research.

Chapter 5

Experiment Design

The VectCore approach is evaluated with a fully operational end-to-end implementation. The system described in this chapter accepts as inputs a VectCore assembly representation of a computation and a resource limit. The outputs of the system are a VectCore co-processor instance tailored to the computation and the microcode program to execute the computation on the co-processor. Sufficient development of the research approach to enable full end-to-end hardware experimentation provides a strong context of "real" implementation details for the experimental data.

The VectCore implementation does not include the functionality necessary to run all the problems selected for experimentation end-to-end. Only vector add and multiply functional units are supported in hardware. This limits full VectCore implementation to one of the six benchmark problems. The full end-to-end runs are therefore performed on this problem to characterize the overhead of the VectCore processor implementation. Chapter 6 shows this overhead is small enough compared to the schedule length that the scheduler output provides an accurate basis for comparing the problems[1]. The remaining performance data is scheduler

[1]End-to-end implementations are only required for a characterization of the S-PAK dispatch and control overhead for the VectCore. Sufficient characterization of this overhead was performed using the TASS problem.

output, collected prior to the hardware implementation step described in Section 4.6.

Details of the evaluation system design are provided in Section 5.1. The input cases selected for experimentation are described in Section 5.2. This chapter concludes with a derivation of a Floating-Point Operations Per Second (FLOPS) performance prediction equation used in the analysis of the experimental results presented in Chapter 6.

5.1 Evaluation System

Figure 5.1 depicts the evaluation system used for this research. In the figure, solid lines indicate fixed physical components of the system, and the dotted lines indicate components of a VectCore processor instance. The system is implemented on the Dini Corporation [63] DN6000K10S board. The board features a Xilinx®Virtex II pro™vp70 FPGA [58]. The VectCore is instantiated as a custom peripheral for one of the two embedded PowerPC™processors on the FPGA. The Xilinx®Embedded Development Kit (EDK) environment [64] is used to implement the VectCore processor. Four independent 133MHz 512K X 36 bit Static Random Access Memory (SRAM) chips provide the external memory for the VectCore design. The Virtex II pro™Processor Local Bus (PLB) is used as a tightly-coupled interface between the embedded PowerPC™processor and the custom peripheral. Design parameters for the implementation are summarized in Table 5.1.

5.2 Benchmark Input Cases

The benchmark problems used for VectCore performance analysis are chosen to meet two criteria: relevance to the research application domain of numerical physical simulations, and problem characteristics impose a range of implementation challenges to achieve high performance. The first criterion ensures relevance to the motivation for this work, and the second

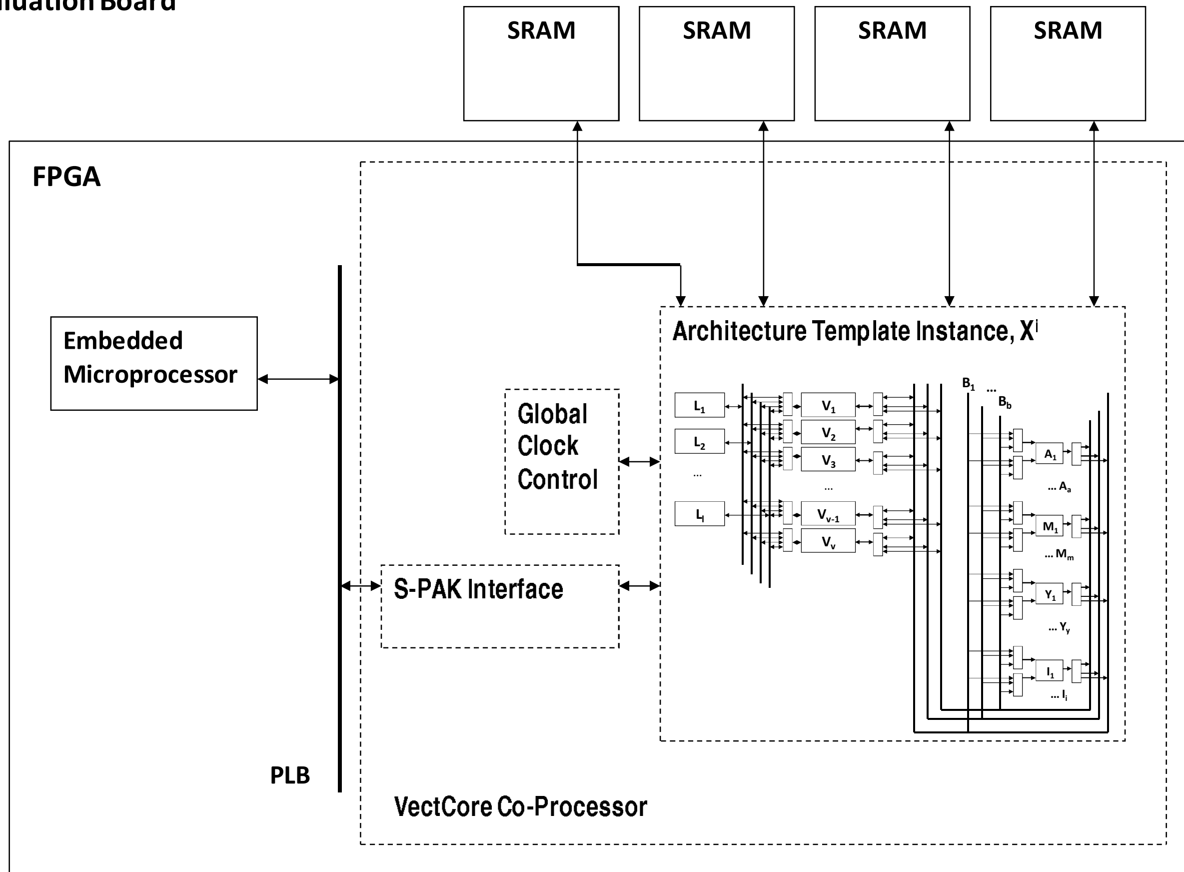

Figure 5.1: VectCore evaluation system organization.

Table 5.1: FPGA and EDK design configuration data.

FPGA	xc2vp70
Package	ff1704
Speed Grade	-5
Processor	ppc405
Processor Frequency	100 MHz
PLB Frequency	100 MHz
OS	Standalone
SRAM Frequency	133 MHz
VectCore Frequency	133 MHz

tests the ability of the VectCore approach to discern between different problem performance limitations and effectively tailor available resources to meet the requirements of a particular problem. Matrix operations arise frequently in science and engineering applications. As presented in Section 2.1, there are many examples in the literature of architectures that are evaluated with matrix computations due to their relevance to this application domain.

Matrix-by-matrix multiplication ($ax = y$, where a, x, and y are matrices) requires more computations than input/output operations, so it provides a computation-bound test case. Many of the computations are data-independent, so the available parallelism is high for this problem. Further, different operation orders have been shown [65] to present different challenges to performance on parallel and vector implementations. The selection of matrix multiplication orderings is described in Section 5.2.1.

The triangular-solve, or the back-substitution step of the LU decomposition method, is another matrix operation widely applicable in the scientific computing domain. The triangular-solve is used to find the solution to the linear equation $ax = b$, where a is a square matrix and x and b are vectors. A characteristic of this problem is a long dependency chain between the operations. As shown in Section 5.2.2, vector implementations of the triangular-solve operate on consistently increasing or decreasing vector lengths as the solution progresses. The available parallelism for these problems is lower than the matrix multiplications, in which the vector lengths are fixed.

Finally, an inner loop of the TASS case study problem introduced in Chapter 1 is also chosen as a benchmark case. The TASS problem represents a real example from the application domain targeted by this research, a numerical weather simulation code tailored for vector implementation. There are few inter-loop data dependencies in the chosen code, so the available parallelism is high. There are more memory load and store operations in each loop iteration than computations, making the TASS case a memory-bound benchmark. More

details of the glstass benchmark are provided in Section 5.2.3.

5.2.1 Matrix Multiplication

In the multiplication of two matrices $ax = y$, the basic operation is an update of an element in the result matrix y, as shown in Equation 5.1.

$$y_{ij} = y_{ij} + a_{ik}x_{kj} \tag{5.1}$$

The subscripts i,j,k are loop indices corresponding to the dimensions o_m, o_p, o_n, where matrix a has dimensions o_m by o_n, x dimensions o_n by o_p, and y dimensions o_m by o_p. Dongarra [65] discusses the impact different orderings of these loop indices have on the performance of the problem on a "Cray-like" machine. Due to this sensitivity of vector implementations to the orderings, several are chosen for the matrix multiplication benchmark problems. The six possible orderings are ijk, jik, kij, kji, ikj, and jki, where the index triplets define the indices used for a triple-nested loop around Equation 5.1, with the leftmost index of the triplet corresponding to the outermost loop. The orderings have different core basic vector operations for their implementation, as well as different data access patterns. In [65], diagrams similar to Figure 5.2 are used to compare the behavior of the orderings.

The double-ended arrows in Figure 5.2 represent a vector operand or result, and the "x" symbols represent a scalar operand or result. For example, the jik (and ijk) ordering performs a vector inner product as its basic operation. A column of the y result matrix is built one scalar element at a time with a series of inner products between the rows of the a matrix and the corresponding column of the x matrix.

In the kji order, each column of a is multiplied by successive elements of a row of x and accumulated to form the columns of y. The kji order must maintain live register values for

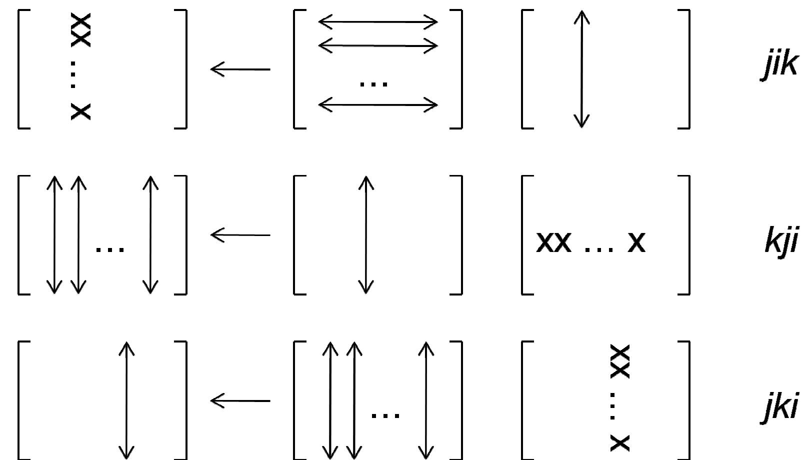

Figure 5.2: Three orderings for the matrix multiplication.

each column of y until the accumulations are complete.

The jki ordering accumulates the multiplication of successive columns of a by successive elements of a column of x to build the columns of y. The jki order builds one complete column of y at a time, so the live register times for jki order are less than the kji order. The basic operation performed in the kji and jki orderings is the $SAXPY$.

Only the three orderings shown in Figure 5.2 are used for the benchmark cases. The remaining three orders only differ in the transposition of a row or column in one of the matrices.

The orderings also determine whether the matrices are loaded and stored as columns or rows. A representative example of the assumed storage scheme for the matrices in the evaluation system SRAM is shown in Figure 5.3 for the jik matrix multiplication. It is assumed that a compiler used with a VectCore system can populate an SRAM cache for VectCore use with data in a manner conducive to the memory access pattern of each input problem. This assumption offers more flexibility in memory access than other alternatives, such as Dynamic Random Access Memory (DRAM). Column-major matrix storage is assumed, so vector load/store with stride (see Appendix D.1.3) operations are assumed available for operations involving matrix rows. Also assumed is that a load/store and load/store with

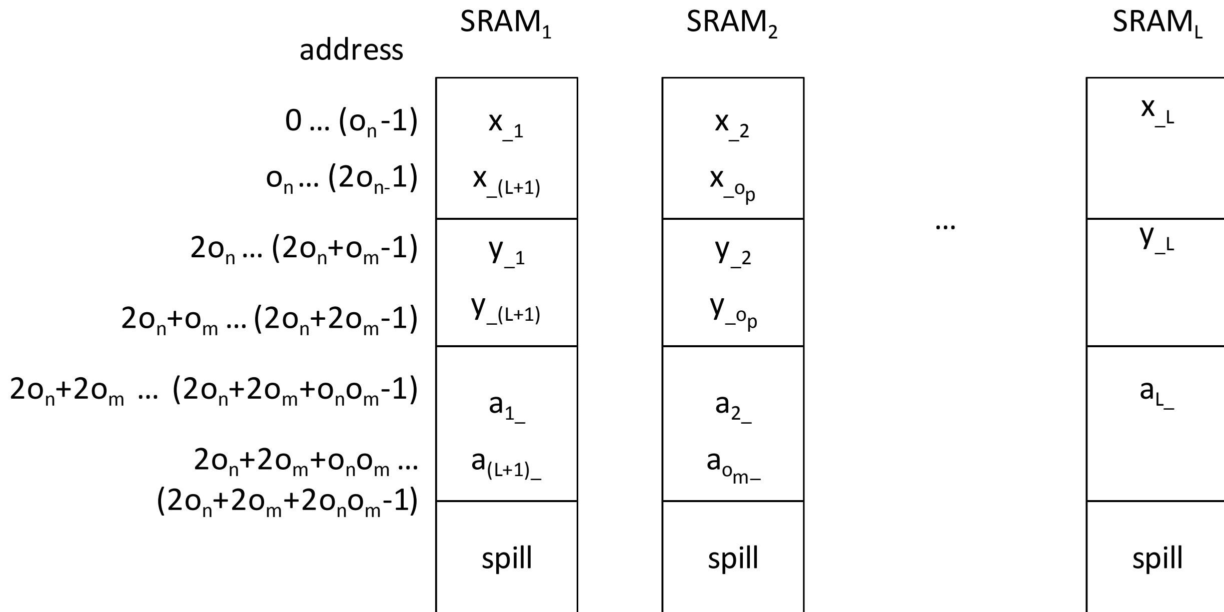

Figure 5.3: Example *SRAM* organization for *jik* matrix multiplication. The nomenclature $a_{1_}$, for example, denotes the first row of matrix a. It is assumed that the compiler for a legacy application targeted by the VectCore approach stores multi-dimensional arrays in column-major order, so the data for the columns of the operand or result matrices occupies contiguous addresses. A buffer for register spills is allocated in each *SRAM* bank.

stride have the same startup latency and similar resource costs. Currently, only contiguous load/store units are implemented in the VectCore hardware.

As shown in Figure 5.3, the approach to the memory layout is to spread the rows or columns of each matrix, depending on how the application loads and stores its data, across the available memory banks to reduce the potential for address conflicts. This scheme applies to all the benchmark problems in a manner similar to that shown for the *jik* problem. A characterization of the potential address conflicts is provided in the discussion of each problem that follows.

Characteristics of the three matrix multiplication problem orderings produce different resource contention scenarios in the VectCore implementation. A discussion of these scenarios follows that refers to task graphs for these cases shown in Figures 5.4 through 5.6.

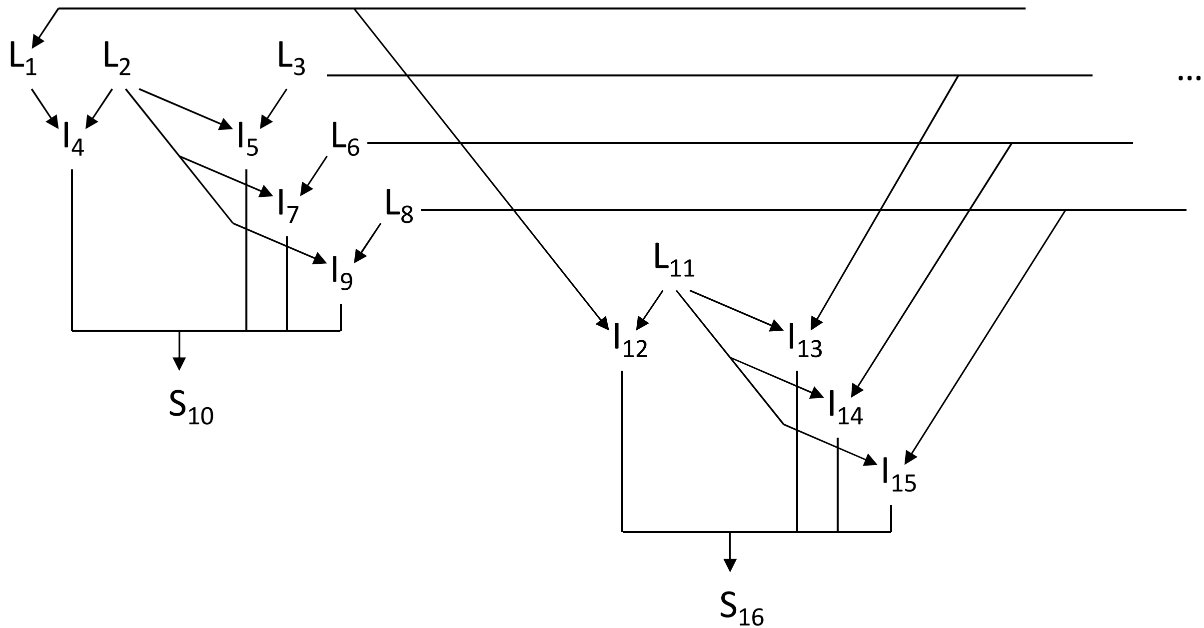

Figure 5.4: Partial task graph for *jik* matrix multiplication, with o_m, o_n, o_p = 4, 4, 4.

The group of vector operations on the left side of the *jik* task graph shown in Figure 5.4 loads a column of the matrix x and each row of the matrix a for each of four vector inner product operations. Subsequent groups can read these operands from registers unless a spill is required. The inner product operations within a group must execute serially due to contention reading the vector register that holds the first column of x. Inner product operations in different groups may execute in parallel as long as the register reads of the same row of the a matrix do not overlap in the schedule. Similar to contention reading registers, if $L < o_m$, o_m/L rows of a must be stored in the same SRAM bank. Inner product operations that load these rows of a must execute serially.

The *kji* ordering partial task graph in Figure 5.5 shows groups of o_p vector $SAXPY$ operations, where each group takes a row of x and a column of a as its inputs. The output shown connected back to the input of the $SAXPY$ operation indicates that each $SAXPY$ operation is one of a series that accumulates partial sums of a column of y. When the series of

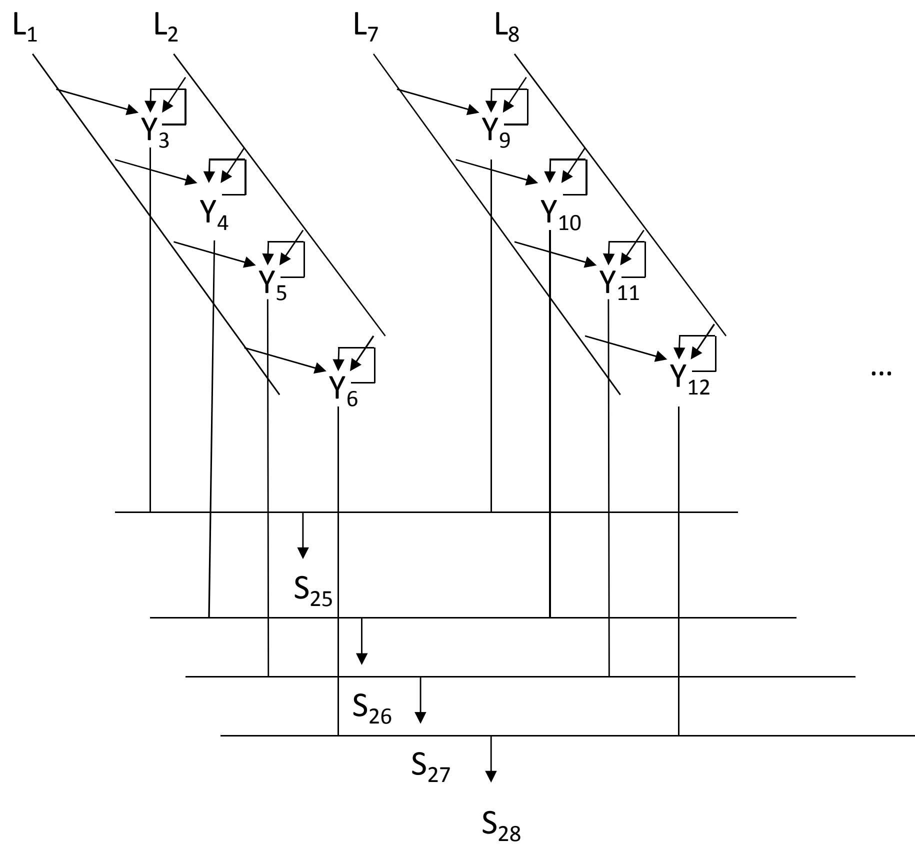

Figure 5.5: Partial task graph for kji matrix multiplication, with o_m, o_n, o_p = 4, 4, 4.

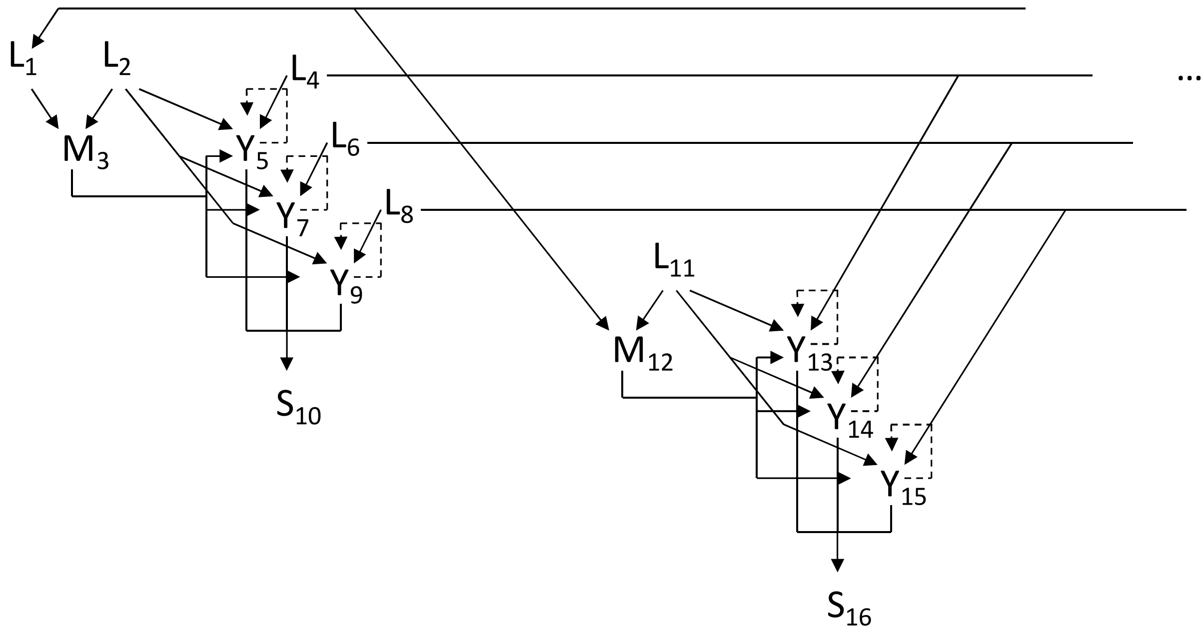

Figure 5.6: Partial task graph for jki matrix multiplication, with o_m, o_n, o_p = 4, 4, 4.

accumulations is complete, the column of y is stored. The partial sums must be accumulated in registers prior to storing.

$SAXPY$ operations in a series (horizontally across groups in Figure 5.5) can form vector chains, and individual $SAXPY$ operations in different groups may execute in parallel. Contention reading the same column of a either from a register or memory causes $SAXPY$ operations within a group to execute serially. There are $2(o_n/L)$ potential memory read conflicts using the SRAM storage approach shown in Figure 5.3. For the kji problem, it is necessary to use different storage banks for the rows of x than the same-indexed columns of a to avoid address conflicts within groups.

Live register requirements are potentially less for the jki problem compared to the kji problem. Figure 5.6 shows the jki ordering task graph, where groups of vector multiplications followed by $o_n - 1$ $SAXPY$ operations compute and store a column of y. The dotted connection shown for each $SAXPY$ operation indicates that each $SAXPY$ updates and

accumulates the output of the multiply operation.

Each group of operations reads one column of x and all columns of the a matrix. $SAXPY$ operations within a group can form vector chains or must execute in serial. Individual $SAXPY$ operations in different groups that access different columns of a can execute in parallel. There are $[o_p(1+o_n)]/L)$ potential memory read conflicts in the jki problem, assuming every operation accesses memory for its operand. Live registers reduce the potential for memory conflicts.

5.2.2 Triangular Solve

In LU decomposition, a lower triangular matrix L and upper triangular matrix U are formed such that $LU = a$, where a is a coefficient matrix from the original linear equation to be solved $ax = b$. This decomposition is then used in the original linear equation to produce the following system of equations [30].

$$a \bullet x = (L \bullet U) \bullet x = L \bullet (U \bullet x) = b \tag{5.2}$$

$$L \bullet y = b \tag{5.3}$$

$$U \bullet x = y \tag{5.4}$$

Equation 5.4 is the triangular-solve benchmark case. The operations performed are shown in Equation 5.5. Both inner product and $SAXPY$ operations can be used in a vector implementation of the triangular solve, and Figures 5.7 and 5.8 show the respective task graphs for these two implementations of a 4 by 4 upper triangular matrix.

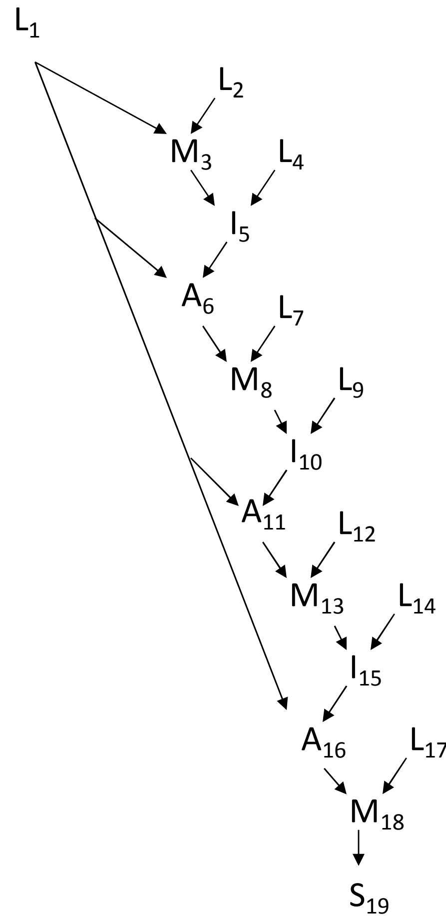

Figure 5.7: Task graph for dimension 4 x 4 upper triangular solve, inner product operation.

$$x_N = \frac{y_N}{u_{NN}}$$

$$x_i = \frac{1}{u_{ii}} \left[y_i - \sum_{j=i+1}^{N} u_{ij} x_j \right], \qquad i = N-1, N-2, ..., 1 \tag{5.5}$$

As shown in Figure 5.7, the result vector x is not stored until the end of a string of dependent vector operations. The structure of the problem provides no opportunities for parallel vector execution. The structure of the problem also does not consistently allow vector operations

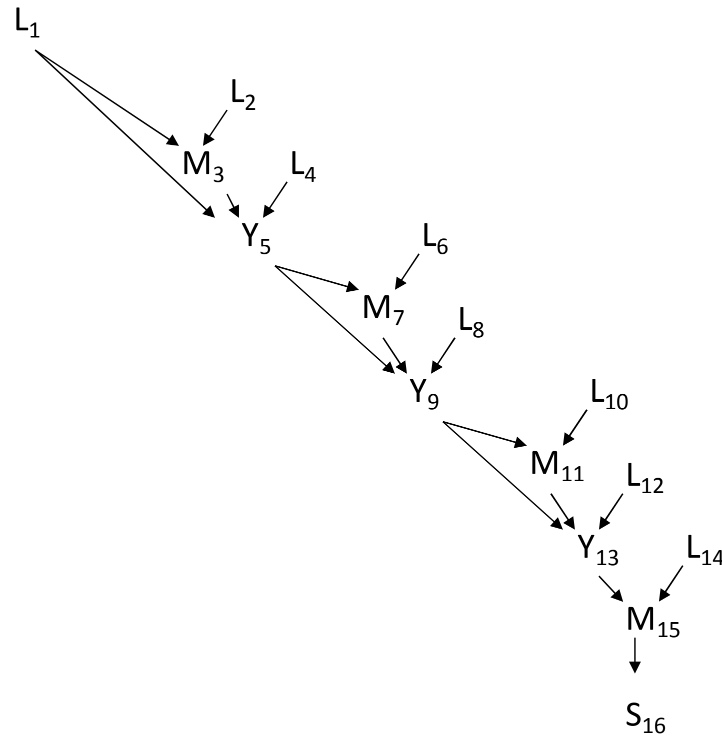

Figure 5.8: Task graph for dimension 4 x 4 upper triangular solve, $SAXPY$ operation.

that are the full length of the result vector. Only the last inner product operation in Figure 5.7, for example, uses a vector length equal to the result vector length. The task graph for the $SAXPY$ implementation in Figure 5.8 shows similar data dependencies. There are chaining opportunities for both implementations, but these are all on operations of vector length one, reducing the effectiveness of the vector chains. The essentially serial dependency chain is minimally affected by memory or register read contention.

5.2.3 TASS Case Study

The code section used for the TASS case study is described in Section C.4. It is a triple-nested loop representative of the code in a routine identified to be a performance bottleneck in the TASS application. Figure 5.9 shows a task graph for one loop iteration where the loop indices (listed from outer- to inner-most) are $j = 2$, $k = 2$, and $i = 4$.

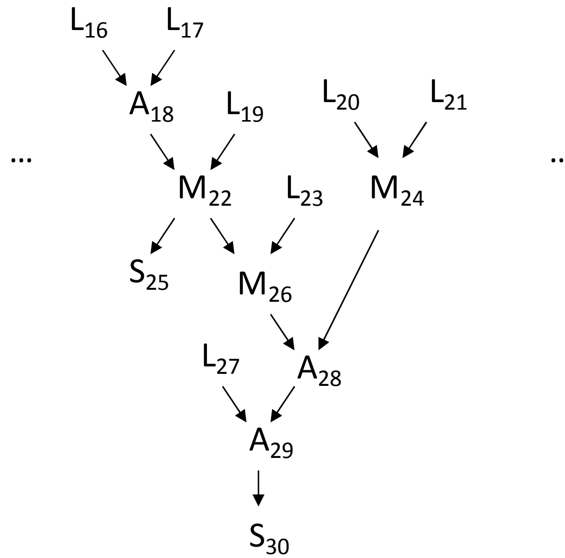

Figure 5.9: Partial task graph for TASS loop, with j, k, i = 2, 2, 4.

As shown, vector load/store operations dominate each TASS loop iteration. Memory bandwidth therefore limits the performance of this problem. There are several chaining opportunities within each loop iteration, and operations in different loop iterations can execute in parallel. Seven loads supply the data for each loop iteration, and there are $(7/L)jk$ potential memory read conflicts.

A variation of the TASS loop referred to as "*mtass*" is also used, where two of the vector by scalar multiplication operations in each loop iteration (not shown explicitly in Figure 5.9) are replaced with vector by vector multiplications. This modification supports the end-to-end hardware tests, because the current version of the VectCore does not have vector by scalar operations implemented in the hardware design. The addition of these capabilities is a straightforward extension of the current hardware design. Vector by scalar operations are supported in the scheduler, allowing performance measurements to be obtained. The ability to evaluate the performance of new hardware units prior to hardware implementation is a utility of the VectCore framework. This capability is discussed further in a custom instruction evaluation for the TASS loop in Section 6.8.

5.3 Vector Performance Parameters

A primary performance metric for the benchmark problems is the FLOPS performance. The number of processor clock cycles to execute a vector sequence of length v_l is $T_{v_l} = T_s + v_l$, where T_s is the startup overhead due to functional unit and memory pipeline startup latencies. The ratio of the number of floating-point operations v_l to T_{v_l}, multiplied by the processor clock frequency ω, yields the FLOPS performance of a single VectCore vector operation. The benchmark cases are all implemented with a sequence of similar repeated vector operations. Multiplying the numerator of the ratio by the number of floating-point operations in one sequence, O_f, for a particular benchmark and the denominator by the number of vector operations in the same sequence, O_v, yields the FLOPS performance per sequence.

The VectCore supports chained and parallel vector operations to enhance performance. Average numbers of parallel vector operations, active vector chains, and chain length are metrics collected for the analysis presented in Chapter 6. If the number of parallel vector operations is less than the number of operations in a sequence, the effect will be to execute the sequence in fewer processor cycles. For example, if a vector sequence has six operations and the average parallel operations is two, then on average the sequence will execute its number of floating-point operations in half the cycles. If the number of parallel operations is greater than the sequence length, then the number of parallel operations multiplied by the number of floating-point operations in a sequence yields the number of floating-point operations executed on average per a sequence duration.

A vector chain of two operations, v_1 and v_2, reduces the execution time for the pair by combining the two operations into one pipelined operation with a startup latency of $T_{s(v_1)} + T_{s(v_2)}$. Equation 5.6 is a FLOPS performance model that incorporates the average number

of parallel vector operations, O_p, the number of vector chains per sequence, O_c, and the average vector length, T_c[2].

$$FLOPS = \left[\frac{(O_f)(v_l)(O_p)}{(O_v - 2(O_c))(T_s + v_l) + \left[\frac{2(O_c)}{T_c}\right] [(T_c)(T_s) + v_l]} \right] \omega \tag{5.6}$$

The first term in the denominator of Equation 5.6 computes the cycles to execute the non-chained operations in the vector sequence, and the second term computes the contribution of the chained operations to the execution time. Equation 5.6 does not account for the impact of register spills on the FLOPS performance. Additionally, active vector chains are not also counted as parallel operations in the current VectCore implementation. A chained pair of vector operations overlaps partially in the schedule, but does not indicate two parallel operations where the overlap occurs. Including the periods of overlap for the chained operations in the parallel operations metric would over-estimate this metric. But a result of this approach is independent parallel vector chains are not counted as parallel operations. To illustrate this limitation, a portion of a schedule for the jki problem is shown in Figure 5.10. In the center of the figure is a sequence of chained $SAXPY$ operations. A portion of the prior and next sequence, which are independent chained sequences, overlap partially with the center sequence. Equation 5.6 underestimates the FLOPS performance in this case. However, Equation 5.6 is a useful approximation for the analysis of the relative performance of the benchmark problems, as discussed in Chapter 6.

[2]It is assumed that vector chains occur only between operations within a vector sequence.

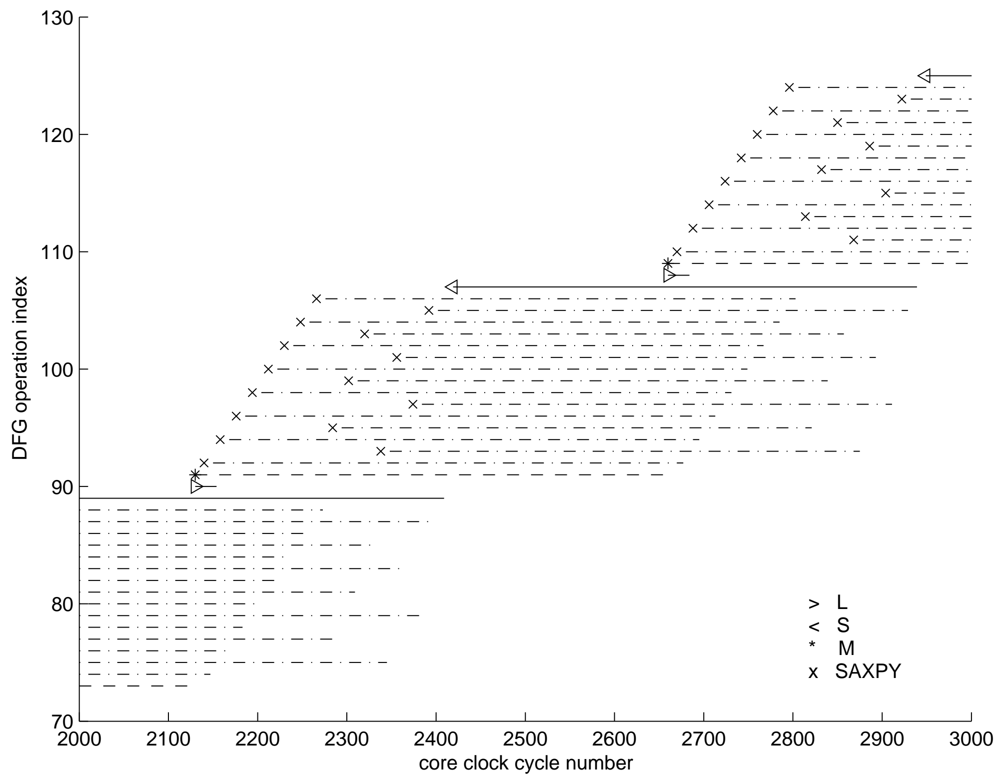

Figure 5.10: A portion of a jki schedule illustrating the potential for parallel independent vector chains.

Chapter 6

Experimental Results

VectCore implementations are evaluated in this chapter. The results presented in Section 6.2 show the VectCore solutions perform better in terms of FLOPS performance and floating-point unit utilization for a range of problems than a fixed vector architecture. The current VectCore implementation is shown not to perform better than a GPP implementation running libraries optimized for the input problem, but does outperform the GPP on the TASS benchmark. A thorough comparison of the VectCore approach to contemporary high performance computing alternatives in Section 6.2.3 shows the VectCore outperforms the GPP running optimized libraries and most other alternatives when targeted to FPGA technologies more recent than the current implementation. In Section 6.3, the VectCore solutions are shown to be specifically tailored to the input problem. Section 6.4 provides data demonstrating the VectCore scales well under scaled workload models [53] for a subset of the benchmark problems. Fixed workload scaling is shown to be good until limited by resource contention in the VectCore implementation.

Section 6.5 presents analysis and measurements of the overhead costs of the VectCore approach. Section 6.5.1 provides a comparison of the VectCore overhead with high performance computing alternatives. Data from end-to-end hardware experiments in Section 6.5.2 shows

Table 6.1: Test case problems with nomenclature.

Problem Description	**Nomenclature**
Matrix Multiplication	*jik_ip*, *jki_saxpy*, *kji_saxpy*
Upper Triangular Solve	*ts_ip_u*, *ts_saxpy_u*
TASS	*tass*
Modified TASS	*mtass*

a low implementation overhead for the VectCore processor. Section 6.6 examines the cost scaling of the research approach in terms of code size and compilation time. Strategies for mitigating high costs are investigated in Section 6.7. Finally, in Section 6.8, the VectCore framework is used to evaluate a custom instruction addition for the TASS problem prior to full hardware implementation.

6.1 Test Cases and Nomenclature

Performance of the benchmark problems is presented for a range of problem sizes and architecture resource limits. The nomenclature shown in Table 6.1 is used to label each problem type.

The three problem sizes used are described in Table 6.2. The nomenclature used is "pN", where N is an integer index corresponding to a problem-specific result matrix size shown in Table 6.2. For the resource limits, multiples of the maximum slices available in the Xilinx®Virtex II pro™vp70 FPGA (33088 slices) are used. The nomenclature is aN, where N is an integer multiplier of the vp70 slice limit. For example, the designation a2 denotes a limit of 66176 slices. Table 6.3 summarizes the problem sizes and architecture limits tested. All problems use the maximum vector length of 512 available in the current implementation of the VectCore design, where applicable. When architecture parameters are referred to, the nomenclature described in Section 4.4 is used.

Table 6.2: Test case workload size nomenclature.

Problem	**Nomenclature**	**Result Matrix Size**	**DFG # Ops**
jik_ip	p1	256	304
	p2	512	592
	p3	1024	1168
jki_saxpy	p1	8192	304
	p2	16384	592
	p3	32768	1168
kji_saxpy	p1	8192	304
	p2	16384	592
	p3	32768	1120
ts_ip_u	p1	4096	319
	p2	16384	639
	p3	65536	1279
ts_saxpy_u	p1	4096	256
	p2	16384	512
	p3	65536	1024
tass	p1	n/a	240
	p2	n/a	480
	p3	n/a	960

Table 6.3: Workload and architecture limit combinations for test cases.

Problem Size	**a1**	**a2**	**a3**	**a4**	**a8**	**a16**
p1	√					
p2		√				
p3			√			
p2		√		√	√	√
p1	√					
p2	√					

As mentioned in Chapter 5, all the VectCore data in this chapter with the exception of Section 6.4.3 is the scheduler output. Scheduler output does not include the overhead of either the PowerPC™-to-VectCore interface or the S-PAK dispatch. In Section 6.4.3, this overhead is characterized and shown to be low enough compared to the schedule length for the scheduler output to be an accurate basis for comparison.

6.2 VectCore Performance Comparisons

This section compares VectCore performance data to the performance of a diverse set of alternative computing approaches. Section 6.2.1 compares the VectCore to a desktop workstation implementation of the test problems. Section 6.2.2 compares the VectCore to a fixed vector implementation characterized by architecture parameters that are not tailored for the input problem. The VectCore architecture exhibits better FLOPS performance than a GPP processor for only the TASS problem, and better FLOPS performance and utilization metrics than the fixed vector implementation across all the benchmarks with the exception of the triangular solve problem. As discussed in Section 5.2.2, data dependencies, resource contention, and small average vector lengths prevent high FLOPS performance for vector implementations of the triangular-solve problem. Section 6.2.3 compares VectCore FLOPS performance and cost per performance to a range of high performance contemporary alternatives. The absolute value of the VectCore approach as compared to these alternatives is also qualitatively discussed in this section.

6.2.1 VectCore/General Purpose Processor Comparison

To compare the VectCore to a desktop GPP implementation, the matrix multiplication and triangular solve problems are implemented using MATLAB®Version 7.9.0 matrix opera-

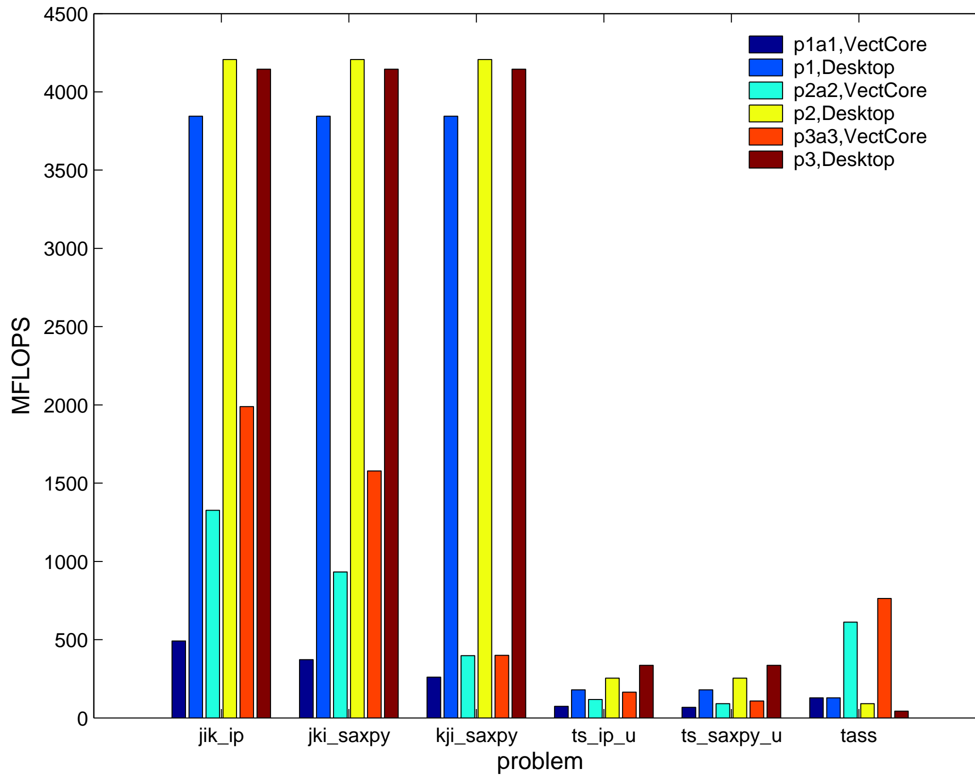

Figure 6.1: FLOPS performance for VectCore and desktop implementations. Workload and architecture limits are scaled linearly (p1a1, p2a2, p3a3).

tions. These operations use vendor BLAS [41] for their implementation [66]. The TASS desktop implementation is in C code, with default gcc [67] compiler optimizations. The desktop implementations are executed on an Intel®Core™i5 2.53 GHz processor with 3.24 GBytes of RAM. All data is pre-loaded in memory for the desktop and VectCore implementations. Section A.6 provides details of the desktop implementation code.

The desktop workstation yields higher FLOPS performance than the VectCore for all problems except TASS. This performance is shown in Figure 6.1, where it is also shown the performance difference scales with increases in workload size and resource limits. The higher VectCore TASS performance is observed despite an operating frequency difference between the desktop and VectCore of 2.53 GHz and 133 MHz, respectively.

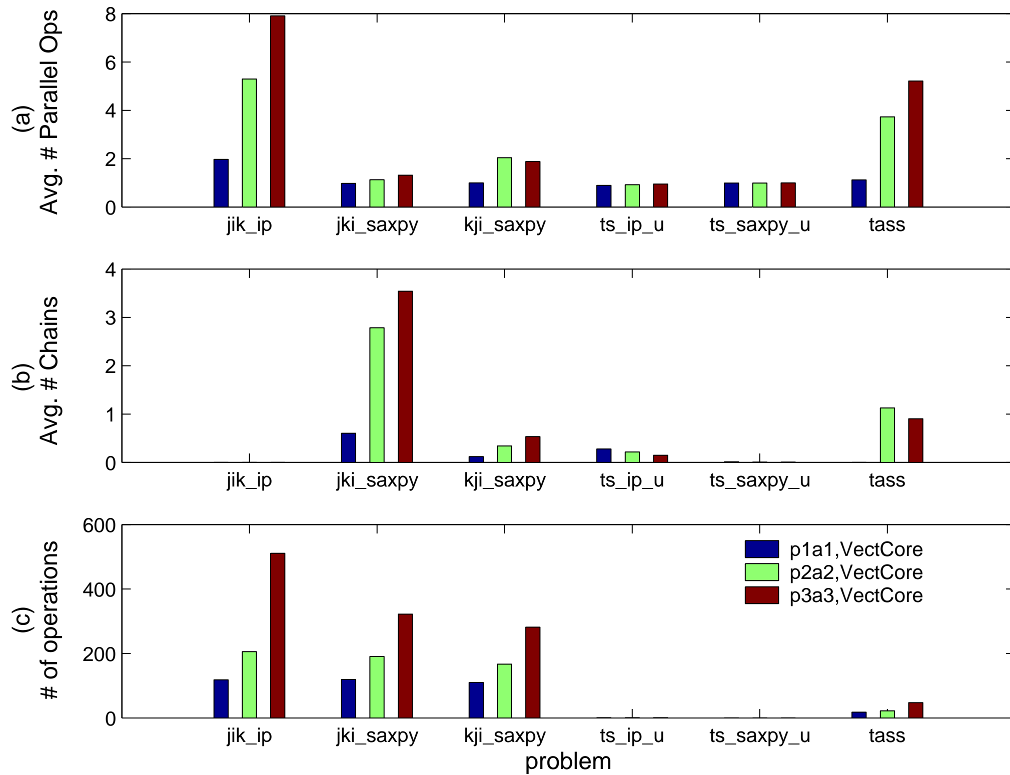

Figure 6.2: VectCore performance: (a) average number of parallel floating-point operations, (b) average number of vector chains, and (c) average ready set size. Workload and architecture limits are scaled linearly (p1a1, p2a2, p3a3).

Figure 6.1 shows variations of the performance difference across the benchmark problem types. The desktop performance difference between the matrix multiplication and triangular solves can be attributed to MATLAB®using a backsubstitution algorithm instead of BLAS for upper triangular matrices [68] as implemented in Section A.6. The low desktop TASS performance is expected because this implementation does not use an optimized vendor library. The VectCore performance differences for the matrix multiplication and *tass* benchmark problems are discussed in Section 6.2.2.

For the triangular solve problems, Figure 6.2(a,b) shows the VectCore average number of parallel and chained operations, considered in combination, is low compared to the other problems. The small amount of parallelism available in the triangular solve problems is

Table 6.4: Average benchmark problem vector lengths for the VectCore implementations, across problem sizes p1, p2, and p3.

Problem	Average Vector Length
jik_ip	478
jki_saxpy	478
kji_saxpy	488
ts_ip_u	29
ts_saxpy_u	36
tass	429

evident in the small average ready set size, shown in Figure 6.2(c). This average ready set size is less than one. Additionally, the average vector lengths for the triangular solve problems are much lower than for the other benchmark problems, as summarized in Table 6.4. Thus, the dependencies and low average vector lengths in the triangular solve problems do not enable the VectCore implementation to use the performance-enhancing capabilities of parallel and chained vector operations to the same extent used in the other benchmark problems.

6.2.2 VectCore/Fixed Vector Architecture Comparison

An examination of the FLOPS performance difference between the VectCore and fixed vector implementation (described in Section A.5) in Figure 6.3(a) shows the VectCore has higher FLOPS performance across all the problem types except the triangular solve problem. This performance difference scales as the problem size and architecture limit are scaled for the *jik_saxpy* and *jki_saxpy* problems. Figure 6.3(b) shows the percent maximum architecture FLOPS for each implementation, problem size, and architecture limit combination. The percent maximum architecture FLOPS is a measure of the architecture utilization achieved by a particular problem, and is computed assuming each floating-point unit is computing all the time. As shown, the higher VectCore utilization for the *jik_ip*, *jki_saxpy*, and *tass*

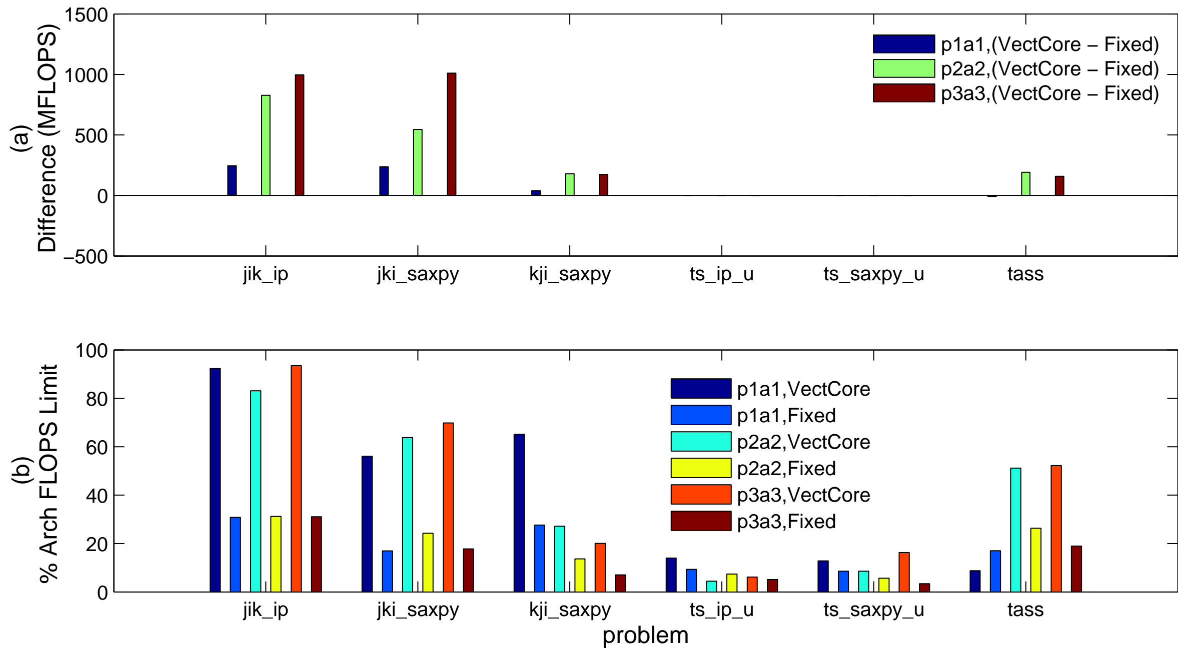

Figure 6.3: FLOPS performance delta (a) and floating-point percent utilization (b) for VectCore and fixed vector implementations. Workload and architecture limits are scaled linearly (p1a1, p2a2, p3a3).

problems indicates the VectCore implementations are more efficient then the fixed vector for these problems. This is expected, given floating-point units are included in the fixed architecture specification that are not used for all problems. For example, the *jik_ip* and *ts_ip_u* problems are the only problems containing inner product operations.

The FLOPS performance differences across the problem types are explained by comparing metrics affecting this performance as shown in Figure 6.4. Parallel vector operations (Figure 6.4(b)) and vector chains (Figure 6.4(c)) combine to directly correlate with the FLOPS performance (Figure 6.4(a)). Register spills (Figure 6.4(d)) have an inverse relationship with the FLOPS performance. The increasing number of register spills for the *kji_saxpy* fixed implementation as the problem size and architecture limit is scaled prevents parallel and chained operations and causes the FLOPS performance to remain constant under this

Table 6.5: Live value *kji_saxpy* problem requirements and register specifications for the VectCore and fixed vector implementations.

Problem Size	# of Live Values	VectCore Registers	Fixed Registers
p1	16	21	8
p2	32	43	22
p3	64	81	28

scaling. The spills are caused by an insufficient number of vector registers specified for the fixed vector implementation of the *kji_saxpy* problem. As discussed in Section 5.2.1, the *kji* ordering accumulates all partial sums of the columns of the result matrix. The accumulations must be completed before storing each of those columns, which increases the number and duration of live values, increasing the register requirements of this problem. Table 6.5 shows the number of live values maintained prior to storing one result matrix column and the corresponding number of vector registers in the VectCore and fixed vector implementations. As shown, the VectCore optimization specifies a sufficient quantity of registers, but the fixed vector has too few registers defined to maintain the *kji_saxpy* live values without spilling.

Additional evidence supporting the correlation between the amount of parallel vector and chained vector operations and FLOPS performance is provided in Figure 6.5(a) with a comparison of the measured VectCore FLOPS to FLOPS predicted with Equation 5.6. The performance model generally agrees and trends similarly with the measured FLOPS. As described in Section 5.3, spills are not included in the model. Consequently, the model over-predicts the performance for cases with many register spills. Another over-prediction discrepancy is shown in Figure 6.5(a) for the *jki_saxpy* and *kji_saxpy* p3a3 cases. This is due to the lack of bandwidth effects in the prediction model. For example, the VectCore specification for the *kji_saxpy* p3a3 problem has sixteen load/store units, but the store operations at the end of the schedule execute serially as shown in Figure 6.5(b). The performance effects of the serialization of the store operations is not captured in the first-order prediction

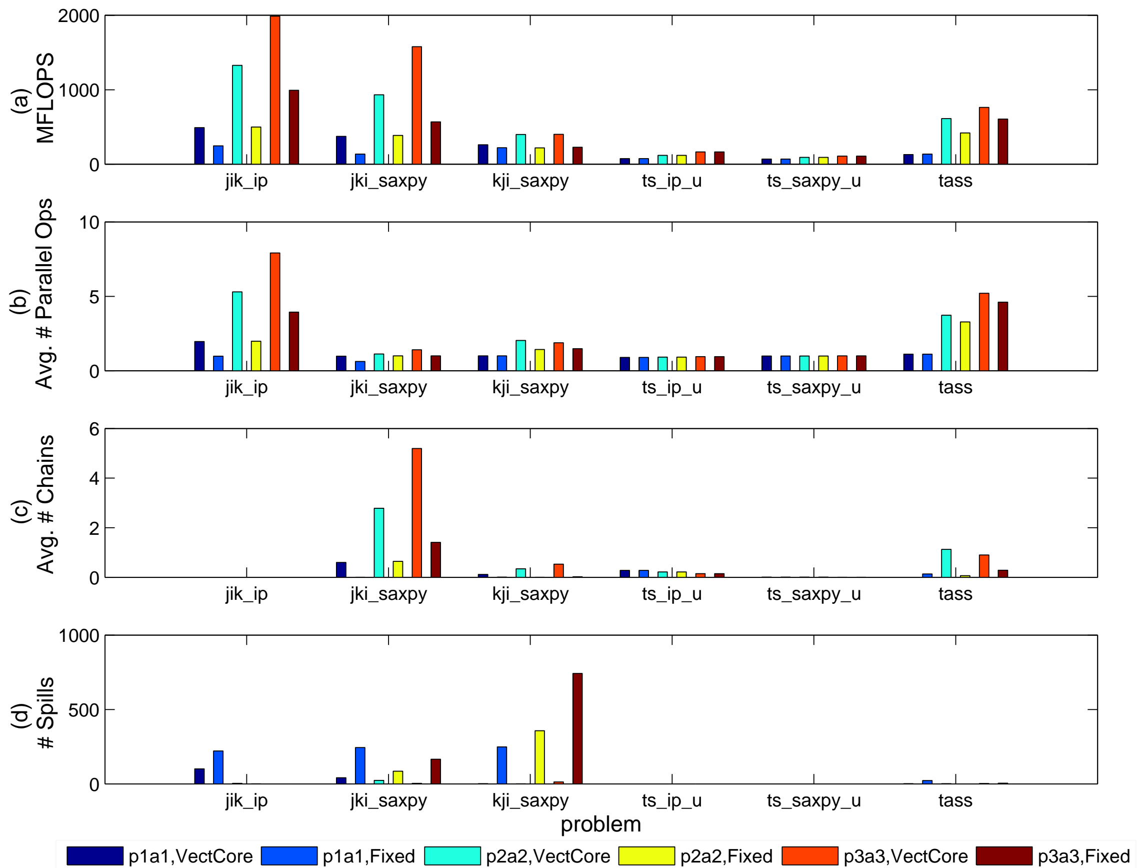

Figure 6.4: VectCore to fixed vector performance comparison: (a) FLOPS performance, (b) average number of parallel floating-point operations, (c) average number of vector chains, and (d) number of spills. Workload and architecture limits are scaled linearly (p1a1, p2a2, p3a3).

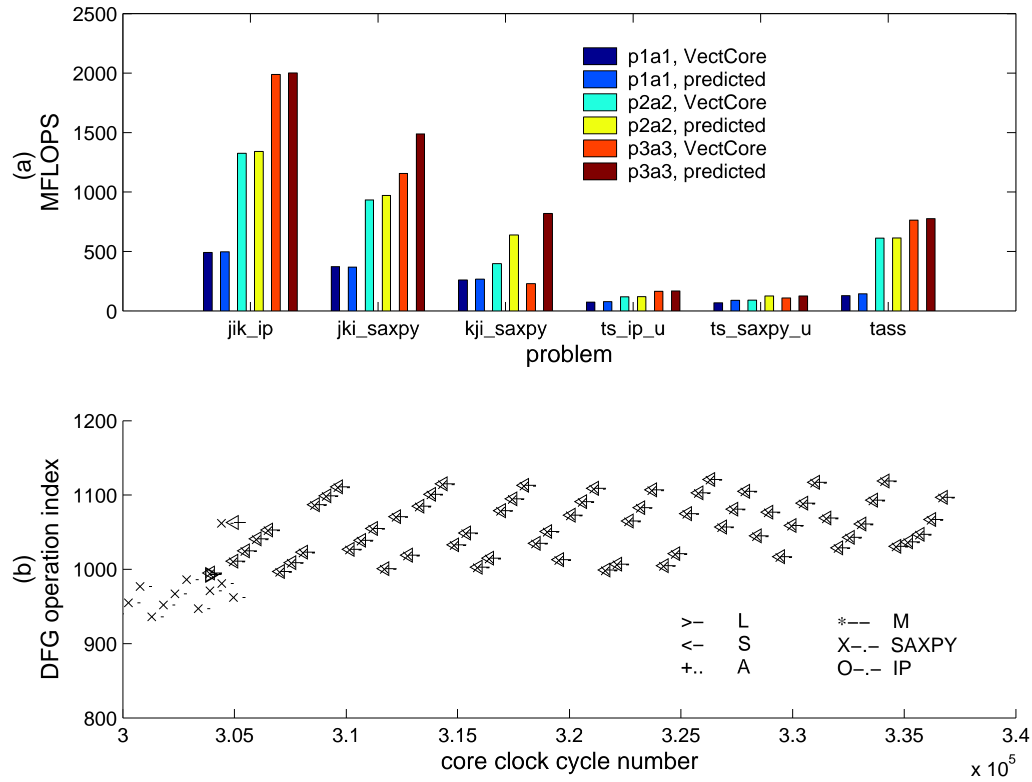

Figure 6.5: (a) Comparison of measured VectCore FLOPS performance to predicted FLOPS performance, and (b), portion of schedule for kji_saxpy p3a3 problem.

model.

Serial execution is caused by conflicts reading a register holding the same column of one of the operand matrices for each $SAXPY$ operations supplying data for each store. This conflict is an implementation-dependent and problem structure-dependent effect, causing lower performance on this problem compared to the other matrix multiplication implementations. Similar effects are observed in fixed workload scaling described in Section 6.4.1.

6.2.3 Comparison of VectCore to High Performance Processing Alternatives

In this section, the VectCore performance is compared to alternative approaches for processing computations in the scientific simulation domain. These alternatives are described in Chapter 2, and include hybrid general-purpose/configurable processor HPC systems, direct algorithm implementation in configurable hardware, configurable vector processors, server and desktop-grade GPP systems, GPU systems, and a traditional supercomputer.

Hybrid configurable HPCs employ FPGAs as computation accelerators in an overall system. For example, the Silicon Graphics (SGI) RASC combines a dual FPGA board with an Altix server [17]. The FPGA board is connected to the Altix memory system through a proprietary interconnect, allowing for simultaneous execution of algorithms partitioned between the FPGAs and the server, and facilitates scaling to multiple FPGA boards. The system is programmed using a choice of C-based high-level language systems that are integrated with hardware synthesis tools. Hybrid HPC systems provide a custom-designed support system for configurable processing resources that allows more of the configurable resources to be applied to floating-point processing. A legacy code may need to be rewritten to use the programming resources available for a particular system. Hybrid systems can cost substantially more than GPP processing systems [16],[17],[18],[69] and single FPGA boards [70],[71], and may not be as easy to upgrade to new FPGA families if changes to the fixed architecture are required.

Direct hardware implementations such as systolic arrays efficiently use the configurable resources in a data-driven hardware realization of a particular computation. These implementations can take advantage of all the parallelism inherent in a computation, to the limit of available resources. Hardware design expertise is required to realize the implementation, which is highly-specific to a particular computation. Although the target device can

be configured to another problem's direct implementation, the substantial design effort to translate a computation to an efficient hardware realization must be performed for each problem. Consequently, systolic array implementations only exist for problems in which this design effort has been invested. The implementation does not execute any software code, so the development effort associated with a legacy code cannot be leveraged by this approach.

Configurable vector processors such as the VectCore and VESPA processors trade specificity of the hardware design to particular application for a more general applicability to the class of vector operations. The parametric designs in both these examples can still be tailored to a particular vector computation, which increases performance and more efficiently utilizes resources than a fixed processor design as shown in Section 6.2.2. This layering of a configurable processor on the architecture of a configurable device such as an FPGA has inherent overhead, which is characterized in Section 6.5.

GPP and GPU systems have the advantage of optimized circuitry implemented directly in the silicon. Processors in these systems are capable of much higher clock speeds than the general circuitry of an FPGA. Wide applicability and high numbers of units produced make these processors the most competitive in price. Application code must be written to use optimized libraries or be suitable for compilation to the system, so a legacy application may still require a re-hosting effort to target these systems.

Finally, a traditional supercomputer represents a design optimized for a class of computation types. They typically have the highest performance (at the time of their introduction) executing the computations for which they are designed. Except for scaling components in parallel systems these designs are not flexible and have the highest entry cost. Upgrades consist of new system releases. Application codes typically have to be written carefully to be complied for high performance execution on these systems.

Table 6.6: VectCore performance and price per performance comparison to implementation alternatives.

System	Workload	Size	Clock (MHz)	FPGA	Device	GFLOPS per proc.	System Price	Price per GFLOPS
Hybrid HPC								
Convey HC-1 [18]	Matrix Mult.	order 16K	150	Virtex^{TM}5	LX 330	19.0	$13K	$171
Cray XD-1 [36]	Matrix Mult.	order 16K	110	Virtex^{TM}2 Pro	vp 50	2.0	$100K [16]	$8.3K
SGI RASC [72]	MatrixVect. Mult.	order 2K	100	Virtex^{TM}4	LX 200	1.9	$40K [17]	$5.2K
Direct Hardware Implementation								
Systolic Array [32]	2D Cavity Flow	24 X 24 grid	60	Stratix®2	EP 2S180	11.5	$7.7K [73]	$665
Systolic Array [74]	2D Cavity Flow	48 X 48 grid	106	Stratix®2	EP 2S180	18.0	$7.7K [73]	$425
Configurable Vector Processor								
VESPA [47]	autocorrelation	512	50	Stratix®	EP 1S80	1.6	$2.7K [75]	$1.7K
VectCore	Matrix Mult.	order 16K	133	Virtex^{TM}2 Pro	vp70	1.3	$7.5K	$5.8K
VectCore	Matrix Mult.	order 16K	350	Virtex^{TM}5	LX 330	6.5	$7.5K	$1.2K
VectCore	Matrix Mult.	order 16K	350	Virtex^{TM}6	LX 760	20.0	$8.5K	$426
GPP								
Dual Xeon 5520 (MKL) [18]	Matrix Mult.	order 16K	2260			17.5	$1.3K	$9
AMD (ACML) [36]	Matrix Mult.	order 2K	2200			3.9	$500	$128
GPU								
NVidia Tesla S1070 [18]	Matrix Mult.	order 16K	1440			94.5	$1.3K	$3
Legacy Vector Supercomputer								
Cray SV1 [76]	Matrix Mult.		300			1.0	$375K [77]	$100K

Table 6.6 shows the GFLOPS per processor performance and the system cost per GFLOPS for particular workload types and sizes for the implementation options. Also included are details of the FPGA type used in each configurable system and the system price. The results for each non-VectCore option are collected from recent literature, and because the literature spans several years, the system prices are as reported when the system was released, in U.S. dollars. The FPGA systems other than the hybrid HPCs are development boards, and may not have all the resources/interfaces to execute an entire scientific application. The single AMD microprocessor system price is assumed at $500 [14], and the Virtex^{TM}6 development board price for the VectCore is extrapolated from the Virtex^{TM}5 board price.

As shown in Table 6.6, the Virtex^{TM}2 Pro VectCore implementation is the second slowest, but it exceeds the performance of the Cray SV1, and a VectCore substitution for this type of system is the original motivation for this research. The VectCore provides this performance at approximately 17 times lower price per GFLOPS than the SV1. Updating the VectCore target to the Virtex^{TM}5 yields a better GFLOPS performance than previous generation hybrid HPCs, and an AMD GPP running optimized code. A Virtex^{TM}6 VectCore imple-

mentation outperforms all the alternatives, including the direct hardware solutions, with the exception of the GPU system. It is acknowledged that none of the other configurable options use as large an FPGA as the Virtex^{TM}6 in this comparison. The VESPA configurable vector processor is tested with image processing computations, using a 16-bit fixed-point format. The implementation of 16-bit fixed point arithmetic is considerably simpler than 32-bit single precision floating point arithmetic. The dynamic range of fixed point arithmetic is not suitable for all scientific computations. Although the VESPA processor cannot be compared directly to the VectCore, it is included for completeness due to its similarities with the VectCore in terms of configurable vector processing. The dual Xeon system is second in price per GFLOPS, and very close in performance per processor to the Convey HC-1, which is the best price per GFLOPS performer among the configurable systems.

The overall best performer in Table 6.6 is the GPU system, with both the highest GFLOPS per processor and the lowest price per GFLOPS. There are numerous examples in the literature [78],[79],[80],[81], of high performance GPU implementations for relatively simple computations such as linear algebra operations. Examples of more complex problems, with more general task graph dependencies, are not as prevalent. GPUs are designed for very specific types of processing as opposed to general computing. Although improving, the application interface is also not easy to program in every case [82],[83]. Success implementing scientific computations in GPUs requires careful code design [84],[85],[86]. VectCore is also not suited for all computation types, but its design automatically accommodates complex dependencies with mechanisms such as vector chaining and controlling the amount of local storage, as demonstrated in Section 6.4.

Similarly, the GPP systems exhibit high performance and value for the matrix multiplication problem, for which optimized libraries exist. As demonstrated in Section 6.2.1, a problem that does not map to an optimized library, such as the TASS problem, does not perform

better than even the vp70 VectCore.

A contemporary hybrid HPC system such as the Convey is the highest performer of all the configurable implementations except for the VectCore targeted to a large Virtex™ device. The Convey represents a category of systems designed around configurable resources for high performance. These systems perform well for problems that map well to the system architecture. Although "personalities" [26] as used in the Convey allow for some tailoring of the computation cores to an application, the supporting resources for those cores is fixed. Upgrades to new FPGA families may require costly hardware design updates to the supporting system. The results of Table 6.6 suggest that the VectCore can outperform hybrid HPC systems by targeting a one release more current FPGA device family, and this re-targeting is relatively simple for the VectCore.

Direct hardware implementations outperform the VectCore on similar FPGA device families. Examples in the literature ranging from basic matrix operations [87],[88] to scientific core computations [32],[74] all describe a dedicated design effort to implement a chosen computation. Implementing a different computation, even if similar to the original, can easily accrue a similar design effort. In addition, optimization to a new FPGA target may require substantial redesign. The VectCore approach is general for a class of computations, and as demonstrated in the results of Table 6.6, can be re-targeted easily to newer FPGA families to realize large performance gains without a specific optimization to the device. If optimization is desired, it can be applied to components of the VectCore approach, such as a particular functional unit, without affecting the overall framework.

6.3 VectCore Solution Specificity

The architecture specifications produced using the VectCore approach are tailored specifically for the input problem. Table 6.7 shows the FLOPS performance for each tailored problem implementation running each problem type. Three FLOPS values are shown that correspond to three problem size and resource limit combinations. For the *jik_ip*, *jki_saxpy*, and *tass* problems, the maximum FLOPS performance occurs for the implementations applied to the target problem of the implementation tailoring. Problems with a zero FLOPS result are cases where the implementation can not be applied to the problem. For example, the *jik_ip* tailoring has zero *SAXPY* units, and the *jki_saxpy*, *kji_saxpy*, and *ts_saxpy_u* problems require at least one of these units to execute. The performance difference between the implementation matched with its target and the unmatched implementation runs scales with resource limit and problem size for all but the triangular solve and *kji_saxpy* problems. The triangular solve challenges are discussed in Section 6.2.1, and *kji_saxpy* scaling performance is discussed in Section 6.4.2.

6.4 Scalability

The VectCore performance scaling under fixed and scaled workload models is examined in this section. Section 6.4.1 shows the matrix multiplication problems exhibit near-linear performance scaling as the resource limit is increased for a fixed workload until resource contention in the VectCore implementation prevents or slows further increases in performance. Scaling the workload with the resource limit also yields approximately linear performance scaling for the first two problem size/resource limit combinations on two of the three matrix multiplication problems, and varying rates of performance increase for the *tass* problem, as described in Section 6.4.2. In Section 6.4.3, data is presented on the VectCore performance

Table 6.7: VectCore performance of each tailored problem implementation for each benchmark problem. Problem size and architecture limits are p1a1, p2a2, and p3a3.

Problem	**Architecture Tailoring Target**					
	MFLOPS, p1a1					
	jik_ip	*jki_saxpy*	*kji_saxpy*	*ts_ip_u*	*ts_saxpy_u*	*tass*
jik_ip	491	0	0	246	0	0
jki_saxpy	0	372	231	0	135	0
kji_saxpy	0	222	260	0	242	0
ts_ip_u	0	0	0	74	0	0
ts_saxpy_u	0	68	68	0	68	0
tass	0	0	0	128	0	128
	MFLOPS, p2a2					
	jik_ip	*jki_saxpy*	*kji_saxpy*	*ts_ip_u*	*ts_saxpy_u*	*tass*
jik_ip	1326	0	0	2432	0	0
jki_saxpy	0	933	807	0	132	0
kji_saxpy	0	313	398	0	0	0
ts_ip_u	0	0	0	118	0	0
ts_saxpy_u	0	91	91	0	91	0
tass	0	0	0	147	0	612
	MFLOPS, p3a3					
	jik_ip	*jki_saxpy*	*kji_saxpy*	*ts_ip_u*	*ts_saxpy_u*	*tass*
jik_ip	1989	0	0	243	0	0
jki_saxpy	0	1155	496	0	0	0
kji_saxpy	0	267	400	0	0	0
ts_ip_u	0	0	0	164	0	0
ts_saxpy_u	0	108	108	0	108	0
tass	0	0	0	149	0	763

for a scaled workload on a fixed resource size that shows nearly constant performance for most benchmark problems.

6.4.1 Fixed-Load

Under fixed workload scaling, the VectCore demonstrates linear FLOPS performance scaling for the first two architecture limits in all problems except the triangular solve. Referring to Figure 6.6 (a), further increases in the architecture limit result in less than linear performance gains that plateau after architecture limit a8 for the matrix multiplication problems. As shown in Figure 6.6 (b) and (c), the plateau also occurs in the average number of parallel and chained vector operations. For a fixed workload, FLOPS performance gains are achieved by increases in parallel and chained operations. When the sum of these operations is sixteen, no further increases in parallel or chained operations are realized with increases in the architecture limit. The size p2 problem for *jik_ip* multiplies a (16 x 512) matrix by a (512 x 32) matrix to obtain a (16 x 32) result matrix. Sixteen 512-element vector inner product operations are repeated 32 times to build each column of the result matrix. Each repetition of these 16 operations reads the same 16 rows of the first operand matrix, and due to contention reading the registers holding these rows, the most operations that can occur in parallel for this problem size is sixteen. The current VectCore implementation maintains only one copy of the first operand matrix rows, and the vector registers have one read port usable by the vector functional units. The *jki_saxpy* and *kji_saxpy* implementations have a similar structural limitation of sixteen parallel operations. This limitation is problem-size dependent and is not observed under workload scaling as examined in Section 6.4.2.

The memory bandwidth of the VectCore specification determines the scaling behavior of the *tass* problem. Each *tass* loop iteration contains two independent operations using four loads total to supply input data. The number of loop iterations that can execute these operations

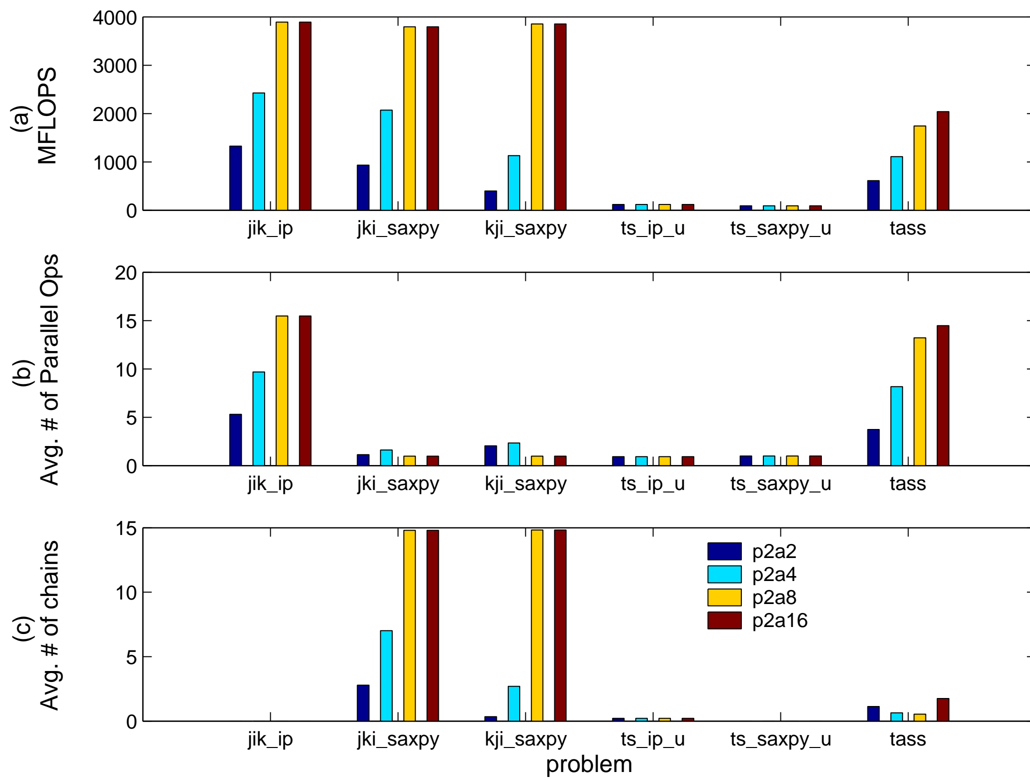

Figure 6.6: FLOPS, parallel vector, and chained vector performance across problem types for fixed workload size p2, and scaling resource limit.

in parallel is constrained by the lower result of either $U/2$ or $L/4$. Table 6.8 shows the L and U specifications for the four architecture limits used for the fixed workload scaling analysis. The supported parallel operations are the approximate parallel operations supported by each specification. For example, architecture limit a2 supports 10/4 or 2.5 concurrent loop iterations, where each loop iteration contains two potential concurrent operations. This results in approximately five parallel operations. Table 6.8 shows the supported parallel operations roughly track the observed average number of parallel operations for the first two architecture limits, and then are significantly higher than the observed parallel operations. The discrepancy is due to address contention between loads. As discussed in Section 5.2.3, there are a total of $(7/L)jk$ potential memory read conflicts. Each conflict causes an i cycle delay (equal to the vector length). For the p2 problem, $i = 512$, $j = 8$, and $k = 4$. Computing the potential number of delayed cycles using the appropriate number of load/store units for each architecture limit, and determining the percent increase in total execution cycles from the no-contention performance[1] results in the percent performance loss data. Applying this factor to the supported parallel operations for the a4, a8, and a16 cases results in adjusted parallel operations that more closely track the observations. The first architecture limit has less supported parallel operations than potential conflicts so there are few conflicts to affect performance for this case.

An evaluation of parallel and chained vector operations, and parallel independent memory operations under fixed workload scaling is shown in Tables 6.9 and 6.10. In Table 6.9, average parallel and chained operations achieved by the VectCore implementations are shown as a percentage of the maximum Data Flow Graph (DFG) level size. The maximum level size provides an approximate problem-dependent upper bound on the number of chained or parallel operations. As shown, a linear increase in the percentage of maximum DFG level

[1] The no contention execution cycles are determined from running the VectCore with address conflict checking disabled.

Table 6.8: VectCore load/store and functional unit specification-derived supported number of concurrent operations and measured average number of parallel operations for *tass* problem size p2 and scaled resource limit.

Resource Limit	**L**	**U**	**Supported Parallel Ops.**	**Average Parallel Ops.**	**Potential Address Contentions**	**Performance Reduction Percentage**	**Adjusted Parallel Ops.**
a2	10	9	5.0	3.7	25	82	-
a4	39	13	13.0	8.2	11	58	5.5
a8	51	22	22	13.2	5	39	13.4
a16	60	47	30.0	14.5	3	35	19.5

Table 6.9: VectCore combined average parallel and chained operations as a percentage of the maximum DFG level size under fixed workload scaling.

Problem	**% Maximum DFG Level Size**			
	p2a2	**p2a4**	**p2a8**	**p2a16**
jik_ip	1.0	1.9	3.0	3.0
jki_saxpy	0.8	1.8	3.3	3.3
kji_saxpy	0.5	1.0	3.3	3.3
ts_ip_u	113.4	113.4	113.4	113.4
ts_saxpy_u	100.0	100.0	100.0	100.0
tass	7.6	13.7	21.5	25.4

size is observed for the first two architecture limits, and then this rate slows and becomes constant similar to the observed FLOPS performance.

The exception to this observation is again the triangular-solve problems. These problems have a maximum DFG level size of one, so even serial execution achieves 100% for this metric. The plateau in parallel and chaining performance for the matrix multiplication and *tass* problems is explained with the same resource contention issues that limit the FLOPS performance described earlier in this section.

As described in Section 4.4.2, the memory bandwidth of a VectCore implementation is directly proportional to the number of vector load/store units in its specification. VectCore memory bandwidth performance is presented in Table 6.10 as the average number of active

Table 6.10: VectCore average number of active load/store units as a percentage of the maximum DFG load/store level size under fixed workload scaling.

Problem	% Maximum DFG LS Level Size			
	p2a2	**p2a4**	**p2a8**	**p2a16**
jik_ip	1.3	2.2	2.2	2.2
jki_saxpy	2.1	2.9	4.4	4.4
kji_saxpy	0.6	1.5	4.6	4.6
ts_ip_u	36.3	36.3	36.3	36.3
ts_saxpy_u	29.9	29.9	29.9	29.9
tass	6.0	10.7	16.6	19.5

load/store units as a percentage of the maximum number of load or store operations found on a DFG level. As shown, only the memory-bound *tass* problem exhibits near-linear scaling of this parameter that slows at architecture limit a8. The reduction in scaling is due to the same limitations in architecture bandwidth support for parallel operations discussed for the *tass* problem previously in this section.

6.4.2 Scaled Load/Resources

Under scaled workload and resource limits, the VectCore FLOPS performance increases nearly linearly for the *jik_ip*, *jki_saxpy* and *tass* problems for the first two problem size and architecture limit combinations, as shown in Figure 6.7(a). The *kji_saxpy* problem specifically shows no performance increase between the p2a3 and p3a3 problem specifications. The FLOPS performance of the triangular solve problems increases slightly with each scaling increment, due to an increasing average vector length discussed in Section 6.4.3. Figure 6.7(b) shows how closely each problem maintains an expected fixed execution time while scaling the workload size and resource limit simultaneously [53].

As discussed in Section 6.2.2, the VectCore appropriately scales the number of registers in the architecture specification for the *kji_saxpy* problem to avoid spills as shown in Figure 6.7(c)

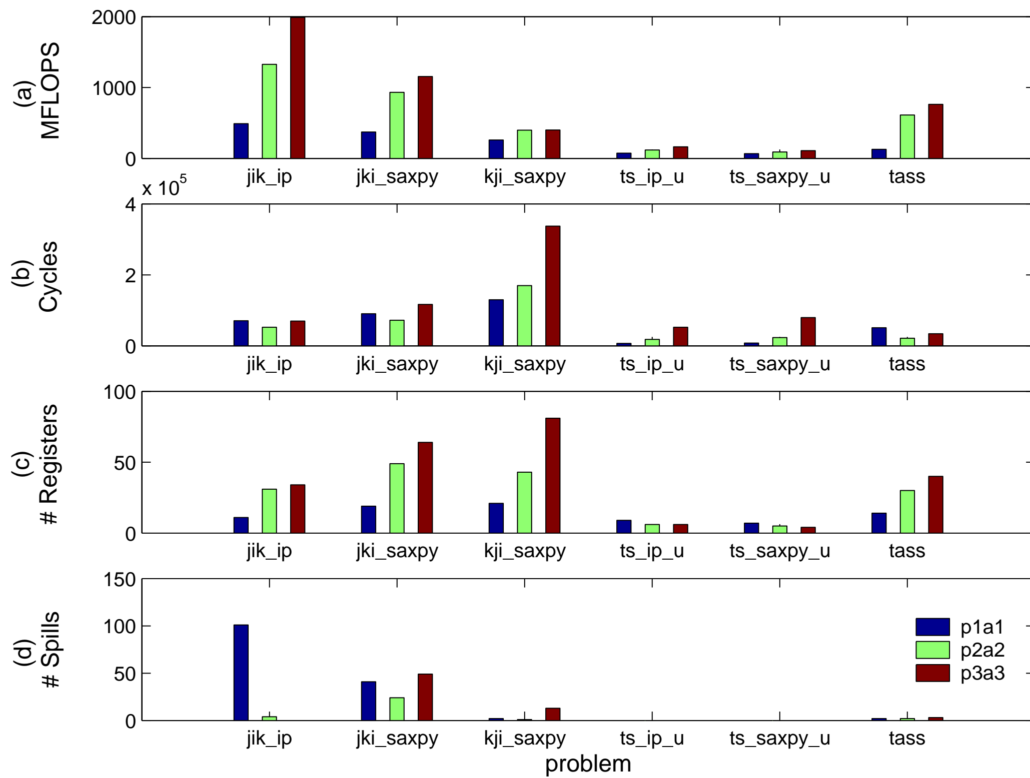

Figure 6.7: VectCore fixed-time scaling performance: (a) FLOPS performance, (b) execution time in VectCore clock cycles, (c) VectCore register specification, and (d) number of spills. Workload and architecture limits are scaled linearly (p1a1, p2a2, p3a3).

and (d). The cause for the lower *kji_saxpy* performance compared to the other matrix multiplication problems is also discussed in Section 6.2.2.

6.4.3 Fixed-Resources

As shown in Figure 6.8(a), the VectCore FLOPS performance remains nearly constant for the *jik_ip* and *kji_saxpy* problems while scaling problem size with a fixed resource limit. The *jki_saxpy* and *tass* problems exhibit performance increases with increases in the number of chaining and/or parallel operations opportunities as the problem size is increased. The average parallel operations and chains are shown in Figures 6.8(b) and (c). The increase in chained vector operations for the The *jki_saxpy* p3a1 problem is enabled by a third $SAXPY$ unit specified in the VectCore implementation for problem size p3, compared to two units specified for problem sizes p1 and p2. For the *tass* p2a1 problem, an increase in chaining opportunities is enabled by the ten interconnect buses in the specification, more than twice the p1a1 value of four.

The increases in FLOPS performance observed for the *ts_ip_u* and *ts_saxpy_u* problems in Figure 6.8(a) are due to increasing average vector length. Table 6.11 summarizes these vector lengths with the measured and predicted FLOPS increases between problem sizes p1 and p2. The FLOPS predicted increases are computed with Equation 5.6, and are in close agreement with the measurements.

6.5 VectCore Overhead

This section provides an analysis of the overhead of the research approach. The overhead elements considered are the performance cost of layering a processor design over existing FPGA resources presented in Section 6.5.1. Section 6.5.2 presents hardware measurements

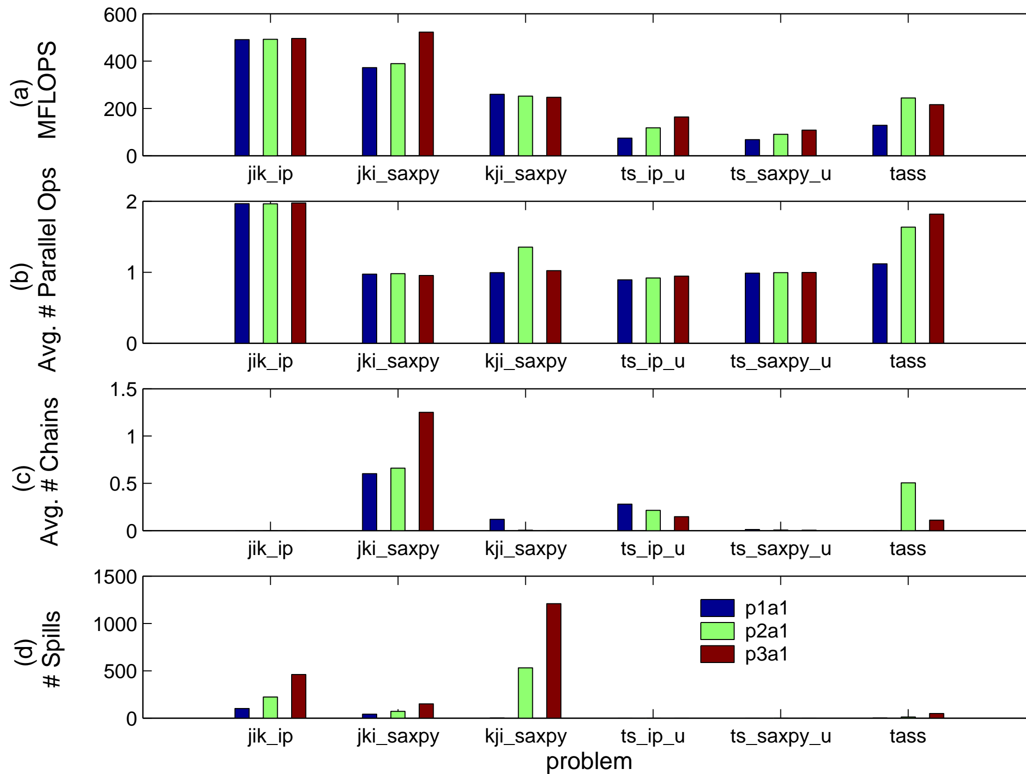

Figure 6.8: VectCore fixed-architecture scaling performance: (a) FLOPS performance, (b) average number of parallel floating-point operations, (c) average number of vector chains, and (d) number of spills. Workload is scaled linearly (p1a1, p2a1, p3a3).

Table 6.11: Average vector length for the VectCore triangular solve implementations, with measured and predicted percent FLOPS increase between problem sizes p1 and p2.

	Average Vector Length		**Measured FLOPS Percent Increase**	**Predicted FLOPS Percent Increase**
Problem	**p1**	**p2**		
ts_ip_u	12.5	25.0	58	55
ts_saxpy_u	14.8	30.8	33	40

of the VectCore interface and S-PAK dispatch scheme overhead while scaling the workload under a fixed resource limit. The percent overhead is shown to be small and nearly constant as the schedule length increases.

6.5.1 VectCore Approach

The overhead of the VectCore approach lies in layering a vector architecture on the existing architecture of an FPGA. The amount of FPGA resources needed to support the overhead instead of being used directly for floating-point computations directly impacts the FLOPS performance of the VectCore. Analysis in this section shows that an observed 60-80% resource overhead for the VectCore approach results in approximately a factor of four reduction in potential FLOPS performance compared to contemporary alternatives.

VectCore components contributing to the overhead are the global clock control system, the general functional unit bus structure, the implementation of vector floating-point units, and the S-PAK interface and distribution system. Of these, some elements are not dependent on the size of the VectCore architecture specification. As described in Section 4.6.1, the global clock control is composed of a state machine and an event FIFO, and an interface to each functional unit that only requires three bits: a ready, a hold, and a go signal. The S-PAK router is also composed of a state machine and FIFO, and its size depends on the number of distinct operations in a VectCore implementation.

The components that scale with the VectCore architecture specification size, and therefore dominate the overhead, are the vector control for each functional unit and the functional unit bus interconnect. The vector control overhead represents the resources required to provide vector control to a single-precision pipelined floating-point unit, and integrate the unit with the VectCore S-PAK dispatch scheme. The resources required for the pipelined floating point unit are not included in the overhead, as this resource cost must be present in any approach

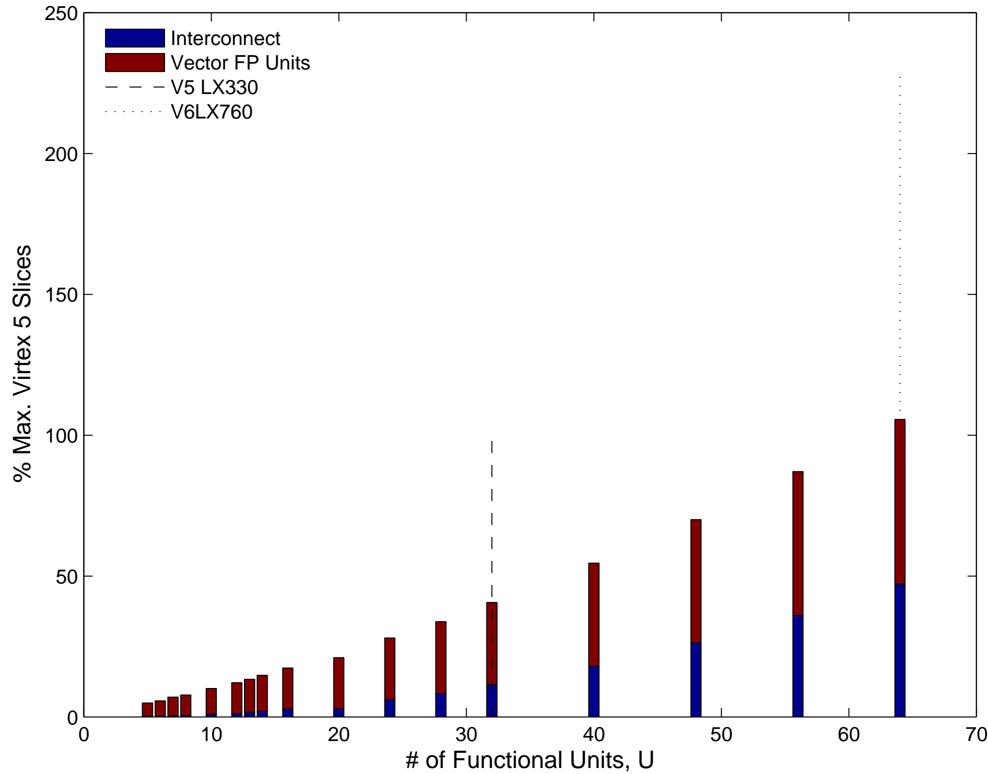

Figure 6.9: Scaling of dominant VectCore overhead components as a function of the number of floating-point units.

that computes with single-precision floating-point operations. As discussed in Section 4.6.1, the interconnect overhead is composed of multiplexers on each functional unit, register, and functional unit bus input, and the wiring between the ports of each multiplexer.

Figure 6.9 shows how these dominant overhead components generally scale as the number of floating-point units is increased in a VectCore specification. The scaling is general in that the functional unit numbers are not the result of a VectCore optimization to a particular problem. Instead, the register and interconnect bus supporting resources are scaled with the functional units, as these resources will generally increase for VectCore optimizations with increasing resource limits. The overhead is shown as a percentage of the number of slice resources in the largest Virtex^{TM}5 part, and produced with Equation 4.2 derived from

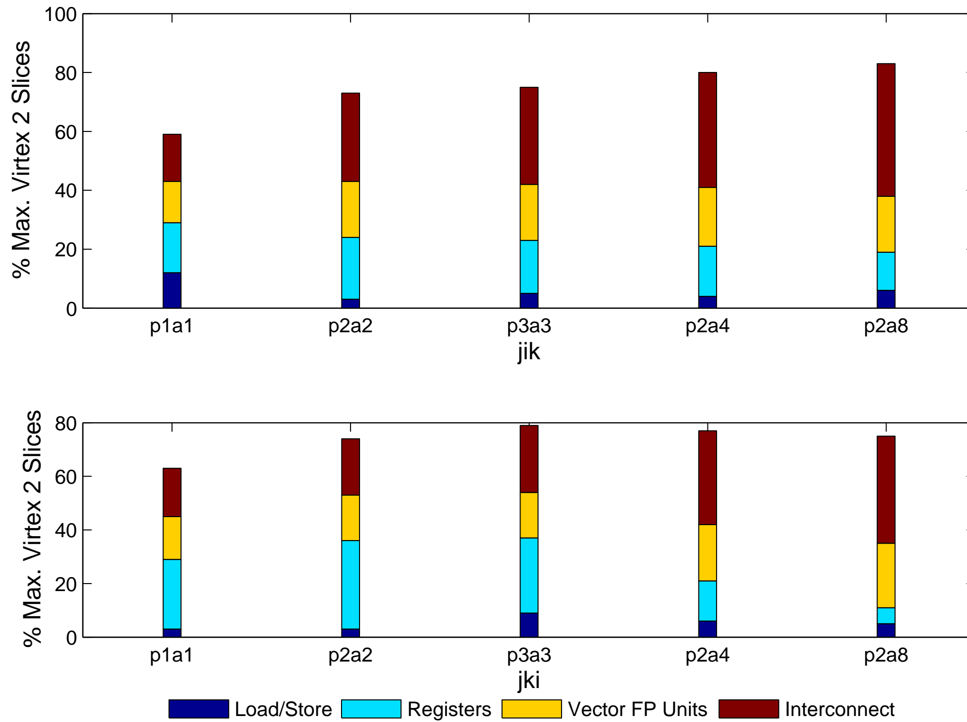

Figure 6.10: VectCore resource overhead components for the jik and jki matrix multiplication problems, with various problem size and architecture limits.

Xilinx®EDK version 12.2 synthesis runs for the VectCore design targeting a LX 330 FPGA. It is noted that the VectCore design was easily updated from EDK 8.2 targeting a Virtex™2 Pro family, without attempting to optimize the design for the features of the Virtex™5[2]. In the figure, vertical dotted lines mark the functional unit values corresponding to the largest Virtex™5 and Virtex™6 sizes[3] As shown, the per-unit overhead for vector control dominates, but the overhead due to the interconnect scheme almost equals this overhead at the Virtex™6 size.

Figure 6.10 shows in detail the scaling of all the VectCore overhead components for two of the matrix multiplication problems, over a range of problem sizes and architecture limits.

[2]New cores were generated for the floating-point operations, to take advantage of the DSP48E resources.

[3]This a conservative extrapolation, because the Virtex 6 has 6-input Lookup Table (LUT)s with twice the number of flip-flops, as opposed to 4-input LUTs in the Virtex™5, and a smaller feature size than the Virtex™5.

The problems are optimized for the Virtex™2 Pro family, and the overhead is shown as a percentage of the maximum slice limit available in the vp100 part. Again, the interconnect and vector functional unit implementation overhead dominate, particularly for larger architecture limits. Depending on the problem type and size, other overhead components can also contribute significantly to the overall resource overhead.

To characterize the performance impact of the VectCore resource overhead, the FLOPS performance achieved by instantiating the maximum number of basic floating-point operations (the operation type is problem-dependent) supported by the resources of a given FPGA is used as a reference measurement. This represents an upper performance bound for any floating-point implementation approach, because it does not account for any communications, interconnect, or memory that would be required to use the floating-point resources. This upper bound therefore represents a zero-overhead implementation.

The effect of an implementation's overhead on performance can then be measured as the difference between the performance on a given problem and this maximum theoretical performance for the same FPGA. For the VectCore, this difference is due to potential floating-point resources lost to overhead, efficiency losses in the utilization of the VectCore floating-point resources, and losses due to overhead in the GPP to VectCore interface. The interface overhead is shown in Section 6.5.2 to be only about 3-4% of the schedule length, so it is not considered a significant factor to incorporate into the following analysis.

Table 6.12 shows the FLOPS performance for the same matrix multiplication problems of Figure 6.10. Results are shown for three Xilinx®product families, the Virtex™2 Pro, Virtex™5, and Virtex™6. The architecture limits for each problem are compared to the maximum slice limits for each family in the second column. The GFLOPS performance is shown in the third column for VectCore optimized architectures. A 133 MHz clock is used for the Virtex™2 family results, and a 350 MHz clock is used for the Virtex™5 and 6 family

Table 6.12: VectCore FLOPS performance as a percentage of the theoretical upper FLOPS bound supported in a given architecture limit.

Problem	% vp100 Capacity	GFLOPS	Maximum Architecture GFLOPS	% Maximum Architecture GFLOPS	Maximum # FP Units	FP GFLOPS Limit	% Limit
jik							
p1a1	47	0.5	0.5	92	35	9.38	6
p2a2	106	1.3	1.6	83	79	21.0	8
p3a3	139	2.0	2.9	93	104	27.6	8
p2a4	191	2.4	0.5	83	142	37.8	8
p2a8	292	3.9	4.3	91	217	57.8	7
jki							
p1a1	55	3.7	0.7	56	41	10.8	3
p2a2	108	9.3	1.5	64	80	21.4	4
p3a3	166	1.6	2.3	70	124	32.9	5
p2a4	172	2.1	2.8	74	128	34.0	6
p2a8	293	3.8	5.5	70	218	57.9	7
	% V5 LX330 Capacity						
p2a1	67	6.5	10.2	64	48	33.6	19
	% V6 LX760 Capacity						
p2a1	67	6.5	10.2	64	48	33.6	19

results[4].

The fourth column in Table 6.12 is the FLOPS performance upper bound for the optimized VectCore specification, X. This result assumes all the floating-point vector units in a given specification are operating all the time and does not directly account for dependencies in the problem (but these dependencies do influence the optimization).

The fifth column reports the number of raw (non-vector) floating-point cores that would fit in the available resources of a given architecture limit. A Xilinx®CORE Generator™floating-point unit (multiply chained with an addition) is used to produce the resource cost of these units. It is assumed that built-in multiplier resources are used for the implementation, and acknowledged that more units may be possible using a logic-only implementation. The last two columns show the upper FLOPS bound achievable with an architecture filled with these raw units, and the percentage of this bound the VectCore implementation achieved,

[4]The Virtex™6 results are a Virtex™5 architecture target extrapolated to a Virtex™6 slice limit size.

Table 6.13: Comparison of percent theoretical FLOPS performance limit for the VectCore and other implementation options.

System	Workload	Size	Clock (MHz)	FPGA	Device	GFLOPS per proc.	Max. # FP Units	FP GFLOPS Limit	% GFLOPS Limit
Hybrid HPC									
Convey HC-1 [18]	Matrix Mult.	order 16K	150	Virtex™5	LX 330	19.0	48	14.4	132
Cray XD-1 [36]	Matrix Mult.	order 16K	110	Virtex™2 Pro	vp 50	2.0	40	8.8	23
SGI RASC [72]	MatrixVect. Mult.	order 2K	100	Virtex™4	LX 200	1.9	48	9.6	20
Direct Hardware Implementation									
Linear Algebra [36]	Matrix Mult.		110	Virtex™2 Pro	vp 100	4.0	74	16.3	25
Systolic Array [32]	2D Cavity Flow	24 X 24 grid	60	Stratix®2	EP 2S180	11.5	96	11.5	100
Systolic Array [74]	2D Cavity Flow	48 X 48 grid	106	Stratix®2	EP 2S180	18.0	96	20.4	88
Configurable Vector Processor									
VESPA [47]	autocorrelation	512	50	Stratix®	EP 1S80	1.6	22	2.2	72
VectCore	Matrix Mult.	order 16K	133	Virtex™2 Pro	vp70	1.3	56	14.9	9
VectCore	Matrix Mult.	order 16K	350	Virtex™5	LX 330	6.5	124	86.8	7
VectCore	Matrix Mult.	order 16K	350	Virtex™6	LX 760	20.0	284	198.8	10

respectively. As shown, the VectCore generally has a high utilization of its floating-point units, and a maximum of 20% of the theoretical FLOPS limit is possible targeting the Virtex™6 FPGA. The percent FLOPS limit metric improves for the newer FPGA families, because the per-unit resource counts contributing to the overhead reduce as a percentage of the resources available. The Virtex™2 Pro FPGAs are built with a 0.13 micrometer process [58], which was reduced to a 65 nanometer process in the Virtex™5 [89], and a 40 nanometer process in the Virtex™6 [90]. The configurable slice resources consequently have increased in density and capability. For example, a Virtex™2 Pro slice contains two 4-input LUTs and two flip-flops. A Virtex™6 slice contains four 6-inputs LUTs and eight flip-flops. The implication of this trend is that the overhead resource cost for the VectCore approach will decrease in proportion to the available resources.

The same method to produce the results in Table 6.12 is used to compare the VectCore percent of theoretical FLOPS limit to the same metric for a selection of other contemporary implementation approaches in Table 6.13. For the Altera FPGAs, the maximum number of floating-point units is limited by the number of dedicated multipliers, so the configurable logic resource cost for each floating-point operation is not required for the comparison.

As shown, the direct hardware implementations and the Hybrid Convey machine achieve the

highest percentage of the theoretical FLOPS limit. The direct hardware implementations are designed to use every multiplier resource available in the target FPGA, so the performance is very close to the theoretical limit. The higher result for the Convey machine suggests that additional logic-only floating-point units beyond the limit of built-in multiplier resources may be present.

The VectCore achieves the lowest percentage of the FLOPS potential for the available resources. As discussed in Section 6.2.3, the other configurable systems shown in Table 6.13 use the FPGA resources either directly for computation, as in the example of the systolic arrays, or specifically for floating-point acceleration, in the case of the hybrid high performance systems. Higher percentages of the FLOPS limit are expected as direct implementations most efficiently use the FPGA resources for a specific computation. The hybrid systems represent custom high performance computer designs that incorporate configurable resources into the architecture. These systems have much more support for the FPGA resources in the overall system, such as interconnection and memory interface hardware external to the FPGA. The VectCore is currently a single-core design, and all the supporting resources for the floating-point units are incorporated into the core design, with a simple interface to a general-purpose processor. So the VectCore has more overhead per-core when compared to the hybrid-systems. But the single-core design can be updated to new FPGA families easily resulting in increased performance. The compatibility of the fixed support architecture in a hybrid HPC may not accommodate FPGA upgrades as easily.

Comparing like-sized parts in Table 6.13, such as the Convey (capped at 100% FLOPS limit) to the Virtex™5 VectCore, and the Cray XD1 to the Virtex™2 Pro VectCore, the reduction in percent maximum FLOPS limit achieved taking the VectCore approach is a factor of 7-14. Differences in design clock frequency reduce the actual FLOPS per processor reduction factor to 3-4.

Although the performance cost of taking the VectCore approach appears significant, it is noted that it is not a goal of this research to determine optimized functional unit designs for a particular FPGA architecture. Lower per-unit resource usage and higher operating frequencies are possible with device-specific optimized designs [91],[92],[93]. Optimized designs can be incorporated into the existing VectCore framework to improve performance and reduce the approach overhead. In addition, the VectCore interconnect is a straightforward, non-optimized design, and its resource usage estimates are pessimistic compared to what is possible using optimizations such as partially-connected subnetworks to reduce overall connectivity. Such an optimization is not investigated as a goal of this research.

6.5.2 VectCore Interface

The VectCore design introduces low overhead when executing a scheduler solution, and this overhead remains approximately constant as the schedule length increases. Performance data is presented in this section for the *mtass* problem for workload scaling under fixed resources. The fixed resource limit is consistent with the experimental platform described in section 5.1 (a1). The data presented in this section is a product of the full end-to-end algorithmic approach, so the architecture specification is auto-coded and synthesized to a working design on the experimental platform. The end-to-end implementation allows a characterization of the overhead of the VectCore interface and dispatch scheme, using data from hardware performance counters included in the design. For all problem sizes, the VectCore approach yields the same architecture specification (4L, 11V, 2A, 2M, 9B, 0Y, 0I).

The total overhead in VectCore clock cycles is the combination of the initial startup latency from first S-PAK read to first operation execution, and the total number of core hold cycles. Table 6.14 summarizes the hardware performance data, where it is shown the VectCore implementation introduces a very low overhead (3-4%) that remains relatively constant as

Table 6.14: MTASS hardware performance data for scaled workload and fixed resource limit (a1).

Problem	**Scheduled Cycles**	**Actual Cycles**	**Hold Cycles**	**S-PAKs per Schedule Event**	**Percent Overhead**	**Overhead/ Floating Point Ops (%)**
p1	20184	20989	610	8.6	4.0	1.7
p2	39873	41122	1054	8.7	3.1	1.3
p3	78621	80763	1949	8.8	1.1	1.1

the problem size is scaled. For all problem sizes in the table, the startup latency is 197 cycles. Also shown is the overhead as a percentage of the total floating-point operations, which is also very low and nearly constant with problem size scaling.

Core hold cycles arise from the interaction between the event dispatch latency, L_d, and the structure of the schedule. The dispatch latency of 10 clock cycles for the VectCore implementation tested in this research is the minimum no-hold time between new schedule events. If the time between two non-parallel schedule events is less than L_d, a hold of L_d minus the difference between the two schedule times will be asserted. For a regular, repeated pattern of operation start times as found in all the benchmark problems, the number of hold cycles will increase with problem size. The overhead due to core holds is therefore both implementation and problem dependent. Table 6.14 shows that the average number of S-PAKs per schedule event, which roughly determines the number of start times around each event, remains approximately constant as expected[5].

6.6 VectCore Cost Scaling

In this section the overhead costs of the VectCore approach in terms of code size and time to solution are examined. Code size is shown in Section 6.6.1 to scale approximately linearly

[5]The number of schedule events is computed by dividing the scheduled cycles by the vector length.

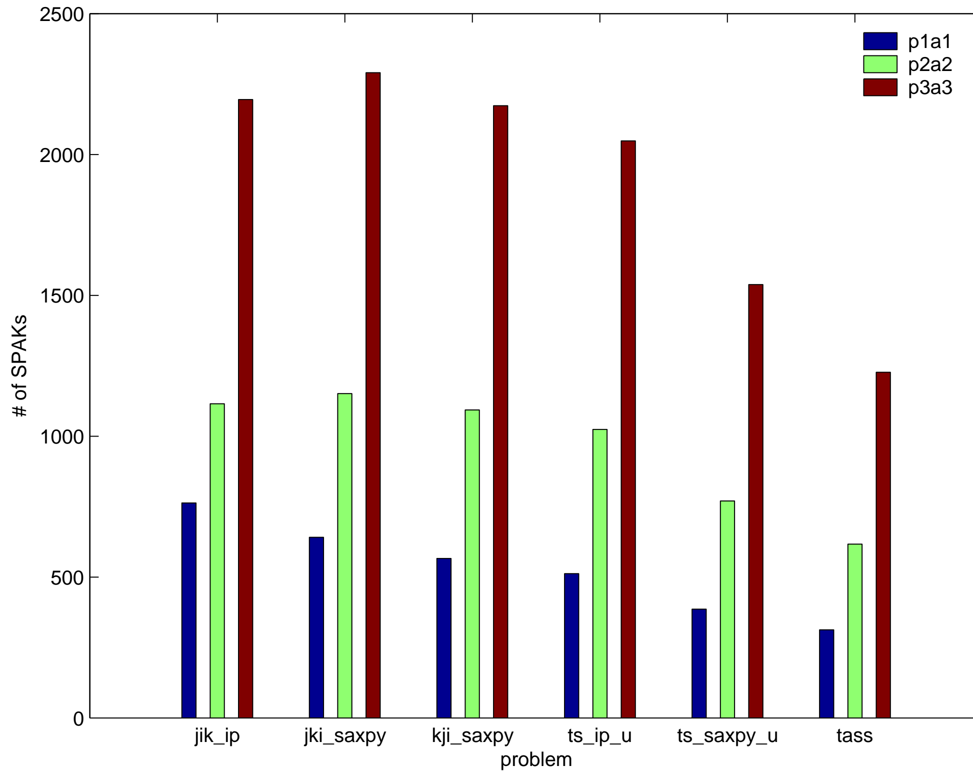

Figure 6.11: Number of S-PAKs, fixed-time and resource scaling.

with workload. Solution time is discussed in Section 6.6.2.

6.6.1 VectCore Code Size Scaling

The data in this section shows the number of S-PAKs required to execute a solution on the VectCore scales approximately linearly with the input problem size. Figure 6.11 shows the number of S-PAKs under fixed-time scaling. As shown, linear increases in problem size result in near-linear increases in the number of S-PAKs.

6.6.2 VectCore Time to Solution

The time to produce a VectCore solution is an important cost measure for the determination of the practicality of the approach. A VectCore solution is an architecture parameter

set tailored for the input problem with the corresponding schedule, and an FPGA configuration file for the VectCore instance designed from the parameter set. The parameter set and schedule are produced by the minimization approach, and the VectCore instance is produced by auto-coding design files and running platform specific synthesis tools that output the configuration file. Experience in generating VectCore instances has shown the minimization process dominates the overall time to solution. The solution time scales as $O(n^2)$ for a workload size n. The focus of this research is not on finding the most efficient implementations for components of the research approach. For example, the scheduler component is implemented in MATLAB®. MATLAB®allows for fast prototyping due to a large library of high-level functions to manipulate sets, for example. But MATLAB®is an interpreted language and has substantially more overhead than a scheduler implementation in a lower-level language, such as C. Additionally, a slow, off-the-shelf optimization algorithm is used, and the scheduling algorithm has not been optimized for execution time.

6.7 Potential Improvements

Two strategies for improving the results of the VectCore approach are presented in this section. The first addresses the floating-point utilization performance of the VectCore under fixed workload scaling. The second strategy targets long VectCore minimization times by examining a means of extrapolating VectCore solutions from small to larger problem sizes.

The implementation of the minimization approach described in Chapter 4 could be improved. As discussed in Section 6.4.1 for the matrix multiplication problems, resource contention in the VectCore processor limits the maximum number of parallel operations to 16 for the p2 problem size. For the *jik_ip* problem, the VectCore inner product unit specification is 24 for problem p2a16. This parameter is higher than the architectural limit for this case. Table 6.15 shows the utilization and FLOPS performance results of manually adjusting the VectCore

Table 6.15: FLOPS utilization and performance changes for the *jik_ip* problem after manual adjustments to the architecture specifications.

Problem	**Architecture Specification Change**	**Percent Floating-Point Utilization Change**	**MFLOPS Percent Delta**
jik p2a16	IP: 24 to 16	61 to 91	0

solutions for the *jik_ip* p2a16 problem. As shown, a simple informed adjustment to the VectCore solution increases utilization significantly, while having no affect on the FLOPS performance. This experiment demonstrates the potential for an automated step after the minimization, where each parameter is reduced until the FLOPS performance changes. This approach would not require inspection of the input problem.

Another mitigation scheme is to reduce the required number of minimization runs. An examination of the ratios of various architecture resources in the VectCore specifications under fixed-time scaling reveals trends that could be used to extend specifications for one architecture limit and problem size to larger architecture limits and sizes without additional minimization runs. Figure 6.12(a) shows approximately consistent ratios of functional unit busses to functional units under workload and resource limit scaling for the *jik_ip*, *kji_saxpy*, and *ts_saxpy_u* problems. Other ratios of interest are registers to functional units, and load/store units to functional units. Figures 6.12(b) and (c) show that only the *tass* problem maintains roughly consistent ratios for both of these parameters. The fixed-time scaling FLOPS performance of architecture specifications for the *jik_ip*, *jki_saxpy*, and *tass* problems derived from these ratios is compared with the VectCore performance in Figure 6.13. The *tass* problem shows the smallest difference between the VectCore and ratio-scaled implementations for two of three problems. This observation suggests that the ratio scaling method will only produce similar results to the VectCore method for problems that exhibit consistent ratios for all the observed parameter pairs. The specifications produced with the observed

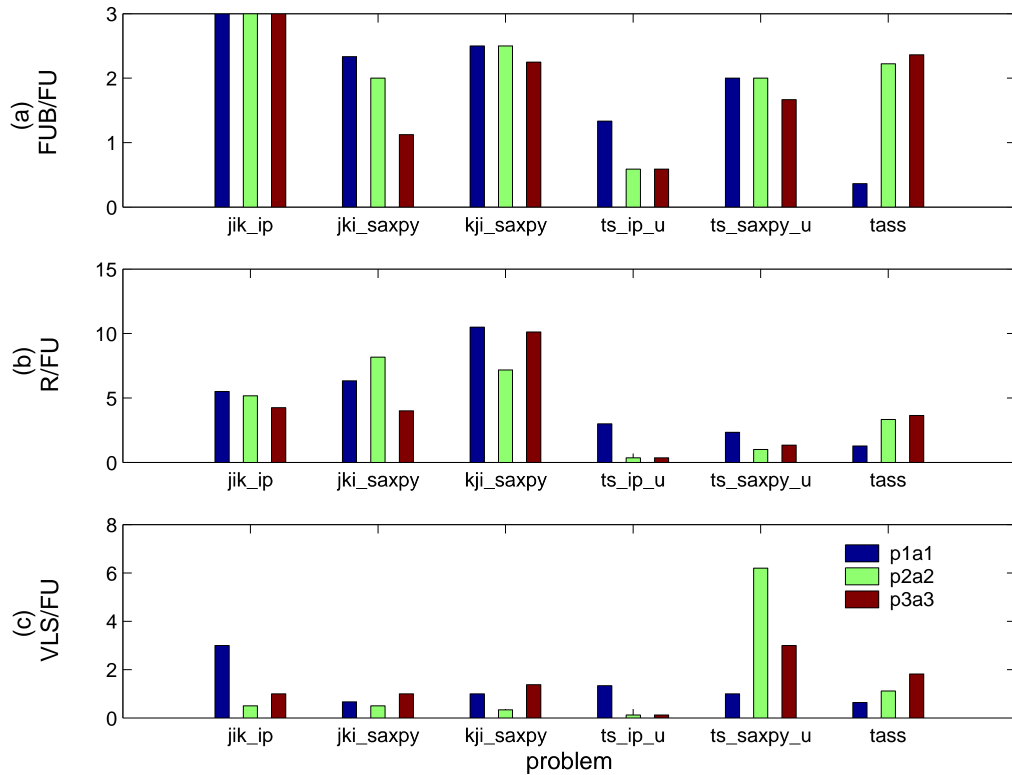

Figure 6.12: Architecture parameter ratios for each problem type under fixed-time scaling.

parameter ratios are described in more detail in Section A.5.

6.8 Custom Instruction Evaluation

The time to solution for the VectCore approach can also be evaluated with a comparison to fully-customized design approach for a particular problem. As mentioned in Section 5.2.3, the VectCore approach contributes a framework for assessing the effectiveness of adding new operations to the architecture template, prior to full implementation in hardware. Upon a favorable assessment, the additions can be made to the framework by extending the current capabilities, without an entire new design effort. This section provides an evaluation of a candidate new operation tested on the *tass* problem.

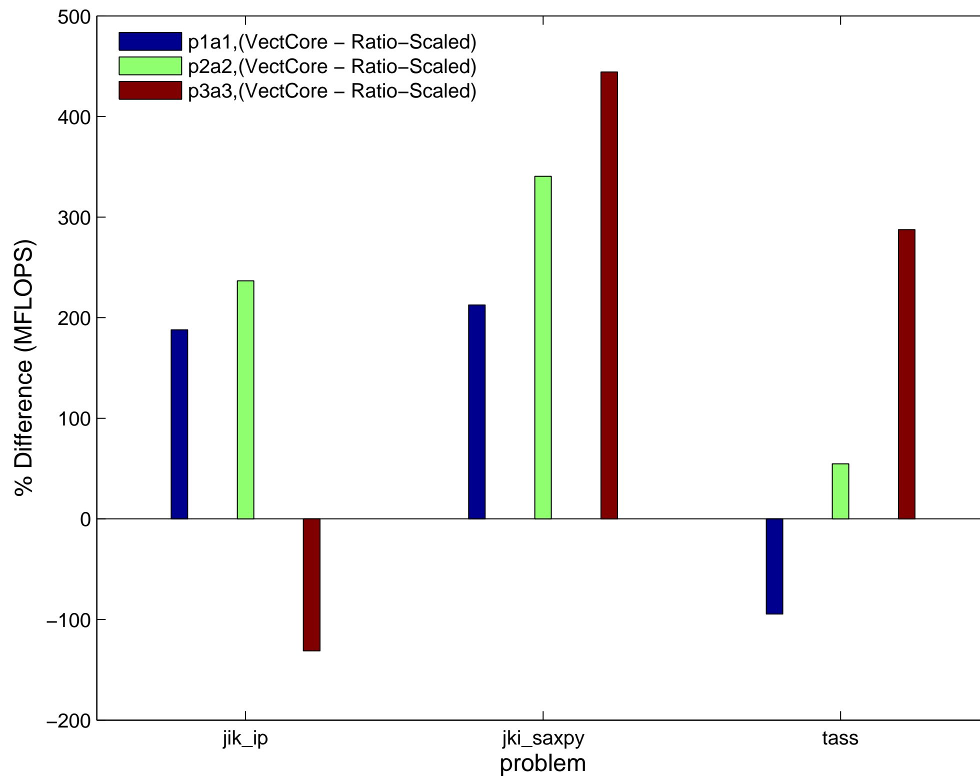

Figure 6.13: VectCore FLOPS performance comparison to parameter ratio-derived implementations under fixed-time scaling.

The new operation implements a difference of two vectors multiplied by a third vector in one 3-input, 1-output instruction, called a scaled difference. The second line in the triple-nested DO loop shown in Figure C.10 is the legacy code implementation for these operations. A scaled difference vector operation has utility in any finite-difference approximation algorithm [94], so its addition to the framework could benefit more than just the *tass* problem.

Figure 6.14(a) shows the percent difference in FLOPS performance between the implementations with three fixed-time scaling data points (p1a1, p2a2, p3a3) and two fixed workload data points (p2a2, p2a4). Despite the new operation reducing the the original *tass* number of operations by nearly 16%, only the p1a1 case exhibits better performance for the scaled difference implementation (labeled *ntass* in the figure). The relative performance differences between the implementations and problem size/architecture limit combinations are accurately predicted as shown in Figure 6.14(b) for variations in the parallel and chained vector performance shown in Figure 6.14(c) and (d). For the predicted performance, the model of Equation 5.6 is used.

The memory bandwidth limitations discussed in Section 6.4.1 affect the parallel and chained vector performance in the *tass* and *ntass* problems similarly. Table 6.16 shows a comparison of architecture parameters for each instance of the two problems, and supported parallel and chained vector operations based on the parameter values. Parallel operations are limited by the number of load/store units, L, as $L/4$ and $L/5$ for the *tass* and *ntass* problems, respectively. The *ntass* problem uses five loads to supply the data for two independent operations per loop iteration, because the scaled difference operation has three inputs and is one of the two operations.

Interconnect bandwidth limits the amount of chaining in each problem. For both problems, the minimum resources required to chain a pair of operations is $1L$, $2U$, and $5B$. The predicted maximum number of concurrent chair pairs is limited by $B/5$ for the specifications

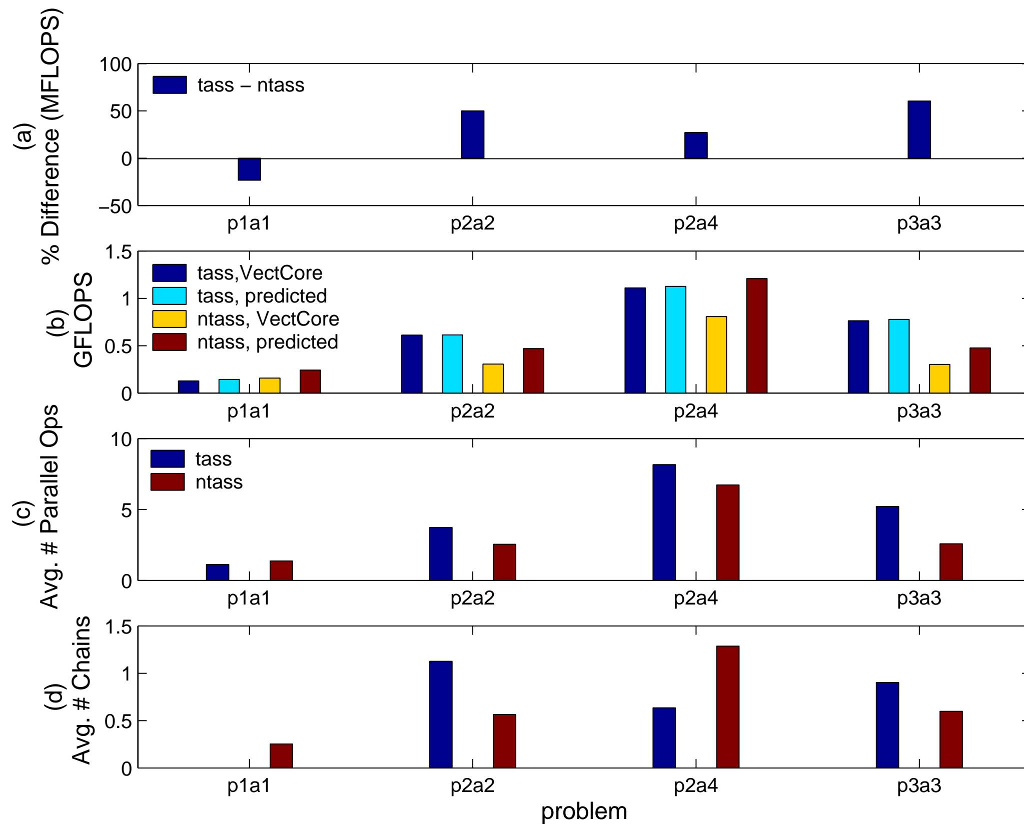

Figure 6.14: Custom *tass* instruction performance comparison: (a) percent difference in FLOPS performance, (b) predicted and measured FLOPS performance for the *tass* and *ntass* implementations, (c) average parallel operations, and (d) average chained vector operations.

tested. The differences in supported parallel and chained operations between the *tass* and *ntass* problems in Table 6.16 roughly tracks the observed performance differences.

For the problem sizes and architecture limits tested, the scaled difference operation shows no clear performance advantage over the original *tass* implementation using only vector add and multiply operations. This counter-intuitive result is determined prior to hardware implementation with the VectCore framework.

Table 6.16: Custom *tass* instruction architecture comparison. The parameters listed determine the amount of parallel and chained operations an architecture will support.

Problem	Size	L	U	B	Supported Parallel Operations	Supported Concurrent Chain Pairs
tass	p1a1	7	11	4	3.5	0.8
	p2a2	10	9	20	5.0	4.0
	p2a4	39	13	32	13.0	6.4
	p3a3	20	11	26	10.0	5.2
ntass	p1a1	5	3	8	2.0	1.6
	p2a2	10	9	21	4.0	4.2
	p2a4	19	18	40	7.6	8.0
	p3a3	18	15	30	7.2	6.0

Chapter 7

Concluding Remarks

7.1 Summary

The benefits of using configurable computing resources to implement the vector processing capabilities of a legacy application's original target platform are characterized in this work. The choice of a vector processing model constrains the solution space such that practical solutions to the underlying resource constrained scheduling problem are achieved. The vector processing model is applicable to a wide range of problems in scientific computation, and these problems often have existing implementations targeted for vector processing platforms.

A problem formulation, implementation framework, and approach are contributions of this work. The problem is formulated to allow for a tradeoff between resource cost versus performance, and the formulation is suitable for incorporation into automated optimization algorithms. The framework includes the following components: (1) a template for a parameterized, configurable vector processing core, (2) a scheduling and allocation algorithm that employs lessons learned from the mature knowledge base of vector supercomputing, and (3) the design of the VectCore co-processor to provide a low-overhead interface and control method for instances of the architectural template. The implementation approach applies the framework to produce VectCore instances tailored for specific input problems that meet

resource constraints.

The VectCore is compared with a diverse set of high performance computing alternatives. The VectCore is shown to outperform all alternatives in FLOPS performance when targeted to recent FPGA technologies, with the exception of a GPU system running optimized libraries on a problem suited for its architecture. The VectCore approach is more flexible than a GPU implementation in its ability to tailor its architecture for the dependencies in a particular problem. The VectCore is also competitive with other configurable alternatives in cost per FLOPS when targeted to recent FPGA technologies.

The overhead of the VectCore approach is characterized and compared with the overhead of the high performance alternatives. The VectCore has a high implementation overhead compared to several alternatives that can affect the FLOPS performance negatively by approximately 10%. This overhead figure is conservative in that there are optimizations not investigated in this work that could be incorporated into the VectCore implementation. In addition, the overhead cost is balanced with numerous advantages of a layered approach, such as flexibility in the problems that can be effectively executed and in performing technology upgrades. The VectCore is also designed to support maximal reuse of a legacy code, which is not supported by direct implementations or the programming interface of GPU systems.

VectCore implementations are shown to exhibit the highest performance running the problems for which they are tailored. Problems with data dependency structures that do not support parallelism such as the triangular solve do not benefit fully from the VectCore approach.

Half the benchmark cases scale nearly linearly under a fixed time scaling model. The fixed workload scaling is also linear for the same cases until becoming constant for the matrix

multiplication problems. This plateau in performance is due to resource contention in the VectCore implementation limiting the maximum achievable parallelism. Duplication of live register values is a potential means to remove this limitation, and is a consideration for future work. The architectural template contributed by this work supports established performance enhancing techniques such as parallel and chained operations. As the hardware resources are scaled, the VectCore approach scales the amount of parallelism applied in a problem implementation, with the exception of those limited by resource contention.

The VectCore is tested in an end-to-end hardware implementation for a subset of the experiment results. This provides a strong context of actual implementation details for the data. The VectCore co-processor design contributes a low overhead interface between the general purpose processor and vector processing resources and a dispatch/control scheme that controls the resources to achieve a correct implementation of a schedule. The overhead is shown to be small (less than 4%) compared to the schedule length.

This research approach is also effective as an assessment tool for new operations prior to hardware design, providing the operations can be integrated with the VectCore framework. This reduces the time to solution as compared to a completely custom design effort for a problem-specific application. Detailed performance analysis to identify the best candidates for hardware development can be accomplished with the framework. This assessment capability is demonstrated with an examination of a candidate compound instruction for the *tass* problem. Although the candidate instruction reduces the *tass* operation count by nearly 16%, it does not provide a consistent performance advantage when examined under fixed workload and fixed time scaling models.

7.2 Future Work

The VectCore approach has been tested on single matrix operations, and one nested loop. Future work should consider how the VectCore approach could be used on a set of input computations, as would be representative of an entire application. Figure 7.1 shows conceptually how the VectCore approach could be incorporated into a system that accepts multiple task graphs as its input. Individual task graphs are scheduled on the same architecture instance, and a global algorithm produces a schedule for the task graphs. Heuristics could be used to control the relative weight of a particular task graph and its influence on the architecture minimization algorithm.

Run-time partial reconfiguration could offer additional flexibility by allowing different architecture specifications, or additional types of functional units, to be available for different sections of an application. The optimization could weigh partial reconfiguration options against static architecture solutions.

The VectCore interconnect implementation significantly affects resource use and synthesis time for large architecture specifications. Large architectures would benefit from an examination of interconnection schemes that are more tailored than the VectCore network, such as a network where only subsets of the available connections necessary for a given problem are possible.

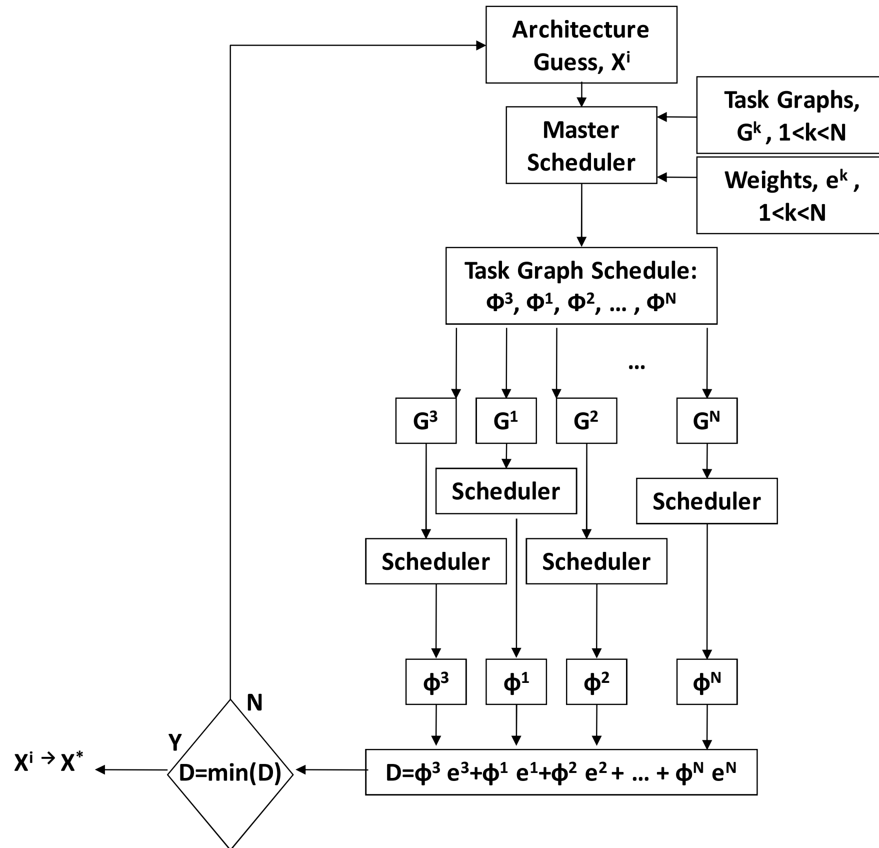

Figure 7.1: Potential VectCore approach applied to a large application. An optimization of the parameters of the VectCore processor considers N task graphs from an input application. Heuristic weights for each task graph are applied to control the affect a particular graph has on the outcome of the architecture optimization.

References

[1] H. Zima, *Supercompilers for Parallel and Vector Computers.* New York: ACM Press, 1991.

[2] "Guide to Parallel Vector Applications," Cray Manual SG-2182 3.1, Document Number 004-2182-001.

[3] G. Switzer, "Terminal Area Simulation System User's Guide, Version 7.0," Dec. 1996, research Triangle Institute Report RTI/4500/039-01F.

[4] G. Estrin, "Organization of Computer Systems: The Fixed-Plus Variable Structure Computer," *Proc. Western Joint Computer Conf., Am. Inst. Electrical Engineers*, pp. 33 –40, 1960.

[5] Xilinx, Inc., "Silicon Devices," http://www.xilinx.com/products/devices.htm, (accessed Sept. 24, 2010).

[6] D. Garey M., Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness.* San Francisco: W.H. Freeman and Company, 1979.

[7] F. H. Proctor, "The Terminal Area Simulation System. Volume I: Theoretical Formulation," *NASA Contractor Report*, vol. 4046, Apr. 1987, available from the National Technical Information Service, Springfield, VA, 22161.

[8] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, 2nd ed. San Francisco: Morgan Kaufmann Publishers, 1996.

[9] P. Athanas and H. Silverman, "Processor Reconfiguration Through Instruction-Set Metamorphosis," *Computer*, vol. 26, no. 3, pp. 11 –18, Mar. 1993.

[10] Altera, Inc., "Altera Devices," http://www.altera.com/products/devices/dev-index.jsp, (accessed Sept. 24, 2010).

[11] M. Gokhale and J. Stone, "NAPA C: compiling for a hybrid RISC/FPGA architecture," in *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, 15-17 1998, pp. 126 –135.

[12] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing Modular Hardware Accelerators in C with ROCCC 2.0," May 2010, pp. 127 –134.

[13] Celoxia, Inc., "Accelerated Trading Solutions," http://www.celoxia.com, (accessed Sept. 24, 2010).

[14] S. Craven and P. Athanas, "Examining the Viability of FPGA Supercomputing," *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, pp. 13 –13, 2007.

[15] V. Kindratenko and D. Pointer, "A Case Study in Porting a Production Scientific Supercomputing Application to a Reconfigurable Computer," Apr. 2006, pp. 13 –22.

[16] internetnews.com, "Cray Unleashes XD1 Opteron/Linux Supercomputer," http://www.internetnews.com/ent-news/article.php/3417221/Cray-Unleashes-XD1-OpteronLinux-Supercomputer.htm, (accessed Feb. 5, 2011).

[17] Government Computer News, "Reconfigurable chips could accelerate HPC," http://gcn.com/articles/2005/10/19/reconfigurable-chips-could-accelerate-hpc.aspx, (accessed Feb. 5, 2011).

[18] J. Bakos, "High-Performance Heterogeneous Computing with the Convey HC-1," *Computing in Science Engineering*, vol. 12, no. 6, pp. 80 –87, 2010.

[19] M. Wirthlin and B. Hutchings, "A Dynamic Instruction Set Computer," in *FPGAs for Custom Computing Machines, 1995. Proceedings. IEEE Symposium on*, 19-21 1995, pp. 99 –107.

[20] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler," *Computer*, vol. 33, no. 4, pp. 70 –77, Apr. 2000.

[21] E. El-Araby, I. Gonzalez, and T. El-Ghazawi, "Virtualizing and Sharing Reconfigurable Resources in High-Performance Reconfigurable Computing Systems," in *High-Performance Reconfigurable Computing Technology and Applications, 2008. HPRCTA 2008. Second International Workshop on*, 2008, pp. 1 –8.

[22] L. Braun, K. Paulsson, H. Kromer, M. Hubner, and J. Becker, "Data Path Driven Waveform-Like Reconfiguration," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, 2008, pp. 607 –610.

[23] E. G. J. Coffman, *Computer And Job-Shop Scheduling Theory.* New York: John Wiley and Sons, 1976.

[24] M. C. McFarland, A. C. Parker, and R. Camposano, "Tutorial on High-Level Synthesis," in *DAC '88: Proceedings of the 25th ACM/IEEE Design Automation Conference.* Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 330 –336.

[25] L. Oliker, A. Canning, J. Carter, J. Shalf, and S. Ethier, "Scientific Computations on Modern Parallel Vector Systems," Nov. 2004, p. 10.

[26] T. Brewer, "Instruction Set Innovations for the Convey HC-1 Computer," *Micro, IEEE*, vol. 30, no. 2, pp. 70 –79, 2010.

[27] J. Yu, C. Eagleston, C. H.-Y. Chou, M. Perreault, and G. Lemieux, "Vector Processing as a Soft Processor Accelerator," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, pp. 12:1–12:34, June 2009. [Online]. Available: http://doi.acm.org/10.1145/1534916.1534922

[28] Y. El-Kurdi, W. Gross, and D. Giannacopoulos, "Sparse Matrix-Vector Multiplication for Finite Element Method Matrices on FPGAs," Apr. 2006, pp. 293 –294.

[29] G. Wu, Y. Dou, Y. Lei, J. Zhou, M. Wang, and J. Jiang, "A Fine-grained Pipelined Implementation of the LINPACK Benchmark on FPGAs," in *FCCM '09: Proceedings of the 2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 183 –190.

[30] W. Press, W. Vettering, S. Teukolsky, and B. Flannery, *Numerical Recipes in C, The Art of Scientific Computing*, 2nd ed. New York: Cambridge University Press, 1992.

[31] G. Wu, Y. Dou, and G. D. Peterson, "Blocking LU Decomposition for FPGAs," *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, vol. 0, pp. 109 –112, 2010.

[32] K. Sano, T. Iizuka, and S. Yamamoto, "Systolic Architecture for Computational Fluid Dynamics on FPGAs," in *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, Apr. 2007, pp. 107 –116.

[33] N. Nakasato and T. Hamada, "Astrophysical Hydrodynamics Simulations on a Reconfigurable System," in *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, Apr. 2005, pp. 279 –280.

[34] K. Eguro and S. Hauck, "Resource Allocation for Coarse-Grain FPGA Development," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 24, no. 10, pp. 1572 –1581, Oct. 2005.

[35] J. O. Filho, T. Schweizer, T. Oppold, T. Kuhn, and W. Rosenstiel, "Tuning Coarse-Grained Reconfigurable Architectures Towards an Application Domain," in *Reconfigurable Computing and FPGA's, 2006. ReConFig 2006. IEEE International Conference on*, Sep. 2006, pp. 1 –7.

[36] L. Zhuo and V. Prasanna, "High-Performance Designs for Linear Algebra Operations on Reconfigurable Hardware," *Computers, IEEE Transactions on*, vol. 57, no. 8, pp. 1057 –1071, Aug. 2008.

[37] T. Baba, T. Yokota, K. Ootsu, F. Furukawa, G. Ishihara, and M. Saito, "YAWARA: A Meta-Level Optimizing Computer System," in *Innovative Architecture for Future Generation High-Performance Processors and Systems, 2004. Proceedings*, 2004, pp. 148 –153.

[38] C. Wolinski and K. Kuchcinski, "Automatic Selection of Application-Specific Reconfigurable Processor Extensions," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '08. New York, NY, USA: ACM, 2008, pp. 1214–1219. [Online]. Available: http://doi.acm.org/10.1145/1403375.1403670

[39] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn, "Solving Dense Linear Systems on Platforms With Multiple Hardware Accelerators," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009, pp. 121–130. [Online]. Available: http://doi.acm.org/10.1145/1504176.1504196

[40] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn, "SuperMatrix: a Multithreaded Runtime Scheduling System for Algorithms-By-Blocks," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 123–132. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345227

[41] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1 –17, 1990.

[42] W. Kohler, "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems," *Computers, IEEE Transactions on*, vol. C-24, no. 12, pp. 1235 –1238, Dec. 1975.

[43] B. Pangrle and D. Gajski, "Design Tools for Intelligent Silicon Compilation," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 6, no. 6, pp. 1098 –1112, Nov. 1987.

[44] A. Parker, J. Pizarro, and M. Mlinar, "MAHA: A Program for Datapath Synthesis," in *Design Automation, 1986. 23rd Conference on*, vol. 29, no. 2, 1986, pp. 461 –466.

[45] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *Computers, IEEE Transactions on*, vol. C-33, no. 11, pp. 1023 –1029, Nov. 1984.

[46] C. Lee, "Code Optimizers and Register Organizations for Vector Architectures," May 1992, PhD thesis, University of California at Berkeley, Computer Science Division (EECS).

[47] P. Yiannacouras, J. G. Steffan, and J. Rose, "VESPA: Portable, Scalable, and Flexible FPGA-Based Vector Processors," in *CASES '08: Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems.* New York, NY, USA: ACM, 2008, pp. 61 –70.

[48] D. Goodwin and D. Petkov, "Automatic Generation of Application Specific Processors," in *CASES '03: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems.* New York, NY, USA: ACM, 2003, pp. 137 –147.

[49] G. M. R. and D. S. Johnson, "Complexity Results for Multiprocessor Scheduling Under Resource Constraints," *SIAM J. Comput.*, vol. 4, pp. 397 –411, 1975.

[50] Lester Ingber, "Adaptive Simulated Annealing (ASA)," http://www.ingber.com, (accessed Sept. 24, 2010).

[51] A. B. Levy, *The Basics of Practical Optimization.* Philadelphia: Society for Industrial and Applied Mathematics, 2009.

[52] N. Shirazi, A. Walters, and P. Athanas, "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines," in *FPGAs for Custom Computing Machines, 1995. Proceedings. IEEE Symposium on*, 19-21 1995, pp. 155 –162.

[53] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* New York: McGraw-Hill, 1993.

[54] D. Kim, K. Kim, J.-Y. Kim, S.-J. Lee, and H.-J. Yoo, "Solutions for Real Chip Implementation Issues of NoC and Their Application to Memory-Centric NoC," in *Networks-on-Chip, 2007. NOCS 2007. First International Symposium on*, May 2007, pp. 30 –39.

[55] J. Nasrullah, A. Amin, W. Ahmad, Z. Qin, Z. Mushtaq, O. Javed, J. Yoon, L. Chua, D. Huang, B. Huang, M. Vichare, K. Ho, and M. Rashid, "A TeraBit/s-Throughput, SerDes-Based Interface for a Third-Generation 16 Core 32 Thread Chip-Multithreading SPARC Processor," in *VLSI Circuits, 2008 IEEE Symposium on*, 2008, pp. 200 –201.

[56] A. Majid and D. Keezer, "Stretching the Limits of FPGA SerDes for Enhanced ATE Performance," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, 2010, pp. 202 –207.

[57] K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing.* New York: McGraw-Hill, 1984.

[58] Xilinx, Inc., "Virtex-II Pro Data Sheets," http://www.xilinx.com/support/documentation/virtex-ii_pro_data_sheets.htm, (accessed Sept. 24, 2010).

[59] C. Lee and J. Smith, "A Study of Partitioned Vector Register Files," in *Supercomputing '92. Proceedings*, 16-20 1992, pp. 94 –103.

[60] W. Mangione-Smith, S. G. Abraham, and E. S. Davidson, "Vector Register Design for Polycyclic Vector Scheduling," in *ASPLOS-IV: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems.* New York, NY, USA: ACM, 1991, pp. 154 –163.

[61] J. ho Tang, E. Davidson, and J. Tong, "Polycyclic Vector Scheduling Versus Chaining on 1-Port Vector Supercomputers," in *Supercomputing '88. [Vol.1]. Proceedings.*, 14-18 1988, pp. 122 –129.

[62] J. Tang and E. S. Davidson, "An Evaluation of Cray X-MP Performance on Vectorizable Livermore FORTRAN Kernels," in *ICS '88: Proceedings of the 2nd International Conference on Supercomputing.* New York, NY, USA: ACM, 1988, pp. 510 –518.

[63] Dini,Inc., "DN6000k10S VirtexII-Pro Based ASIC Prototyping Engine," http://www.dinigroup.com/new/DN6000k10S.html, (accessed Sept. 24, 2010).

[64] Xilinx, Inc., "Platform Studio and the Embedded Development Kit (EDK)," http://www.xilinx.com/tools/platform.htm, (accessed Sept. 24, 2010).

[65] G. F. G. Dongarra, J. J. and A. Karp, "Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine," *SIAM Review 26*, pp. 91 –112, 1984.

[66] Mathworks, "Mathematics, MATLAB Version 7.0.1 (R14SP1)," http://www.mathworks.com/help/techdoc/rn/f14-998197.html, (accessed Mar. 21, 2011).

[67] GNU, "GCC, the GNU Compiler Collection," http://gcc.gnu.org/, (accessed Mar. 22, 2011).

[68] Matlab, "mldivide mrdivide /," http://www.mathworks.com/help/index.html, (accessed Mar. 23, 2011).

[69] Dell Inc., "Shop for Solutions for Your Small and Medium Business," http://www.dell.com/us/business/p/?~ck=bt, (accessed Mar. 13, 2011).

[70] Hitech Global, "Xilinx Virtex-5 LX110/LX330 PCI/PCI X Development Board," http://www.hitechglobal.com/Boards/Virtex5PCIX.htm, (accessed Mar. 13, 2011).

[71] Xilinx Inc., "Virtex-6 FPGA Broadcast Connectivity Kit," http://www.xilinx.com/products/devkits/DK-V6-BCCN-G.htm, (accessed Mar. 13, 2011).

[72] J. Koo, D. Fernandez, A. Haddad, and W. Gross, "Evaluation of a High-Level-Language Methodology for High-Performance Reconfigurable Computers," in *Application -specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, 2007, pp. 30 –35.

[73] The Dini Group, "Hardware for ASIC Prototyping & FPGA Systems," http://www.dinigroup.com/pages/3/files/2006-03-30_7000K10PCI_press_release.pdf, (accessed Mar. 13, 2011).

[74] K. Sano, L. Wang, Y. Hatsuda, and S. Yamamoto, "Scalable FPGA-Array for High-Performance and Power-Efficient Computation Based on Difference Schemes," in *High-Performance Reconfigurable Computing Technology and Applications, 2008. HPRCTA 2008. Second International Workshop on*, 2008, pp. 1 –9.

[75] Terasic Technologies Inc., "Altera DE3 Development System," http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=260, (accessed Mar. 13, 2011).

[76] Cray Inc., "The Benchmarkerś Guide for CRAY SV1 Systems," http://parallel.ksu.ru/ftp/computers/cray/sv1_bmguide.pdf, (accessed Mar. 13, 2011).

[77] IDC Inc., "The Cray CX1 Supercomputer: Leveraging the Cray Brand in the HPC Workgroup Market," http://www.cray.com/Assets/PDF/products/cx1/IDC%20whitepaper-CrayCX1.pdf, (accessed Mar. 14, 2011).

[78] G. Cummins, R. Adams, and T. Newell, "Scientific Computation Through a GPU," in *Southeastcon, 2008. IEEE*, 2008, pp. 244 –246.

[79] Y. Zhang, Y. Shalabi, R. Jain, K. Nagar, and J. Bakos, "FPGA vs. GPU for Sparse Matrix Vector Multiply," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, 2009, pp. 255 –262.

[80] Z. Cao, S. Xu, W. Xue, and W. Chen, "Improving Dense Linear Equation Solver on Hybrid CPU-GPU System," in *Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on*, 2009, pp. 556 –562.

[81] M. Taher, "Accelerating Scientific Applications Using GPU's," in *Design and Test Workshop (IDT), 2009 4th International*, 2009, pp. 1 –6.

[82] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka, "An Efficient, Model-Based CPU-GPU Heterogeneous FFT Library," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008, pp. 1 –10.

[83] S.-L. Chu and C.-C. Hsiao, "OpenCL: Make Ubiquitous Supercomputing Possible," in *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, 2010, pp. 556 –561.

[84] G. Stantchev, D. Juba, W. Dorland, and A. Varshney, "Using Graphics Processors for High-Performance Computation and Visualization of Plasma Turbulence," *Computing in Science Engineering*, vol. 11, no. 2, pp. 52 –59, 2009.

[85] K. Esler, J. Kim, L. Shulenburger, and D. Ceperley, "Fully Accelerating Quantum Monte Carlo Simulations of Real Materials on GPU Clusters," *Computing in Science Engineering*, 2010.

[86] G. Chalkidis, M. Nagasaki, and S. Miyano, "High Performance Hybrid Functional Petri Net Simulations of Biological Pathway Models on CUDA," *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 2010.

[87] S. Qasim, S. Abbasi, and B. Almashary, "A Proposed FPGA-Based Parallel Architecture for Matrix Multiplication," in *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, 30 2008.

[88] M. Karra, M. Bekakos, I. Milovanovic, and E. Milovanovic, "FPGA Implementation of a Unidirectional Systolic Array Generator for Matrix-Vector Multiplication," in *Signal*

Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on, 2007, pp. 153 –156.

[89] Xilinx, Inc., "Virtex-5 Family Overview," http://www.xilinx.com/products/virtex5/, (accessed Mar. 14, 2011).

[90] ——, "Virtex-6 Family Overview," http://www.xilinx.com/products/virtex6/, (accessed Sept. 24, 2010).

[91] A. Guntoro and M. Glesner, "High-Performance FPGA-based Floating-Point Adder With Three Inputs," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, 2008, pp. 627 –630.

[92] P. Karlstrom, A. Ehliar, and D. Liu, "High Performance, Low Latency FPGA based Floating Point Adder and Multiplier Units in a Virtex 4," in *Norchip Conference, 2006. 24th*, 2006, pp. 31 –34.

[93] K. S. Hemmert and K. D. Underwood, "Fast, Efficient Floating-Point Adders and Multipliers for FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, pp. 11:1–11:30, September 2010. [Online]. Available: http://doi.acm.org/10.1145/1839480.1839481

[94] J. D. Anderson, *Computational Fluid Dynamics: the Basics with Applications.* New York: McGraw-Hill, Inc., 1995.

[95] K. Iverson, *A Programming Language.* New York: Wiley and Sons, 1962.

[96] H. Cheng, "Vector Pipelining, Chaining, and Speed on the IBM 3090 and Cray X-MP," *Computer*, vol. 22, no. 9, pp. 31 –42, 44, 46, Sep. 1989.

[97] R. U. J. Aho, A.; Sethi, *Compilers, Principles, Techniques, and Tools.* Reading, Massachusetts: Addison-Wesley, 1986.

[98] G. Paul, "VECTRAN and the Proposed Vector/Array Extensions to ANSI FORTRAN for Scientific and Engineering Computations," *The Proceedings of the IBM Conference on Parallel Computers and Scientific Computations*, pp. 143–162.

[99] J. Allen and K. Kennedy, "PFC: A Program to Convert Fortran to Parallel Form," *The Proceedings of the IBM Conference on Parallel Computers and Scientific Computations*, pp. 186–203.

[100] K. H. Scarborough, R., "A Vectorizing Fortran Compiler," *IBM Journal of Research and Development*, vol. 30, no. 2, pp. 163–171.

[101] F. H. Proctor, "Numerical Simulation of Wake Vortices Measured During the Idaho Falls and Memphis Field Programs," *14th AIAA Applied Aerodynamic Conference*, Jun. 1996, AIAA Paper No. 96-2496.

[102] J. Holton, *An Introduction to Dynamic Meteorology.* San Diego: Academic Press Inc., 1992.

Appendix A

Implementation Details

A.1 Scheduler Example

This section describes an example of the scheduler operation, referring to the annotated data flow graph in Figure A.1 and the list of schedule events in Table A.1. In the figure, the nomenclature $_3$L1$_{D1}$ indicates a load operation, index 1, with path length 3, and one descendant. In Table A.1, the rows represent schedule events. The column labeled "Schedule Set" shows the operations scheduled at each schedule event. The architecture specification is $4L$, $6V$, $1A$, $1M$, $6B$, $0Y$, and $0I$. An n-input functional unit requires n+1 B, 1 V, and the required type of functional unit to be available before scheduling. Note the full set of resources must be available before scheduling, even if a chaining allocation allows some resources to be deallocated. Vector load/store units are also part of the resource ready checks for functional units that have a load as an input. These loads are scheduled together with the consuming functional unit. The scheduler operation on each schedule event in Table A.1 is described as follows.

- At schedule event E1, the operation M1 is the only member of the ready set, and sorted ready set (or pickset). All resources are available so M1 is scheduled, along with the loads required for its inputs. The counts of available resources are reduced by $2L$, $3V$ (one register for each load, and one result register for M1), $1M$, and $3B$ (one B for each I/O of M1).
- Operation A1 is ready at schedule event E2. The resource ready time is computed on the full set of resources required for A1 (not assuming any resources are de-allocated if A1 can chain with M1) as shown on the first E2 line of Table A.1. Before scheduling, a test occurs to see if the M1 output ready time is within a chaining threshold of the schedule time for A1. If true, A1 chains with M1, and the register previously allocated for the output of M1 is de-allocated. Also, 1 B can be de-allocated since A1 and M1

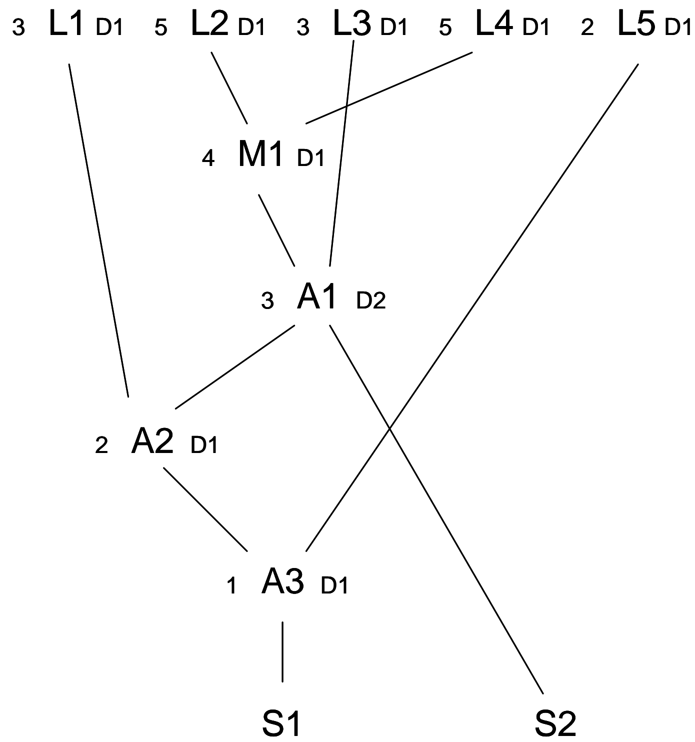

Figure A.1: Annotated data flow graph for one *tass* loop iteration.

Table A.1: Scheduler operation example.

Event	**Ready Set**	**Ready Set Sorted**	**Schedule Set**	**Available Resource Units**
E1	{M1}	{M1}	{M1 L2 L4}	$2L, 3V, 1A, 0M, 3B$
E2	{A1}	{A1}		$1L, 1V, 0A, 0M, 0B$
E2			{A1 L3}	$1L, 2V, 0A, 0M, 1B$
E3	{A2 S2}	{S2 A2}	{S2}	$0L, 2V, 0A, 0M, 1B$
E3				$4L, 5V, 1A, 1M, 6B$
E4	{A2}	{A2}	{A2 L1}	$3L, 3V, 0A, 1M, 3B$
E5	{A3}	{A3}	{ }	$4L, 4V, 1A, 1M, 6B$
E6	{A3}	{A3}	{A3 L5}	$3L, 3V, 0A, 1M, 3B$
E7	{S1}	{S1}	{S1}	$2L, 2V, 0A, 1M, 3B$

Table A.2: VectCore scheduler heuristic weights.

Heuristic	h_s	h_l	h_p	h_d	h_i
Weight	400	100	1	1	20

share a B when forming a chain. If the chaining check fails, M1 must store its output in its already allocated output register. This causes the busy times for all units in M1 path to be updated to reflect this operation, and the output ready time for M1 is updated to include the store. This example assumes A1 chains with M1. The second E2 line shows the scheduled operation set and the resource available counts are updated to reflect the de-allocation of a register and a B.

- As shown in the schedule event E3 line, the heuristic that prioritizes stores over other operations causes the store S2 to appear first in the sorted ready set. Resources support the scheduling of this store, but no add resources are available to allocate for A2, so A1 stores its vector in its output register. S2 can execute using the same register due to a second available port in the dual-port VectCore design. When S2 is complete, several resources become available for scheduling, reflected in the resource counts on the second E3 line. A similar vector store occurs at event E5. Resources support A3 and S1 scheduling on events E6 and E7, respectively.

The sorted ready set in Table A.1 is produced with Equation 4.3. The heuristic weights used in the equation are provided in Table A.2. The values are the result of many trials during the scheduler development to determine magnitudes that allow each heuristic to function as intended over a range of input problem types and sizes.

A.2 VectCore Assembly Syntax Description

This section provides a description of the VectCore assembly syntax used to define input problems for the VectCore approach. Table A.3 summarizes the mnemonics for the operations defined, and the level of implementation in the current VectCore design. In the third column of the table, “sched” indicates the operation is implemented to the scheduler output, without an S-PAK implementation or VectCore hardware support, and “h/w” indicates a full implementation in VectCore hardware is supported. Operations that lack S-PAK implementation still have an estimate included for the number of S-PAKs metric.

Figure A.2 shows an example VectCore assembly line of code with each parameter field named. A detailed description of the fields follows:

op The first field in the comma-separated list is the operation mnemonic.

Table A.3: Pseudo-assembly operation mnemonics.

Mnemonic	Description	Implementation
vl	vector load	h/w
vs	vector store	h/w
vadd	vector add	h/w
vmult	vector multiply	h/w
vadds	add scalar to vector	sched
vmults	multiply vector by scalar	sched
vlstr	vector load with stride	sched
vsstr	vector store with stride	sched
vip	vector inner product	sched
vsaxpy	vector sum of vector and vector multiplied by a scalar	sched
vscd	difference of 2 vectors multiplied by another vector	sched

dest The destination field either has a virtual register name, beginning with a “v”, for example, “$v1$”, “$v2$”, or in the case of a store operation this field has a memory address. The memory address is designated with the “*mem*” keyword and an integer address in parenthesis. For example, “*mem*(16384)”.

in1,in2, ... inN The input fields are either virtual vector registers or memory locations in the case of a load operation.

vLen The vector length field contains the desired vector length minus 1.

slrIdx The scalar index field is included for operations that either use a scalar input, such as the *vmults*, *vadds*, or produce a scalar output, as in the *vip*. This field is the index of a single element in a vector input or destination register. There are no scalar registers in the current implementation of the VectCore. But for most of the operations tested, scalar inputs or outputs are elements in a matrix column or row that are loaded or stored as a vector. For vector loads/stores with stride, this field is the integer stride.

iter The iteration field is a loop iteration number. It enables the loop iteration heuristic described in Section 4.5.1.

alias The alias register field allows a destination register name to be an alias for the register name in this field. This allows multiple operations to update the same register, where a specific order of the updates must be maintained. This is used in both implementations of the triangular solve problem. If not used, “$v0$” appears in this field.

comment The comment field extends to the end of the current line, and begins with a semicolon.

```
vip,v8,v2,v7,3,0,2,v8;(a11:a14)(x12:x42)
```

```
op,dest,in1,in2,vLen,slrIdx,iter,alias;comment
```

Figure A.2: VectCore assembly syntax example.

```
19:vmults,v12,v2,v11,3,0,3,v0;(a11:a41)x13
20:vsaxpy,u12,v5,v11,v12,3,1,3,v0;(a12:a42)x23 + (y13:y43))
21:vsaxpy,u12,v6,v11,v12,3,2,3,v0;(a13:a43)x33 + (y13:y43))
22:vsaxpy,u12,v7,v11,v12,3,3,3,v0;(a14:a44)x43 + (y13:y43))
23:vs,mem(40),u12,3,3;y13:y43
```

Figure A.3: Update register syntax example.

Another syntax used for the jki and kji matrix multiplication problems is the update register, named with a "u" instead of a "v". When multiple operations need to read the same register and update its contents, such as in a series of accumulate operations, this syntax is used. In Figure A.3, the *vmults* operation on line 19 fans out to each $SAXPY$ operation on lines 20-22. The $SAXPY$ operations can execute in any order, but each must read the same input register and then update its contents, followed by storing the final value shown on line 23. The VectCore design cannot use the same physical register as an input and an output for an operation, so the "u" syntax is used by the scheduler logic to determine all operations with the same u-register destination must be dependent on the latest member of the group to execute. So if the $SAXPY$ on line 20 executes first, then the remaining $SAXPY$ operations in the update group on lines 21 and 22 are dependent on the output of line 20. Update groups such as in this example still have the potential to chain.

A.3 S-PAK Word Format

The S-PAK is designed to facilitate a low-bandwidth general purpose processor to VectCore interface, and allow for all the functional units and load/store units specified for an architecture to execute in parallel on the same schedule event. Figure A.4 shows the bit definitions for the upper 32 bits of the 64-bit S-PAK. The word size for the current VectCore design is limited to 64 bits, the maximum width of the Virtex II pro^{TM}Processor Local Bus.

Each vector load/store operation requires 1 S-PAK for configuration, and floating-point operations such as add and multiply require 2 S-PAKs to be configured. As shown in Figure A.5, the S-PAK word definition is different depending on the operation. Indices and encoded unit or operation types are used to avoid having a dedicated bit field for each unit, making the S-PAK size architecture specification dependent.

Figure A.4: S-PAK word format, bits 63-32.

Figure A.5: S-PAK word format, bits 31-0.

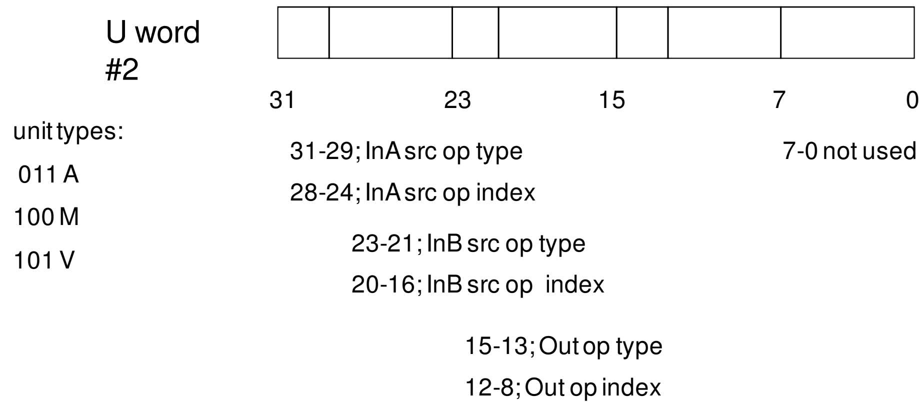

Figure A.6: S-PAK 2nd word format, bits 31-0.

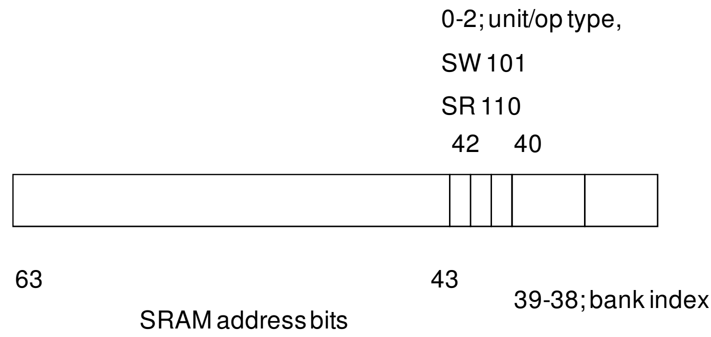

Figure A.7: S-PAK SRAM test word format, bits 63-32.

Figure A.6 shows the format for the lower 32 bits of the second S-PAK word for floating-point operations. The upper 32 bits of this second S-PAK word are spare. Figure A.7 shows an alternative S-PAK definition for a test configuration. This mode of operation is used to read or write directly from the SRAM memory on the Dini board through the VectCore load/store units. The lower 32 bits of the test word are used for data.

A.4 Constraint Function Development

The constants C_q and C_B in Equation 4.2 are determined empirically with resource measurements from synthesis tools. The slice usage is measured for a range of architecture specifications, and for each specification, C_0 and the basic per-unit slice costs are subtracted from the measured slices. The resulting slices are set equal to the equation $B[qC_q + (V + 2U)C_B]$, and the two unknowns, C_q and C_B, are determined. Figure A.8 shows the elements of the matrix equation $Ax = y$, which is solved with a least squares method to determine the constants.

A			y	x	
Bq	2U+V		slices		
20	28		-1419	C_q	-23.91
24	32		-1562	C_B	28.29
18	27		-472		
66	90		-946		
70	98		-741		
66	96		-864		
252	348		1787		
260	364		2120		
252	360		3018		
480	672		3158		
384	480		5035		
480	576		7386		
768	960		9148		
432	576		10488		
960	1152		7744		
864	1152		11015		
528	768		10673		
984	1368		15919		
1000	1400		16923		

Figure A.8: Data used to compute constraint function constants C_q and C_B.

A.5 Manually Generated Architecture Specifications

The fixed vector specifications are produced by a linear scaling of the parameters until approximately 60% slice utilization is achieved. Simple heuristics are used to establish relationships between the parameters. For example, the number of registers parameter and number of functional unit buses parameter are each increased in increments of three, because a functional unit typically requires three of each of these resources to be available before being allocated. Some trial and error is required to arrive at solutions that do not exceed the resource limit. Figure A.9(a) shows the spreadsheet used for this approach. Figure A.9(b) is the spreadsheet used to scale VectCore specifications based on parameter ratios identified in fixed-time scaling runs. The scaled specifications support a potential time-to-solution mitigation approach described in Section 6.7.

A.6 C Implementations of Benchmark Problems

In this section, listings of the C-code implementations of the benchmarks problems are provided. They are shown in Figure A.10. Note these implementations are not optimized for performance.

(a)

Fixed	Architecture Specification							SLICES	LIMIT
	L	V	A	M	B	Y	I		
	2	8	1	1	6	1	1	19742	19853
	9	22	2	2	18	2	2	39526	39706
	14	28	4	4	22	4	4	59539	59558

(b)

Problem	Architecture Specification							SLICES	LIMIT	Observed Paramter Ratios		
	L	V	A	M	B	Y	I			B/U	V/U	L/U
jik									**a1**			
Ratio-Derived	2	10	0	0	6	0	2	18299	19853	3	5	1
VectCore	4	10	0	0	6	0	2	19101	**a2**			
Ratio-Derived	5	25	0	0	15	0	5	35918	39706	3	5	1
VectCore	6	28	0	0	15	0	5	37288	**a3**			
Ratio-Derived	9	45	0	0	27	0	9	59410	59558	3	5	1
VectCore	6	38	0	0	27	0	9	55946				
jki									**a1**			
Ratio-Derived	2	14	0	1	4	1	0	17309	19853	2	7	1
VectCore	2	18	0	1	4	1	0	18601	**a2**			
Ratio-Derived	6	42	0	3	12	3	0	38821	39706	2	7	1
VectCore	3	43	0	1	11	4	0	37764	**a3**			
Ratio-Derived	8	56	0	4	16	4	0	49577	59558	2	7	1
VectCore	1	60	0	1	18	8	0	55671				
tass									**a1**			
Ratio-Derived	4	12	2	2	8	0	0	19747	19853	2	3	1
VectCore	5	11	2	2	8	0	0	19825	**a2**			
Ratio-Derived	10	30	5	5	20	0	0	39538	39706	2	3	1
VectCore	9	24	5	5	22	0	0	38083	**a3**			
Ratio-Derived	16	48	8	8	32	0	0	59329	59558	2	3	1
VectCore	17	40	7	8	35	0	0	57516				

Figure A.9: Spreadsheets for design of (a) fixed and (b) ratio-scaled VectCore implementations.

```
m=512;
n=16;
p=16;
A = rand(512,16);
B = rand(16,16);
for i=1:50
  tic;
  C = A*B;
  tElapsed(i) = toc;
end
fpops_p1 = 2*m*n*p;
disp('p1 flops');
disp(fpops_p1/mean(tElapsed));
```

(Matrix Multiplication, p1)

```
…
m=512;
n=16;
p=32;
A = rand(512,16);
B = rand(16,32);
for i=1:50
  tic;
  C = A*B;
  tElapsed(i) = toc;
end
fpops_p2 = 2*m*n*p;
disp('p2 flops');
disp(fpops_p2/mean(tElapsed));
```

(Matrix Multiplication, p2)

```
m=512;
n=16;
p=64;
A = rand(512,16);
B = rand(16,64);
for i=1:50
  tic;
  C = A*B;
  tElapsed(i) = toc;
end
fpops_p3 = 2*m*n*p;
disp('p3 flops');
disp(fpops_p3/mean(tElapsed));
```

(Matrix Multiplication, p3)

```
%p1
n = 64;
%p2
%n = 128;
%p3
%n = 256;
y = rand(n,1);
u = rand(n,n);
ut = triu(u);
for i=1:50
  tic;
  x = ut \ y;
  tElapsed(i) = toc;
end
fpops = 0;
for i=(n-1):-1:0
    fpops = fpops + 1;
    for j=(i+1):(n-1)
        fpops = fpops + 2;
    end
end
disp('p1 fp ops');
disp(fpops);
disp('p1 flops');
disp(fpops/mean(tElapsed));
```

(Tri-Solve)

```
…
static float u[M][N][P];
static float u4[M][N][P];
static float a[M][N][P];
static float p[M][N][P];
static float e[M][N][P];

static float als;
static float bts;
…
als = 5.0;
bts = 10.0;
rdtsc(low1,high1);
for(k=0;k<P;k++) {
  for(j=0;j<N;j++) {
    for(i=1;i<M;i++) {
      x1 = u[i][j][k];
      u[i][j][k] = (p[i][j][k]-p[i-
1][j][k])*a[i][j][k];
      e[i][j][k] =
als*u[i][j][k]+bts*x1+u4[i][j][k];
    }
  }
}
rdtsc(low2,high2);
```

(TASS)

Figure A.10: C-code listings for benchmark implementations.

A.7 Global Clock Control

This section provides additional details of the global clock control logic. A state diagram of the global clock control process is shown in Figure A.11. As shown in the figure, there are a set of initialization states that are used the first time the process runs after a hardware reset signal is asserted, and a set of run states that control to operation all other times. The two sets of states are needed because there is a two event pipeline used to hold the next event time and resource configuration word, and the pipeline must be filled to begin normal operation. The READFIFO state updates the next event start time and configuration. In the POLLRES state, the event time is compared to the current global clock count and the configuration word from the event FIFO is compared to the word built from the resource ready signals. If the global clock is equal to the next event start time - 1, and the configuration words match, a "GO" signal to the resources is asserted. The control progresses to the STARTRES state, where it stays until all resource ready signals have been reset, indicating those resources have begun operation.

There are two conditions causing the global clock hold and core stall. The first is evaluated in the POLLRES state, where the global clock is equal to the next event start time, but the resources assigned to the event are not ready. The second condition is associated with the global clock control implementation. If the global clock has reached the next schedule event, but the control state machine has not cycled through its states to the next POLLRES state, then the clock is held and core stalled until the control state machine reaches the POLLRES state. This implies a limitation on how close together in time two schedule events can be without causing a hold. Once the "GO" signal has been sent for the first scheduled event, the global clock is incremented every core clock cycle except under the hold conditions just described.

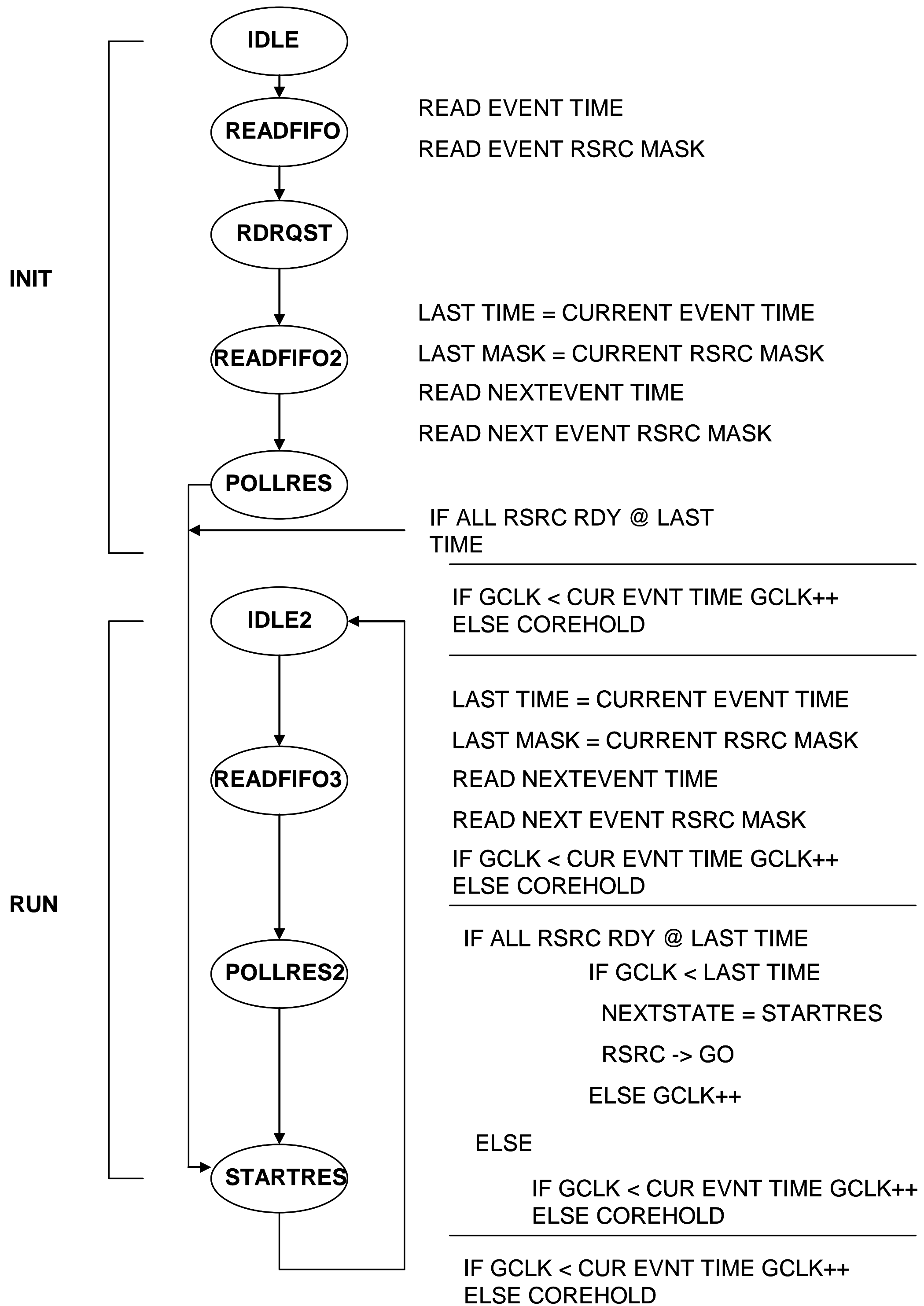

Figure A.11: Global clock control finite state machine description.

Appendix B

Verification

In this chapter, test matrices for both VectCore hardware capability testing and scheduler verification of the benchmark problems are provided.

B.1 VectCore Allocation

All basic VectCore configurations are verified with end-to-end testing of the research approach. Figure B.1 shows the test matrix and architecture specifications used. As shown, each hardware configuration supporting parallel chained vector operations is tested, as well as all the main logic paths through the scheduler.

B.2 Scheduler Operation

The benchmark problems described in Chapter 5 are verified using scheduler output and the test matrix shown in Figure B.2. As shown, the problems are verified for correctness in terms of execution order and no resource contention. All possible chained and parallel operations are verified as scheduled. In addition, all metrics reported by the scheduler are verified for accuracy.

		Arch capabilities							Scheduler logic paths						
		direct chain	L -> FU flex chain	FU -> S flex chain	temp result to reg	fanout >1 direct chain	fanout>1 reg	comb fanout >1 direct and reg	Both in loads	InA load	InB load	direct chain	no chain	DLY to match inputs	DLY to match resources
case #	**arch (L,V,A,M,B)**														
1	4,4,1,2,5		x	x					x				x		
2	4,5,1,1,6	x	x	x					x		x	x			
3	4,5,1,1,6	x	x	x					x	x		x			
4	4,5,1,1,6		x	x	x				x		x		x		
5	4,4,1,2,5		x	x	x		x		x						
6	4,8,2,2,12	x	x	x		x			x			x			
7	4,8,2,2,12	x	x	x	x	x	x	x	x			x	x		

Figure B.1: Test coverage of VectCore allocation capabilities and scheduler logic paths.

4x4 prob, arch1,2	**jik**	**jki**	**kji**	**ts_ip**	**ts_saxpy**	**tass**
execution order correct	y	y	y	y	y	y
no resource conflicts	y	y	y	y	y	y
resource reservations correct						
regs for all loads/stores	y	y	y	y	y	y
regs only for non-chained results	y	y	y	y	y	y
FUBs except for chained inputs	y	y	y	y	y	y
chain timing correct	n/a	y	y	y	y	y
non-chain timing correct	y	y	y	****	****	y
no conflicts btw LS and FU r/w	y	y	y	y	y	y
correct live reg durations	y	y	y	y	y	y
spill logic execution correct				n/a	n/a	
metric correct	y	y	y	n/a	n/a	y
same LS reserved for S,L	y	y	y	n/a	n/a	y
VLS reserved for S,L	y	y	y	n/a	n/a	y
reg reserved for L	y	y	y	n/a	n/a	y
check DFG one case	y	y	y	n/a	n/a	y
no unecessary no-op gaps	y	y	y	****	****	y
avail. chaining performed						
valid chains taken	n/a	y	y	y	y	y
no chains when invalid	y	y	y	y	y	y
avail. parallel ops performed	y	y	y	y	y	y
metrics correct and accurate	y	y	y	y	y	y

Figure B.2: Test coverage for benchmark problem scheduler verification.

Appendix C

Cray Performance Benchmarks

Benchmark data is necessary to compare the performance of custom computing implementations of supercomputer vector architectures to actual supercomputer implementations. Although it is not anticipated the custom machines will always match the performance of the supercomputers, it is useful to have quantitative data on the performance comparisons. Various performance data was collected on the TASS simulation running on a Cray SV1 platform. The data was collected using the Cray Hardware Performance Monitor (HPM), a utility that collects and displays hardware operational parameters during program execution. Hardware resources (counters) in the machine collect the performance parameters. Table C.1 shows the four hardware counter groups and parameters monitored by each of the counters in a group for the Cray SV1 computer.

The next section provides details of the types of information output by the HPM, and the HPM reports for various benchmarks from the TASS 7.0 code.

C.1 Cray Hardware Performance Monitor description

Various reports can be issued using the HPM [2]. The reports correspond to each of the counter groups shown in Table C.1, and are as follows: 1) program execution summary (Group 0), 2) hold-issue conditions (Group 1), 3) memory activity (Group 2), and 4) vector events and instruction summary (Group 3). The program execution summary report includes statistics on FLOPS, Millions of Instructions Per Second (MIPS), average clock cycles per instructions, memory references, and others. The hold-issue report includes details on the causes of various hold-issue conditions for resources such as functional units, registers, and memory. The memory activity report provides a detailed profile of memory activity and conflicts. Finally, the instruction summary shows the number of instructions issued, grouped by type. HPM reports these statistics on entire programs, so the benchmarks described in the following subsections had to be coded as standalone programs, even though two of the three

Table C.1: Cray SV1 system counters descriptions. [2]

Counter Group	Counter Number	Description
0	0	Instructions issued
	1	Clock periods holding issue
	2	Instruction buffer fetches
	3	Floating-point add operations
	4	Floating-point multiply operations
	5	Floating-point reciprocal operations
	6	CPU port memory references
	7	Cache hits
1	0	Waiting on A registers
	1	Waiting on S registers
	2	Waiting on V registers
	3	Waiting on B, T registers
	4	Waiting on Vector Functional Units
	5	Shared registers
	6	Waiting on memory ports
	7	Waiting on miscellaneous
2	0	Instruction fetches
	1	Cache hits
	2	Scalar memory writes
	3	B, T memory references
	4	Scalar memory references
	5	CPU memory writes
	6	CPU memory references
	7	CPU memory conflicts
3	0	000-017 instructions
	1	020-077 instructions
	2	100-137 instructions
	3	140-157 and 175 instructions
	4	160-174 instructions
	5	176 and 177 instructions
	6	Vector integer/logical operations
	7	Vector floating-point operations

```
Group 0:  CPU seconds   :   90.74126      CP executing     :    45370631300

Million inst/sec (MIPS) :      75.76      Instructions     :     6874941145
Avg. clock periods/inst :       6.60
% CP holding issue      :      78.89      CP holding issue :    35792144013
Inst.buffer fetches/sec :       0.18M     Inst.buf. fetches:       16533020
Floating adds/sec       :      43.69M     F.P. adds        :     3964381135
Floating multiplies/sec :      36.72M     F.P. multiplies  :     3331572756
Floating reciprocal/sec :       0.30M     F.P. reciprocals :       27350746
Cache hits/sec          :      72.97M     Cache hits       :     6621788501
CPU mem. references/sec :     123.83M     CPU references   :    11236805090

Floating ops/CPU second :      80.71M
Floating ops/wall second:      79.73M     CPU/wallclock time ratio:    0.99
```

Figure C.1: HPM Group 0 report for Beltrami benchmark.

```
Group 1:  CPU seconds  :       88.66483  CP executing:    44332416530

  Hold issue condition             % of all CPs       actual # of CPs
Waiting on A-registers/funct. units:   3.08             1366918830
Waiting on S-registers/funct. units:   2.50             1106172502
Waiting on V-registers            :  26.04            11542097302
Waiting on vector functional units :  42.09            18659759849
Waiting on B,T registers          :   0.37              163171284
Waiting on memory ports           :  63.30            28064216670
Waiting on miscellaneous          :  15.31             6786021877
```

Figure C.2: HPM Group 1 report for Beltrami benchmark.

are subroutines or code fragments. HPM does not incur any overhead due to the dedicated hardware counters used to collect performance data. The reports in the following sections should be considered approximate values that may vary from run-to-run. The following subsections describe the specific TASS benchmarks chosen for HPM analysis.

C.2 Beltrami Benchmark

Figures C.1-C.4 show the HPM group reports for the Beltrami TASS benchmark. A Beltrami TASS run is an end-to-end simulation in its simplest form, and is often used for troubleshooting because its results can be compared to an analytic solution. The microphysics are turned off for the Beltrami simulation. The test case used a 3-D grid 15x15x15 points, and typical input parameters (e.g. atmospheric sounding data).

As shown in Figure C.1, the amount of vectorization of the code is medium, indicated by comparing the Floating-point operations per CPU second (FLOPS, 80.7) and the Millions of Instructions per second (MIPS, 75.8) data. A high FLOPS value (around 800 million) coupled with a low MIPS value (around 30) implies many operations are being performed in relatively few instructions, evident when a high percentage of vector instructions utilize long vectors. This benchmark has a low FLOPS rate and a medium MIPS rate, indicating predominately scalar operations. But the high number of vector memory references per

```
Group 2:  CPU seconds   :       86.38900     CP executing  :     43194502255

Inst. buffer fetches/sec   :       0.19M  total fetches    :        16532189
Scalar memory refs/sec     :       2.15M  actual refs      :       186119085
    % of all data refs:    1.66%
  Scalar memory writes/sec :       0.40M  actual writes    :        34412304
  Scalar memory reads/sec  :       1.76M  actual reads     :       151706781
Block memory refs/sec      :     127.92M  actual refs      :     11050686021
  B,T memory refs/sec      :       1.25M  actual refs      :       107924036
  V memory refs/sec        :     126.67M  actual refs      :     10942761985
Cache read hit rate:      75.54%          actual hits      :      6087028139
CPU memory refs/sec        :     130.07M  actual refs      :     11236805106
    avg conflict/ref:   0.01              actual conflicts :       102795827
  CPU memory writes/sec    :      42.92M  actual refs      :      3707904471
  CPU memory reads/sec     :      87.15M  actual refs      :      7528900635
```

Figure C.3: HPM Group 2 report for Beltrami benchmark.

```
Group 3:  CPU seconds  :       87.61513     CP executing:    43807565460

 (octal) type of instruction     inst./CPUsec      actual inst.  % of all inst.
(000-017)jump/special          :       5.16M       451684732      6.57
(020-077)scalar functional unit :     51.99M      4554903119     66.25
(100-137)scalar memory         :       2.12M       186119086      2.71
(140-157,175)vector integer/log.:      0.55M        48078113      0.70
(160-174)vector floating point  :      7.69M       673429396      9.80
(176-177)vector load and store  :     10.97M       960726504     13.97

  type of operation                ops/CPUsec       actual ops   avg. VL
Vector integer&logical&floating :     86.60M      7587170765     10.52
Scalar functional unit         :      51.99M      4554903119

                                 ops/wallsec
Vector integer&logical         :       3.75M      CPU/wallclock
Vector floating point          :      82.75M      time ratio:     1.00
Scalar functional unit         :      51.93M
```

Figure C.4: HPM Group 3 report for Beltrami benchmark.

```
--------------------------------------------------------------------------------------
               TOTAL     DIFF     MARCHL     MICPHY     BNDRY     BUOY     UWP       FILTER + DIAGS
               85.85     0.00     0.00       0.00       0.00      0.04     85.62     0.18     1.70
               98.05%    0.00%    0.00%      0.00%      0.00%     0.04%    97.79%    0.21%    1.95%
 CPU TIME FOR RESTART OUTPUT:   0.00323 (  0.00%)

 TOTAL CPU TIME (IN SEC) FOR SIMULATION:       87.60
 TOTAL = INITIAL + DIAGS + TOTAL + RESTART
```

Figure C.5: Beltrami benchmark output file.

second (126.7 million) compared to the scalar memory references (2.2 million) reported in Figure C.3 supports a high level of vectorization achieved in the code. The disparity could be due to the small domain (15x15x15 grid points) used for the benchmark. This domain size is much smaller than that of a typical simulation and not sufficient to exploit the efficiency of the code design. Figure C.4 reports the average vector length of 10.5. This is small compared to the maximum possible vector length of 128. This is again due to the small domain chosen for the benchmark.

Figure C.1 also shows similar numbers for floating-point multiplies and adds, which coupled with a high percentage (42) of hold issue instructions for vector functional units reported in Figure C.2, implies chaining of vector operations could not be accomplished. Well-balanced numbers for the various operations and low percentages of holds for the functional units indicates the functional units are running simultaneously or operations are being chained. This condition is desirable as it results in high FLOPS rates.

Finally, Figure C.1 shows a high number of CPU memory references per second, and Figure C.2 shows a high (63) percentage of hold issues for memory ports. This information shows the memory is a performance bottleneck for this benchmark. This does not necessarily indicate inefficient code, as with the vector length parameter the benchmark is too small a domain to be representative of the typical simulations targeted by the TASS design. The two following sections describe progressively more specific benchmarks.

C.3 UWP Subroutine Benchmark

The second benchmark examined is an entire subroutine (*uwp.f*) in the *advect.f* module that updates the U,V, and W wind component prognostic variables. As mentioned previously from experience running TASS, this routine is the second highest bottleneck in the code, with the first being *micro.f*. Figure C.5 shows the last few lines of an output file TASS produced for the Beltrami benchmark. The lines in the figure are diagnostic statistics confirming the high percentage of time spent in the *uwp.f* routine for the Beltrami benchmark, with the microphysics turned off.

To gain more information on the vectorization of a user's code, the Cray compiler can be directed to output a *modulename.lst* file, that provides an annotated listing of a code module, with line-by-line indications of code that is vectorized and parallelized by the compiler. An

```
 311                1          C  CALCULATE 3-D DIVERGENCE
 312                1          C
 313                12 -------          DO  K=2,KS2
 314                123 ------             DO  J=JB,JS1
 315                123v -----             DO  I=IB,IS1
 316                123v                     WF(I,J,K)=U(I,J,K,1)*ADIV(I,J,K)
 317                123v                     P(I,J,K,3)=V(I,J,K,1)*P(I,J,K,4)
 318                123v ---->             enddo
 319                123 ----->             enddo
 320                12 ------>             enddo
 321                12 -------          DO  K=2,KS2
 322                123 ------             DO  J=JB,JS2
 323                123v -----             DO  I=IB,IS2
 324                123v                     DIV(I,J,K)=WF(I+1,J,K)-WF(I,J,K)+
 325                123v           *                p(I,J+1,K,3)-p(I,J,K,3)
 326                123v ---->             enddo
 327                123 ----->             enddo
 328                12 ------>             enddo

38) <cf90-6204,Vector,Line=315> A loop starting at line 315 was vectorized.
39) <cf90-8135,Scalar,Line=315> Loop starting at line 315 was unrolled 2 times.
40) <cf90-6204,Vector,Line=323> A loop starting at line 323 was vectorized.
```

Figure C.6: List file example, *uwp* subroutine.

```
Group 0:  CPU seconds   :    3.90667      CP executing     :     1953334900

Million inst/sec (MIPS) :     179.35      Instructions     :      700645116
Avg. clock periods/inst :       2.79
% CP holding issue      :      32.24      CP holding issue :      629772787
Inst.buffer fetches/sec :       4.14M     Inst.buf. fetches:       16161111
Floating adds/sec       :       1.88M     F.P. adds        :        7352674
Floating multiplies/sec :       2.90M     F.P. multiplies  :       11344102
Floating reciprocal/sec :       0.09M     F.P. reciprocals :         367413
Cache hits/sec          :     155.73M     Cache hits       :      608391245
CPU mem. references/sec :      42.91M     CPU references   :      167654553

Floating ops/CPU second :       4.88M
Floating ops/wall second:       4.72M     CPU/wallclock time ratio:    0.97
```

Figure C.7: Group 0 report for greater-than MVL *UWP* benchmark.

example of the *uwp.lst* file is shown in Figure C.6.

For *uwp* and the single-loop benchmark in the next section, runs were performed on different size data domains to compare the performance of different vector loading. Runs were made for domains less-than (15), greater-than (100), and a multiple of (64) the maximum vector length of 64 elements for the Cray SV1. Figures C.7-C.9 are the hpm Group 0 reports for the greater-than, less-than, and multiple of the Maximum Vector Length (MVL) *uwp* benchmarks.

The Figures C.7-C.9 look similar for many of the performance parameters but the main difference is seen in the second line, second column, number of instructions parameter. To execute the same code with a domain size greater-than the MVL results in approximately 700 million instructions counted as compared to around 15 million instructions for the less-than MVL domain. This could be attributed to the additional overhead of stripmining (see

```
Group 0:  CPU seconds   :    0.08726      CP executing     :       43631975

Million inst/sec (MIPS) :     176.28      Instructions     :       15382825
Avg. clock periods/inst :       2.84
% CP holding issue      :      33.72      CP holding issue :       14714693
Inst.buffer fetches/sec :       3.76M     Inst.buf. fetches:         327899
Floating adds/sec       :       1.71M     F.P. adds        :         148840
Floating multiplies/sec :       2.30M     F.P. multiplies  :         200318
Floating reciprocal/sec :       0.09M     F.P. reciprocals :           7948
Cache hits/sec          :     144.15M     Cache hits       :       12579285
CPU mem. references/sec :      43.71M     CPU references   :        3814226

Floating ops/CPU second :       4.09M
Floating ops/wall second:       3.42M     CPU/wallclock time ratio:    0.84
```

Figure C.8: HPM Group 0 report for less-than MVL *UWP* benchmark.

```
Group 0:  CPU seconds   :    1.61339      CP executing     :      806695490

Million inst/sec (MIPS) :     176.99      Instructions     :      285552155
Avg. clock periods/inst :       2.83
% CP holding issue      :      32.45      CP holding issue :      261748598
Inst.buffer fetches/sec :       4.09M     Inst.buf. fetches:        6605732
Floating adds/sec       :       1.68M     F.P. adds        :        2704945
Floating multiplies/sec :       2.75M     F.P. multiplies  :        4438290
Floating reciprocal/sec :       0.09M     F.P. reciprocals :         149901
Cache hits/sec          :     154.26M     Cache hits       :      248885287
CPU mem. references/sec :      42.67M     CPU references   :       68849393

Floating ops/CPU second :       4.52M
Floating ops/wall second:       4.30M     CPU/wallclock time ratio:    0.95
```

Figure C.9: HPM Group 0 report for multiple of MVL *UWP* benchmark.

```
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
```

Figure C.10: Typical loop benchmark, *UWP* subroutine.

```
Group 0:  CPU seconds   :   27.89401      CP executing     :    13947004700

Million inst/sec (MIPS) :     170.59      Instructions     :     4758542648
Avg. clock periods/inst :       2.93
% CP holding issue      :      33.78      CP holding issue :     4711363438
Inst.buffer fetches/sec :       4.40M     Inst.buf. fetches:      122784768
Floating adds/sec       :       0.89M     F.P. adds        :       24892205
Floating multiplies/sec :       1.47M     F.P. multiplies  :       41113994
Floating reciprocal/sec :       0.00M     F.P. reciprocals :              0
Cache hits/sec          :     163.60M     Cache hits       :     4563412025
CPU mem. references/sec :      39.37M     CPU references   :     1098061362

Floating ops/CPU second :       2.37M
Floating ops/wall second:       2.32M     CPU/wallclock time ratio:    0.98
```

Figure C.11: HPM Group 0 report for greater-than MVL loop benchmark.

section D.1.2) for the greater-than MVL domain.

C.4 Typical Loop Benchmark

A single, typical loop from the *uwp* routine was chosen for the last benchmark. It represents the most basic building block of TASS that still exercises vector computing concepts, and consequently is a starting point for the research. There are approximately 25 similar loops in the routine *uwp*. Figure C.10 shows the FORTRAN source code for the loop, and the list file for the entire loop benchmark is in Section C.5.

Figure C.10 shows no cyclic dependencies, allowing the loop to be vectorized without any transformations from the compiler. Lines 40-45 of the *cmnloop* list file for this benchmark confirm this vectorization (See Section C.5). Figure C.11 is the HPM Group 0 report for the

greater-than MVL benchmark case. The comparing Figures C.11 and C.7 shows a similar MIPS rate between the two cases, and the smaller code shows about half the FLOP rate as the entire *UWP* subroutine. This illustrates the care that must be taken when extrapolating smaller code case study results to the larger codes that they are extracted from. In the case of the two benchmarks, the codes executed at similar rates but fewer floating-point operations are present in the smaller benchmark, showing this overall performance metric does not scale linearly in this case. If the 2.4 MFLOPS rate for the typical loop is scaled by the 25 such loops in the *uwp* routine, a number much larger than the reported 4.9 MFLOPS rate in Figure C.7 results.

C.5 CMNLOOP List File

```
Page 1

 ftnlist 3.0.0.15:  initiated by f90 11:13 Sun Jun15,2003  (created 12:43 Wed Feb14,2001)

 Output written to "cmnloop.lst"
        NLSPATH:      /opt/ctl/cf90_sv1/cf90_sv1/nls/En/%N.cat
               :    /opt/ctl/CC_sv1/CC_sv1/nls/En/%N.cat:/opt/ctl/cal/cal/nls/En/%N.cat
               :    /opt/ctl/cf90_sv1/cf90_sv1/nls/En/%N.cat:/opt/ctl/nls/%l/%N.cat
               :    /opt/ctl/craytools/craytools/nls/En/%N.cat
               :    /opt/ctl/craytools/craytools/nls/%l/%N.cat:/opt/ctl/nls/En/%N.cat
               :    /opt/ctl/craylibs/craylibs/nls/En/%N.cat
               :    /opt/ctl/craylibs/craylibs/nls/%l/%N.cat
               :    /opt/ctl/craytools/craytools/nls/%l/%N.cat:/opt/ctl/nls/%l/%N.cat
               :    /opt/ctl/nls/En/%N.cat:/opt/ctl/craytools/craytools/nls/%l/%N.cat
               :    /opt/ctl/craytools/craytools/nls/En/%N.cat

 ftnlist   -o cmnloop.lst cmnloop.f


        ftnlist:  "cmnloop.f" - last modified at 11:09 Sun Jun15,2003
               :  "/u/bd/rutishau/micro_bench/ltmvl/cmnloop.f"
            f90:  3.5.0.2 (on CRAY-SV1 "bright" for target CRAY-SV1) at 11:12 Sun Jun15,2003
        f90 cmdline:  /opt/ctl/cf90_sv1/cf90_sv1/bin/ftn_driver.exe -c -Scmnloop.s -r2
               :    -Ovector3,scalar3 cmnloop.f -I/opt/ctl/craytools/craytools/include
               :    -I/opt/ctl/craylibs/craylibs/include -I/opt/ctl/CCtoollib/CCtoollib/CC
               :    -I/opt/ctl/CCmathlib/CCmathlib/CC
               :    -L/opt/ctl/CCtoollib/CCtoollib/lib,/opt/ctl/CCmathlib/CCmathlib/lib
               :    -L/opt/ctl/craylibs/craylibs -L/opt/ctl/craytools/craytools/lib
               :  -e pqrSt  -d afijnuvBAX  -Caox  Int64  MsgLevel=3  Vopt=0
               :  Optimizations:  BLoad,Recurrence,Scalar=3,Vector=3,VSearch,Task=1
               :  -I /opt/ctl/craytools/craytools/include
               :  -I /opt/ctl/craylibs/craylibs/include  -I /opt/ctl/CCtoollib/CCtoollib/CC
               :  -I /opt/ctl/CCmathlib/CCmathlib/CC
               :  -p /opt/ctl/craylibs/craylibs/libmodules.a

 CMNLOOP    cmnloop.f    11:09 Sun Jun15,2003                                                     Page 2
 -------
      1                               PROGRAM CMNLOOP
      2
      3
      4                         C initialize memory arrays
      5                               PARAMETER (IS=10,JS=10,KS=10)
      6
      7                               INTEGER I,J,K,Q
      8                               REAL U(IS,JS,KS,4),P(IS,JS,KS,4),A(IS,JS,KS),E(IS,JS,KS)
      9                               REAL ALS,BTS,X1
     10
     11                         C FILL ARRAYS WITH DUMMY DATA
     12
     13             1 --------        DO 1 K=1,KS
     14             12 -------          DO 2 J=1,JS
     15             12v ------            DO 3 I=1,IS
     16             1234 -----               DO 4 Q=1,4
     17             1234                        U(I,J,K,Q) = 0.5+Q+I
     18             1234                        P(I,J,K,Q) = 1.5+Q+I
     19             1234 ---->   4           CONTINUE
     20             12v                      A(I,J,K) = K+J+0.5
     21             12v                      E(I,J,K) = K+0.4
     22             12v ----->   3         CONTINUE
     23             12 ------>   2       CONTINUE
```

```
    24              1 ------->   1      CONTINUE
    25                                  ALS = 6.7
    26                                  BTS = 0.2
    27                                  X1 = 2.8
    28
    29                                  OPEN(23,FILE='TEST.OUT')
    30                           C
    31                           C
    32                           C  CALCULATE U COMPONENT OF VELOCITY
    33                           C
    34                           C
    35                           C
    36                           C   ADVANCE U TO NEXT TIME LEVEL
    37                           C
    38                           C
    39              1 --------            DO 11 K=1,KS
    40              1v -------              DO 11 J=1,JS
    41              123 ------              DO 11 I=2,IS
    42              123                        X1=U(I,J,K,2)
    43              123                        U(I,J,K,2)=(P(I,J,K,1)-P(I-1,J,K,1))*A(I,J,K)
    44              123                        E(I,J,K)=ALS*U(I,J,K,2)+BTS*X1+U(I,J,K,4)
    45              12v =====>   11      CONTINUE
    46                           c must do something with data so loop will not be removed by optimization
    47              1 --------            DO 12 K=1,KS
    48              12 -------              DO 12 J=1,JS
    49              123 ------              DO 12 I=2,IS
    50              123                      write(23,1000) U(I,J,K,2),E(I,J,K)
    51              123          1000        format(F10.4)
    52              123 =====>   12         CONTINUE
    53                                      close(23)
    54                                  END

CMNLOOP    cmnloop.f    11:09 Sun Jun15,2003                                                            Page 3
-------


f90 Compiler - 7 messages:

  1) <cf90-6005,Scalar,Line=14> A loop starting at line 14 was unrolled 2 times.
  2) <cf90-6205,Vector,Line=15> A loop starting at line 15 was vectorized with a single vector iteration.
  3) <cf90-6008,Scalar,Line=16> A loop starting at line 16 was unwound.
  4) <cf90-6003,Scalar,Line=39> A loop starting at line 39 was collapsed into the loop starting at line 40.
  5) <cf90-6204,Vector,Line=40> A loop starting at line 40 was vectorized.
  6) <cf90-6008,Scalar,Line=41> A loop starting at line 41 was unwound.
  7) <cf90-6005,Scalar,Line=41> A loop starting at line 41 was unrolled 2 times.

Note:  Enter "explain cf90-6005" (for example) to get a more detailed explanation of f90 Scalar message 6005

=====================================================================================================================================

"ftnlist  3.0.0.15" 11:13 Sun Jun15,2003  1 Subprogram:  0 Local
Messages
```

Appendix D

Background on Vector Processing

This section provides a brief introduction to the concepts of vector processing, to provide background on the processing model chosen for this research. In traditional scalar processing, instructions operate on scalar operands, for example, one 64-bit integer. Vector processors have instructions whose operands are vectors, or ordered sets (arrays) of v_l elements, where v_l is the length of the vector. Each vector element is a scalar, and can be any data type supported by the machine. Vector processing is a form of Single Instruction Multiple Data (SIMD) computing. Vector processing was devised to address limitations in scalar processing that schemes such as pipelining and superscalar processing cannot overcome. These limitations have to do with data dependencies in deep pipelines and between instruction operands [8]. Vector processing by definition assures the scalar elements of a vector operand are independent for any particular computation. In addition, a vector operation is equivalent to an entire loop on a scalar processor, reducing the required instruction bandwidth. There are many scientific applications with large working sets that lend themselves to the SIMD computing paradigm. The TASS model, for example, is a gridded domain representing a portion of the atmosphere, where identical prognostic equations are computed for various physical parameters such as wind velocity, pressure, and temperature.

A typical vector machine has all the components of a scalar processor, with the addition of vector functional units and vector control units. Most vector machines also utilize vector registers, although direct access to memory by the functional units can also be used.

Vector functional units are implemented using the concept of pipelining. In the Cray-1, with a vector length of 64, the vector floating-point addition unit does not have 64 64-bit adders implemented in silicon for its design. Instead, a vector functional unit has one adder fed by a pipeline. When a vector instruction is decoded, it is issued to a control unit for the appropriate vector functional unit. The control unit manages the vector operation in the hardware or firmware, by moving data between the vector register and the functional unit pipelines and performing the operation.

Table D.1: Example vector assembly instruction.

Mnemonic	**Operands**	**Function**
ADDV	V1 V2 V3	Add elements of V2 and V3 and store result in V1

The types of vector operations can be broken down into the following four primitive operations [57]:

$$f_1 = V \to V \qquad f_2 = V \to S \qquad f_3 = V \times V \to V \qquad f_4 = V \times S \to V, \tag{D.1}$$

where V is a vector and S is scalar. Examples are vector square root (f_1), vector maximum (f_2), vector add (f_3), and vector-scalar multiply (f_4). On a vector-register machine, a vector addition instruction may be structured as shown in Table D.1.

To maximize the potential to utilize the resources of a vector computer, additional instructions are typically included to facilitate data mapping and restructuring to produce vectors from data not explicitly organized as a vector. Examples of these instructions are COMPRESS, MASK, SCATTER, and GATHER. These instructions use a bit vector to control manipulation of vector operands. For instance, in the COMPRESS instruction, a bit vector (0,1,1,0) may be used to operate on a vector A=(23,41,56,44) to produce a resultant vector B=(41,56). The notation for these operations was first developed in the Iverson language [95]. Using instructions like the ADDV in Table D.1, it is easy to see how a vector instruction eliminates the loop branching and conditional testing overhead in a scalar processor.

D.1 Performance Issues in Vector Computers

The benefits of vector processing units over scalar processing units for a number of applications are clear, but there are numerous performance issues associated with vector architecture implementations. These performance issues center on the complications of data movement between the memory, registers, and functional units, and the desire to keep the functional pipelines full as often as possible to maximize performance. There is inherent overhead in the pipelined operation of the functional units as well. This section describes the issues of vector startup, strip-mining, stride, and a performance enhancement technique called chaining.

D.1.1 Startup Latency

As with any pipeline, results can theoretically be obtained each clock cycle, once the pipeline is filled. Vector processors have a startup time or latency that is directly related to the depth of the functional unit pipelines. The vector break even length [96] is the minimum vector length for which a vector processor speedup surpasses an equivalent scalar processor. Another

parameter is the initiation rate, defined as the time per result once the vector instruction is running (usually 1 per cycle) [8]. Thus computing the clock cycles per completed element of a vector result is:

$$\text{Cycles per result} \;=\; \frac{\text{startup latency + N * initiation rate}}{N}, \tag{D.2}$$

For a 15-cycle startup, 1 cycle initiation rate, and 64-element vector:

$$\text{Cycles} = (15 + (1 * 64))/64 = 1.23 \text{ cycles per element}, \tag{D.3}$$

The initiation rate is directly related to the functional unit execution time, and can be kept to one cycle by controlling the functional pipeline depth [8].

$$\text{Pipeline depth} = \frac{\text{Total functional unit execution time}}{\text{Machine clock cycle time}}, \tag{D.4}$$

The same performance parameters can be used for vector loads and stores. Because these operations involve the memory, startup times can be much higher than for the functional units. To minimize this, independent memory banks are often used to allow for parallel memory accesses. If there are more memory banks than the clock cycle latency for memory bank access, memory accesses can occur with a one-cycle latency (effectively pipelining a memory operation.) Note operations involving more than one vector operation, such as $S = a * X + Y$ ($SAXPY$, for "sum of a times X plus Y") will have initiation rates for the result S that are higher (the sum of the rates for the vector multiply and add units) than for a single functional unit.

D.1.2 Stripmining

A subcomponent of a typical vector load/store unit is the vector length register (VR). This register holds an integer from 1 to the Maximum Vector Length (MVL), and is used for a procedure called stripmining. Stripmining is a means for handling a vector operation for an operand with a length that is greater than the MVL. For operands with v_l <MVL, the vector length register is simply set to v_l. But, for example, an operand with length $v_l = 100$ on a machine with a MVL of 64 must be handled differently. In this case, an operation with $VR = 64$ must be followed by an operation with $VR = 36$. The compiler must compute the number of VR =MVL operations and the remaining operation with VR <MVL, and introduce loops in the code generation phase to handle the computation. Figure D.1 shows the overhead in terms of additional instructions required for stripmining, using pseudo vector code. The operation performed is the $SAXPY$ operation, where S is the vector sum of a vector X multiplied by a scalar a and added to another vector Y ($S = a * X + Y$). As seen

(a) Data = 16 elements, MVL = 16

```
F0 = a  //F is a scalar register
VR = 16 //load vector length register
V1 = mem[address(X)]
V2 = V1 * a
V3 = mem[address(Y)]
V1 = V2 + V3
mem[address(result)] = V1            //6 instructions
```

b) Data = 20 elements, MVL = 16

```
VR = 4
F0 = a
V1 = mem[address(X)..address(X+VR)]
V2 = V1 * a
V3 = mem[address(Y)..address(Y+VR)]
V1 = V2 + V3
mem[address(result)..address(result+VR] = V1
F1 = VR
VR = 16
V1 = mem[address(X+F1)..address(X+F1+VR)]
V2 = V1 * a
V3 = mem[address(Y+F1)..address(Y+F1+VR)]
V1 = V2 + V3
mem[address(result+F1)..address(result+F1+VR] = V1  //14 instructions
```

Figure D.1: Pseudocode comparison of non-stripmined (a), and stripmined (b) vector code.

in Figure D.1, with just one additional iteration required for an operand between the MVL and 2MVL, the number of instructions more than doubles.

D.1.3 Vector Stride

Vector stride refers to the distance between two vector elements in physical memory. The stride is largely dependent on the method a given high-level language stores arrays in memory. C, for example, is row major, or for a two-dimensional array A, A(i,j) and A(i,j+1) are adjacent in physical memory. By contrast, FORTRAN stores arrays in column-major order, so elements A(i,j) and A(i+1,j) are contiguous in memory. Vector machines may include special load and store instructions in their Instruction Set Architecture (ISA) to support operations with stride. These are effectively load and store operations with a step applied to the address between each element that can be greater than one. Vector stride introduces the performance issue of memory bank conflict [8]. If the stride is such that it is multiple of the number of memory banks, attempts to access the same memory bank would occur each clock cycle. This is similar to "thrashing" in virtual memory systems.

D.1.4 Vector Chaining

Unlike the previous sections highlighting potential performance problems in vector processors, this section describes the performance enhancement vector chaining. Vector chaining is a form of vector functional pipelining, and it exploits parallelism among the elements of a vector. Consider a modification of the $SAXPY$ operation:

$$C = a * X + Y \qquad S = X + C, \tag{D.5}$$

Here the computation of S will have to wait until the computation of C is complete. With vector chaining, the output pipeline for C can be connected to an input of the addition functional unit used to compute S, such that the functional units are connected in a pipeline, and elements of S can be computed as soon as elements of C are ready.

D.2 Vector Compilers

The vector architecture described in the previous section is of little value without a means of translating the high-level problem description (code) into a representation that can utilize the hardware resources effectively. This translation is done with a vectorizing compiler. The task of the vectorizing compiler is essentially to recognize portions of the input code that can be vectorized, typically coded as loops. To vectorize a loop, the semantics of the vector operation must be the same as the original sequential operation. Conditions inhibiting vectorization include conditional and branch statements, data dependencies, indirect indexing, and subroutine calls within loops [57]. Much of vector compilation focuses on dependence analysis. In general, a loop can be vectorized if a value computed in one iteration of a loop is not used in a later iteration. Transformations exist [1] to convert code not initially vectorizable into vector code.

Figure D.2 shows the typical components of a vectorizing compiler. The goal is to transform the input source code (e.g. FORTRAN) to a vector and parallel version, called "VP-Source Code" in the figure. Note VP-source representation is still high-level, and the usual compilation steps resulting in machine code generation still remain.

Similar to compilers for scalar machines, the initial steps performed on the source input are scanning, parsing, and semantic analysis [97]. The syntactic normalization and simplification steps prepare for the final two front-end steps of flow and dependence analysis. The syntactic normalization includes operations such as converting DO loop indices so they all start at one and increment by one. Syntactic simplification includes removing "dead" code and algebraic simplifications. The flow analysis and dependence analysis are then performed. During these analysis steps, an alternate internal representation (e.g. dependence graph) of the input program is produced. The back-end of the vectorizing compiler includes various

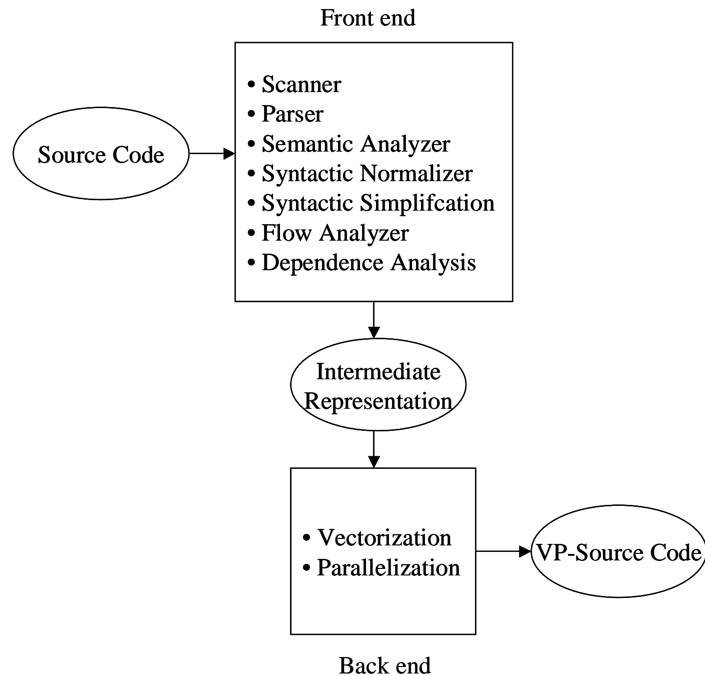

Figure D.2: Vectorizing compiler components [1].

levels of optimization, including vector code generation that transforms scalar loops into vector operations, pipeline parallelization and chaining, and vector register optimization. The register optimization exists to attempt to keep a continuous stream of operands supplied to the execution units, resulting in optimum computational performance [57].

Early work to arrive at a FORTRAN language standard supporting vectorization can be found in [98]. These efforts were driven by the fact that scientific codes are commonly written in a manner that facilitates vectorization by a particular platform's compiler. Efforts such as VECTRAN attempted to build the vector facilitation into the language as a standard, with the goal of eliminating some of the platform-specific recoding. Reference [99] describes a translator for converting legacy FORTRAN to a vector-capable FORTRAN standard, called FORTRAN 8x at the time. In [100], the translator in [99] is adapted to a production compiler for the IBM System/370.

Appendix E

Case Study Application

The Terminal Area Simulation System (TASS) is a three-dimensional large eddy simulation model with a meteorological framework [7]. TASS is capable of modeling complex atmospheric phenomenon, such as thunderstorms, turbulence, tornadoes, microbursts, and aircraft wake vortices. The simulated phenomena represent hazards to aviation. TASS has been used successfully on a variety of NASA projects such as the Aircraft Vortex Spacing System (AVOSS), NASA/FAA Windshear Program, Space Shuttle Exhaust Studies, Nuclear Winter and Cloud Rise Studies, and three NTSB accident investigations. The TASS model uses 12 prognostic equations that include one each for momentum (east, west, and vertical components), pressure deviation, and potential temperature; six equations for water continuity; and a final equation for a massless tracer. From [101] the equations are as follows:

Momentum:

$$\frac{\delta u_i}{\delta t} + \frac{H}{\rho_o}\frac{\delta p}{\delta x_i} = -\frac{\delta u_i u_j}{\delta x_j} + u_i \frac{\delta u_j}{\delta x_j} + g(H-1)\delta_{13} - 2\Omega_j (u_k - u_{o_k})\varepsilon_{ijk} + \frac{1}{\rho_o}\frac{\delta \tau_{ij}}{\delta x_j}, \tag{E.1}$$

Buoyancy Term:

$$H = [\frac{\Theta}{\Theta_o} - \frac{pC_v}{P_o C_p}][1 + 0.61(Q_v - Q_{vo}) - Q_T], \tag{E.2}$$

Pressure Deviation:

$$\frac{\delta p}{\delta t} + \frac{C_p P}{C_V}\frac{\delta u_j}{\delta x_j} = \rho_o g u_j \delta_{j3} + \frac{C_p}{C_v}\frac{P}{\Theta}\frac{d\Theta}{dt}, \tag{E.3}$$

Thermodynamic Equation (Potential Temperature):

$$\frac{\delta \Theta}{\delta t} = -\frac{1}{\rho_o}\frac{\delta \Theta \rho_o u_j}{\delta x_j} + \frac{\Theta}{\rho_o}\frac{\delta \rho_o u_j}{\delta x_j} + \frac{1}{\rho_o}\frac{\delta S_j(\Theta)}{\delta x_j} + \frac{\Theta}{TC_p}[L_v s_v + L_f s_f + L_s s_s], \tag{E.4}$$

where the *Potential Temperature* is:

$$\Theta = T(\frac{P_{oo}}{P})^{\frac{R_d}{C_p}}, \tag{E.5}$$

The symbology in the equations above is as follows: u_i is a tensor component of velocity, t is time, p is deviation from atmospheric pressure P, T is atmospheric temperature, ρ is air density, Ω is the Earth's angular velocity, C_p and C_v are the specific heats of air, g is the Earth's gravitational acceleration, R_d is the gas constant for dry air, P_{oo} is a constant, Q_v is the mixing ratio for water vapor, Q_T is the sum of the mixing ratios for liquid and ice water, L_v, L_f, and L_s are the latent heats of vaporization, fusion, and sublimation of water, and finally s_v, s_f, s_s are the corresponding water source terms. The quantities with subscript "o" are initial values obtained from an input environmental sounding. The equation set is compressible, non-Boussinesq, with subgrid turbulence closure. Non-Boussinesq indicates the Boussinesq approximation, where the density of the atmosphere is treated as a constant except where it is coupled with gravity in the vertical momentum equation, is not used [102]. Sub-grid turbulence closure refers to a scheme to parameterize the effects of turbulence eddies in the atmosphere that are on scales less than the chosen grid size. Details of this process can be found in [7].

The equations are computed using a finite-differencing scheme in a spatial domain defined by a two or three-dimensional grid. The grid is classifed as an Arakawa C Mesh [7], and is shown for one two-dimensional horizontal slice in Figure E.1. The grid scheme allows for conservation of quadratic quantities such as kinetic energy, and leads to improved accuracy of the model. In the figure, all variables other than the velocities are computed at the midpoint of each cell. Velocities are computed at the midpoints of each cell face, where the velocity component normal to each face is computed. The finite-differencing scheme computes grid node values using the following space-difference operator.

$$(\delta_x, Q) = \frac{1}{\Delta x}[Q(x + \frac{\Delta x}{2}) - Q(x - \frac{\Delta x}{2})], \tag{E.6}$$

where Q is the quantity of interest, x is the spatial variable of interest, and Δx is the distance between the points for which the finite difference is taken. The average between the same two points is also used in the scheme and is as follows.

$$\overline{Q}^x = \frac{1}{2}[Q(x + \frac{\Delta x}{2}) + Q(x - \frac{\Delta x}{2})], \tag{E.7}$$

Further details of how these operators are applied to Equations E.1 to E.4 can be found in [7]. The equations are integrated in time with a multi-step scheme. A generalized Adams-

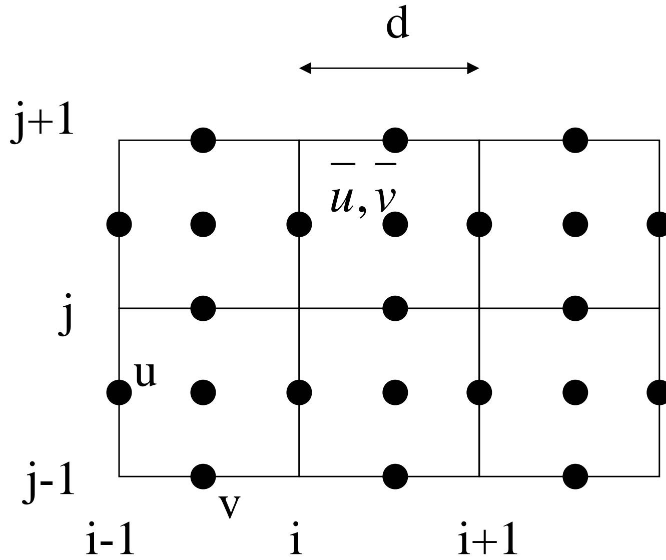

Figure E.1: TASS grid scheme.

Bashforth time-differencing scheme that allows for a variable time step is used. An example of the approximation for the u component of velocity is:

$$u^{n+1} = u^n + \frac{\Delta t_N}{2m}[3(\frac{\delta u}{\delta t})_s^n - (\frac{\delta u}{\delta t})_s^{n-1}] + \frac{\Delta t_N}{m}[(1 + \frac{\Delta t_N}{2\Delta t_{N-1}})(\frac{\delta u}{\delta t})_L^N - \frac{\Delta t_N}{2\Delta t_{N-1}}(\frac{\delta u}{\delta t})_L^{N-1}], \quad \text{(E.8)}$$

where the subscripts L and s denote the derivatives being taken over the large and small time steps, respectively, and there are m small time steps per each large time step. The computations for Equation E.8 are used in the benchmark analysis appearing in Appendix C.

TASS is written in FORTRAN and has approximately 34,500 lines of code in the Cray implementation (version 7.0 used for the case study). The TASS code is divided into eleven modules, whose names and functions are described in Table E.1.

Experience in running the code [7] has revealed more than half of the execution time is spent in the microphysics computations module *micro.f*, and the second-highest percentage of the execution time is in the small time step integration of velocities and pressures (see Equation E.8), module *march.f*. TASS can be run without the complicated microphysics calculations, either for test or an application where the microphysics are not relevant, such as for simulating aircraft wake vortices. The benchmarks in Appendix C use code from *march.f* to capture the second-most computationally intensive routine.

Table E.1: TASS module descriptions. [3]

Module Name	**Module Function**
driver.f	Serves to control execution of overall code.
advect.f	Computes advection and diffusion of all prognostic variables. Updates all scalar prognostic variables.
bounds.f	Sets the boundaries.
dataproc.f	Selects and processes data for output.
diagvar.f	Computes diagnostic terms such as eddy viscosity, buoyancy, and divergence. Also outputs monitoring data from the TASS run.
initial.f	Initializes the domain, and sets up any necessary data areas needed for the run.
initper.f	Initializes thermal and velocity perturbation and wake vortex fields.
io.f	Contains subroutines for input and output of data.
march.f	Updates time integration terms necessary for stability, monitors the movement of the domain for tracking, and updates the velocity and pressure, based on acoustically active terms.
micro.f	Computes cloud microphysics, including condensation and evaporation. Also computes hydrometer terminal velocities.
util.f	Contains utility files such as interpolation and random number generation.