5-2009

A Methodology to Design Pipelined Simulated Annealing Kernel Accelerators on Space-borne Field-Programmable Gate Arrays

Jeffrey Michael Carver

Utah State University

A METHODOLOGY TO DESIGN PIPELINED SIMULATED ANNEALING KERNEL ACCELERATORS ON SPACE-BORNE FIELD-PROGRAMMABLE GATE ARRAYS

by

Jeffrey Michael Carver

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Engineering

Approved:

Dr. Aravind Dasu Major Professor	Dr. Brandon Eames Committee Member
Dr. Edmund A. Spencer Committee Member	Dr. Byron R. Burnham Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2009

Abstract

A Methodology to Design Pipelined Simulated Annealing Kernel Accelerators on Space-borne Field-Programmable Gate Arrays

by

Jeffrey Michael Carver, Master of Science

Utah State University, 2009

Major Professor: Dr. Aravind Dasu
Department: Electrical and Computer Engineering

Increased levels of science objectives expected from spacecraft systems necessitate the ability to carry out fast on-board autonomous mission planning and scheduling. Heterogeneous radiation-hardened Field Programmable Gate Arrays (FPGAs) with embedded multiplier and memory modules are well suited to support the acceleration of scheduling algorithms. A methodology to design circuits specifically to accelerate Simulated Annealing Kernels (SAKs) in event scheduling algorithms is shown. The main contribution of this thesis is the low complexity scoring calculation used for the heuristic mapping algorithm used to balance resource allocation across a coarse-grained pipelined data-path. The methodology was exercised over various kernels with different cost functions and problem sizes. These test cases were benchedmarked for execution time, resource usage, power, and energy on a Xilinx Virtex 4 LX QR 200 FPGA and a BAE RAD 750 microprocessor.

(56 pages)

To my beautiful daughters, Emily and Sarah

Acknowledgments

I am grateful for the many hours of help my advisor gave me in editing my papers and refining my ideas. Dr. Eames was helpful for bouncing off ideas as well as encouraging me to finish when my life got tough. I am grateful for my parents for helping me grow up to learn the skills enabling me to do my thesis. I also thank my friends and other family members not specifically mentioned for their support. Lastly, I am grateful for my girls, Emily and Sarah, for without them I would not have been able to keep the motivation to finish my thesis.

Jeffrey Michael Carver

Contents

Page

Abstract . . . . **iii**

Acknowledgments . . . . **v**

List of Tables . . . . **vii**

List of Figures . . . . **viii**

1 Introduction **1**

2 Background **3**
2.1 Field-Programmable Gate Arrays 3
2.2 Template-Based Architecture Exploration Techniques 6
2.3 Scheduling and Binding Techniques 8

3 Hardware Architecture Template for Accelerating Pipelined Simulated Annealing Kernels **12**

4 Architecture Derivation Methodology **21**
4.1 The PLB Algorithm 21
4.2 Resource Estimation 24
4.3 The Scheduler 27
4.4 The Mapper 31
4.5 FPGA Architecture Generation 36

5 Results **38**

6 Conclusions **44**

References **45**

List of Tables

Table Page

4.1 Estimated vs. actual device primitive usage for a circuit composed of an integer adder followed by an integer multiplier. 25

4.2 Estimated vs. actual device primitive usage for a circuit composed of an integer adder, integer multiplier, and three 2:1 multiplexers. 26

4.3 Different integer components implemented on a Xilinx Virtex 4 FPGA. . . 26

4.4 WSDPs for implementations of Table 4.3. 27

5.1 Device primitive usage, time to complete, power, and energy required for the FPGA based pSAKs compared to PPC 750 based SAKs. 39

5.2 Final latencies (measured in number of clock cycles) of the different sub-systems for the nine test cases. 39

5.3 Time to run the tool, number of nodes for *Alter* and *Evaluate* sub-systems, and the number of iterations of loops in the PLB algorithm (*Loop*1 and *Loop*2 in fig. 4.1). 40

5.4 Final scores and %Error averaged over 100 runs of the pipelined (pSAKs) vs. non-pipelined (SAKs) versions for the nine test cases. 43

List of Figures

Figure Page

2.1 Example circuit ((not (A) and B) or (C and D)) being mapped to a LUT on an FPGA. 4

2.2 Slice on an FPGA shown with a LUT, FF, and specialized carry-chain logic. 4

2.3 Virtex 4 FPGA architecture showing heterogeneous components. 5

3.1 Multiple solution storage approach to illustrate pipelining in pSAKs. 13

3.2 Template for pSAK on an FPGA. 15

3.3 Memory banks and memory multiplexing for pSAK on an FPGA. 17

3.4 Template of a RAM module to support reading from/writing to one, two, or four contiguous locations. 17

3.5 Common templates for sub-systems of the *CAEA* pipeline for interacting with the memory banks. 20

4.1 Data structure used for keeping track of component bindings. 35

5.1 Working of the PLB algorithm on the tsp100 problem. 41

5.2 Performance of the *Scheduler* and *Mapper* for the tsp100 problem. 41

Chapter 1

Introduction

It is expected that in the future spacecrafts/rovers will have a set of tasks or events, that need to be completed subject to some constraints such as time, energy, etc. These tasks may depend upon each other and/or may compete for limited resources. For example, a complex sequence of thruster firings and robotic arm maneuvers might be necessary to grasp a passing object, without violating known types of dependencies. Such a problem can be modeled as a dependency graph violation (DGV) removal problem. A second example could be to determine the least number and types of robots (in a swarm with different sensors) necessary to explore a terrain. Such a problem can be interpreted and solved as swarm-based graph coloring (GC) problem. A third example could be a plan to visit a specific set of sites of scientific interest by a rover in the best order possible to minimize total distance travelled hence minimizing expenditure of energy in the batteries. Such a problem can be modeled as traveling salesperson (TSP) problem.

While missions of the past have needed relatively simple on-board schedulers, future missions to explore outer space planets with the Moon and Mars as home bases, will require unprecedented levels of autonomy. One of the components necessary to support high levels of autonomy is a sophisticated activity planner. A data point that brings things into perspective is the recent Remote Agent experiment [1] flying on-board the New Millennium Deep Space One mission. It executes on a 25 MHz RAD 6000 flight processor, and takes approximately 4 hours to produce a 3-day operations plan. The conclusions from this experiment are:

> While this is a significant improvement over waiting for ground intervention, making the planning process even more responsive (e.g., on a time scale of seconds) to changes in the operations context, would increase the overall time for which the spacecraft has a consistent plan [1].

Software descriptions of classic Simulated Annealing (SA) algorithms are sequential, and not directly well-suited for acceleration on parallel computing platforms like Field Programmable Gate Arrays (FPGAs). However, if carefully modified, these algorithms can be converted into pipelined versions, allowing for multiple solutions to be evaluated simultaneously. Such altered and valid forms of Simulated Annealing Kernels (SAKs), henceforth termed as pipelined Simulated Annealing Kernels (pSAKs), can be accelerated considerably on FPGAs if the underlying micro architectures of data paths, memory, and control sub-systems are appropriately designed. This paper presents the methodology to design pSAK accelerator circuits on FPGAs through the use of (i) a hardware template to aid architecture exploration, (ii) scheduling and mapping (binding) algorithms to balance resource allocation across the coarse grained pipelined data-path, and (iii) a method to calculate the relative weight of components implemented on heterogeneous FPGAs.

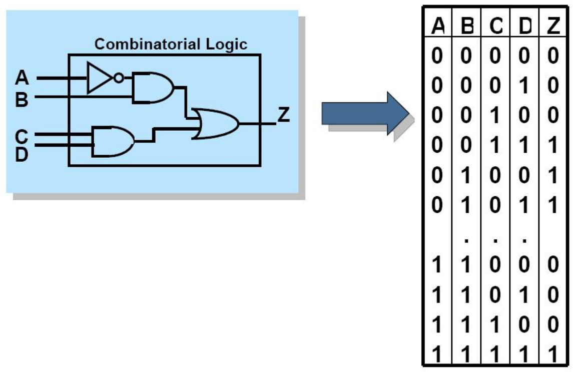
The rest of the paper is organized as follows: Chapter 2 briefly introduces FPGAs, FPGAs in space and techniques to mitigate Single Event Upsets (SEUs), and reviews the literature on template-based architecture exploration techniques, and high level synthesis algorithms for scheduling and mapping. In Chapter 3 the hardware architecture template for accelerating pSAKs is presented along with details on the parameters, memory banks, and memory multiplexing used in the architecture. In Chapter 4 the semi-automated architecture derivation methodology (tool flow) is discussed. Chapter 5 presents the results obtained by exercising this methodology over various kernels with different cost functions and problem sizes. The paper is concluded by summarizing the major contribution of this thesis.

Chapter 2
Background

2.1 Field-Programmable Gate Arrays

An FPGA is a silicon device that has programmable interconnect points (PIPs), memory elements, and lookup tables (LUTs). The main providers of FPGAs are Xilinx [2] and Altera [3]. The PIPs are turned on or off depending on the functionality desired routing required for the FPGA. FPGAs also have memory elements to store information from the computations being performed such as flip-flops (FFs). These memory elements on FPGAs can also be configurable in terms of operating such as storing memory on the negative or positive rising edge of the clock, asynchronous or synchronous reset, and whether to reset to a logic '0' or logic '1'. The basic building blocks of FPGAs is a LUT. The basic LUT consists of multiple input lines and a single output line. The input lines act as an address which is used to lookup the value in the table to use to drive the output line of the LUT. The values inside the lookup table are programmed with the desired mapping to be done. An example mapping of a circuit to a LUT is shown in fig. 2.1 [4]. Multiple LUTs are used if mapping functionality that requires more inputs than are available on a LUT. A simplified view of a Slice (contains LUTs, FFs, PIPs, and other ASIC components) on a Xilinx FPGA is shown in fig. 2.2 [4].

Modern FPGAs are also increasingly offering the option of reprogramming during runtime some of the PIPs, memory elements, and LUTs on part of the chip while the other part of the chip continues to operate without interruption [5] from [4]. To increase performance, FPGAs are now offering some embedded application specific integrated circuits (ASICs) such digital signal processors (DSPs), block random access memories (BRAMs), digital clock managers, and embedded multiplexers. An example of this can be seen in Virtex-4 architecture shown in fig. 2.3. The advantage of FPGAs when creating a small number of



A	B	C	D	Z
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
	.	.	.	
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Fig. 2.1: Example circuit ((not (A) and B) or (C and D)) being mapped to a LUT on an FPGA.

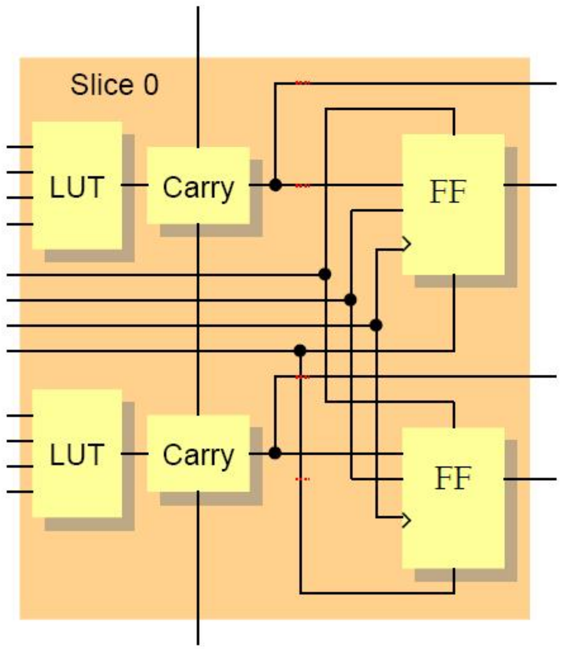


Fig. 2.2: Slice on an FPGA shown with a LUT, FF, and specialized carry-chain logic.

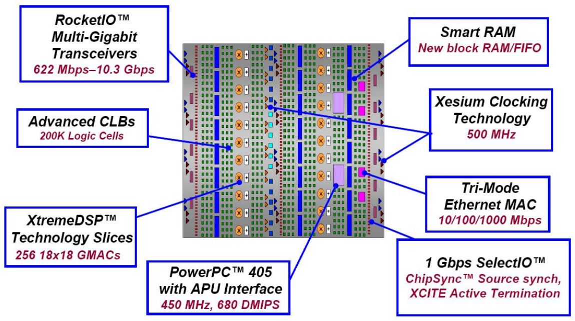


Fig. 2.3: Virtex 4 FPGA architecture showing heterogeneous components.

circuits compared to ASICs is the lower non-recurring engineering (NRE) cost because of not having to fabricate the circuit. The disadvantage of FPGAs compared to ASICs is the higher power usage that comes from having reprogrammable parts instead of hard-wired parts, increased chip area usage, and a lower clock rate.

Such scheduling or planning problems necessary for autonomous space exploration can be solved through techniques like simulated annealing (SA). For instance the Generalized Robotic Autonomous Mobile Mission Planning System (GRAMMPS) [6] uses a Simulated Annealing Kernel (SAK) for path planning. Sanchez-Ante presents a SA algorithm for path planning in multiple robot systems [7]. Fayard proposes and explores the need for SA based schedulers in future space robotic applications [8]. Lee describes a SA based technique for optimizing trajectories of spacecraft driven by propulsion systems that generate low thrusts, subject to the goals of minimizing fuel and time spent [9]. Scherer discusses the benefits of using SA techniques for spacecraft event scheduling [10].

Since space-based radiation hardened microprocessors have failed to keep pace with the computation capabilities of their commodity counterparts, it is unrealistic to continue on that path and expect real-time support for high levels of autonomy. Therefore, the aerospace community in general has been making a paradigm shift in the area of on-board computer chips to adopt FPGAs as the primary compute intensive platform of choice. For example, the Venus Express, the Mars Reconnaissance Orbiter, GRACE, OPTUS, TACSAT2, CIBOLA, and a number of classified programs have included Xilinx FPGAs [11]. Quad-redundant XQR4062XL FPGAs performed mission-critical landing duties (pyrocontrol) on the Mars Exploration Rover (MER) 2003 landers, which delivered the Spirit and Opportunity MERs to the surface of Mars. Virtex XQRV1000 FPGAs handle motor control functions on both MERs as they explore the Martian landscape [11].

Today's state-of-the-art radiation hardened SRAM FPGAs [11] are computationally powerful enough for demanding-space-borne applications such as image processing, radar signal processing, software defined radios, event scheduling, etc. But space-borne FPGAs are susceptible to a variety of problems due to exposure to space radiation. Among them, serious ones that are detrimental to the device, such as total ionization dose (TID) and single event latch-up (SEL), are mitigated/protected against by the device manufacturer. For instance, the Xilinx V4 radiation-hardened series offers protection up to 300 krads for TID and a SEL immunity of greater than 125 MeV-cm2/mg [11]. However SEUs, that change memory element values, seem to be the only potentially significant problem. These need to be protected against or their effects mitigated by design techniques such as TMR, Double Modular Redundancy (DMR), etc. TMR [12] is triplication of a hardware module along with voters as necessary to select the correct output of the device. However, if multiple modules are involved in a design, intermediate voting of TMR protected modules is optional at the cost of not being able to mitigate multiple independent upsets [13]. This technique can tolerate up to one of the three modules being affected by an SEU because a majority voter can be used to mitigate the error without interruption to the computation of the circuit. A prior publication [14] has shown that TMR is a more reliable and area effective compared to techniques such as time shared TMR (TSTMR), quadruple time redundancy (QTR), explicit error correction (EEC), and implicit error correction (IEC). This technique has also shown to be more useful than Hamming codes for protecting small memories [15]. A scrubber circuit like the one specified by Jones, is used to avoid accumulation of errors in the PIPs, LUTs, and other programmable elements on the FPGAs [16].

2.2 Template-Based Architecture Exploration Techniques

While there are several publications that describe the advantages of template-based design space exploration, a subset of papers is reviewed in this section. Mei presents architecture exploration for a reconfigurable architecture template [17]. Architectures can be composed of either a homogeneous set of functional units (FUs) or a combination of multipliers and FUs, with various options for interconnections among them. This description is converted into a graph-based representation, which is followed with architecture exploration using a combination of modulo scheduling and SA techniques. The scheduler also perform the task of placement and routing. It uses a combination of congestion negotiation and simulated annealing techniques.

Jozwiak presents a quality driven template-based architecture synthesis for real-time system on a chip [18]. A semi-automated methodology is used that emphasizes system architecture exploration and synthesis efficiently. An architecture platform corresponding to a given application class and its main modules (processors, data routing network, and memory sub-system) has to be developed in advance based on an analysis of the application class.

Mishra take the approach of exploration of architectures, by specifically looking at pipelined and programmable microprocessors [19]. They allow designers to describe microprocessor architecture in terms of a graph whose nodes represent FUs, registers, ports, and buses. The authors carry out resource constrained scheduling (RCS) where the designer must specify the number of FUs available of each operation type. Ziegler describe the automated mapping of coarse-grained pipelined applications onto FPGA systems [20]. They carry out selective loop unrolling across pipeline stages to balance latencies. Their explorer takes a greedy approach to perform inter-pipeline optimization.

Kienhuis describes the design space exploration of stream-based dataflow architectures [21]. The author considers an architecture framework composed of a set of processing elements (PEs) that communicate with each other via a communication network under control of a global controller. The architecture exploration views the search space in terms of number of PEs, number of functional elements (FEs) in each PE, throughput rate and latency of each FE, etc. The designer has to choose an architecture instance from a template by selecting parameter values such that a feasible design is found, and then allow for a mapping of applications onto that instance.

2.3 Scheduling and Binding Techniques

While there is a rich repository of published scheduling and mapping algorithms, a sample set was selected which is most related to the algorithms proposed in this paper. Heuristic scheduling methods such as List Scheduling [22] and Force-Directed Scheduling (FDS) [23] play a key role in high-level synthesis approaches for architecture design. List Scheduling for instance, attempts to minimize execution time by finding the best schedule of a dataflow graph given a set of resources. FDS on the other hand attempts to derive the smallest set of resources needed to schedule a dataflow graph within a fixed execution window. But neither method takes care of actually mapping graph nodes to resources; thus timing and routing overheads (i.e., registers and multiplexers) are ignored. Researchers have explored variations to the basic list scheduling algorithm, such as dynamic critical path scheduling [24], topological clustering [25], and critical nodes parent trees [26]. These algorithms have been shown to improve the performance of the basic list scheduling algorithm at the expense of increased algorithm complexity.

Nestor does scheduling of nodes by using a simulated annealing loop [27]. The objective is to schedule a control data flow graph (CDFG) satisfying the timing constraint while minimizing the amount of resources used. Each node has a scheduling window with bounds given by the as soon as possible (ASAP) and as late as possible (ALAP) algorithms. Mutations/Alterations to the schedule are done by moving a node only one control step, as they saw no improvement doing this for more than one control step. Any data graph violations are immediately resolved by rescheduling nodes until there are no violations. The cost of a current solution is evaluated on the worst case resource use in any control step and the number of slack nodes required. The costs for a solution are computed incrementally to avoid high costs to compute the fitness of the current solution. The scheduling algorithm's complexity grows linearly with respect to the schedule length.

Purna takes as input a DFG for an application and outputs a set of configurations for FPGAs [28]. The input DFG is partitioned into temporarily interconnected subtasks.

Level-based partitioning is shown to have better performance in execution time compared to using cluster-based partitioning. An ASAP schedule is generated and implemented to run on a separate FPGA that is used for switching in/out the different configurations. Kaul takes an application and divides it into multiple reconfigurable segments [29]. Multiple blocks of data are processed in each reconfigurable segment. An iterative search algorithm is used to break the application into the reconfigurable segments to be run on an FPGA.

So takes an input sequential application description in C and outputs an implementation for an FPGA [30]. Each exploration for a design is synthesized for area and frequency. Based on the result the next exploration is guided to improve performance by doing techniques like loop unrolling, data layout, and scalar replacement. The design space explorer tries to balance computation and memory access rates.

Chaudhuri designed a design space explorer that optimizes a solution for resource usage, schedule length, and clock frequency [31]. The input application cannot have loops. The explorer balances between combining operations into one cycle using chaining and allowing operations to take more than one cycle to try to increase clock frequency. The explorer solves for either RCS or time-constrained scheduling (TCS).

Gu takes an input CDFG and generates out a floorplan using minimum power [32]. An incremental SA floorplanner is used to find the optimal floorplan. Alterations to the floorplan are done by rescheduling nodes that have slack and by changing the number of resources available. Operations are split into multiple cycles until the CDFG meets timings.

Schreiber used a Scheduler and Mapper that assigns operations/nodes to functional units in a template designed to be implemented on an FPGA or Application Specific Integrated Circuit (ASIC) [33]. It takes as input C code and outputs the Hardware Description Language (HDL) to implement the accelerator for the systolic array code. Mapping is done using orthogonal projection and clustering. Scheduling is done using a shifted-linear technique.

Wu solves RCS, which has the goal to minimize the control steps given the constraint of total area, by using an A* search algorithm which prunes the non-promising paths [34].

A priority queue is maintained to contain the search nodes represented by partial schedules. The search starts from an ASAP schedule. Each node that violates the resource constraint is delayed one time step and then this new solution is added to the queue. A heuristic is used to evaluate partially scheduled solutions to determine which solution is closest to the goal. This heuristic has complexity $O(n^2 + c^2)$ where n is the number of nodes and c is the critical path length.

Theodoridis uses integer linear programming (ILP) model to map applications on hardware platforms that consist of microprocessors, ASICs, and FPGAs [35]. Using this model, the authors simultaneously solve scheduling and mapping problems. The computation complexity of ILP algorithms is usually large when compared to heuristic algorithms. As the ILP algorithm proposed in this paper is intended for task graphs, where the number of nodes and edges is small (usually up to 15 tasks and 12 edges), the authors claim that the complexity of this algorithm is acceptable.

Mohanty presented two alternate polynomial-time complexity heuristic algorithms for simultaneous scheduling and mapping of a data flow graph (DFG), optimizing for gate-oxide leakage [36]. These algorithms selectively map the nodes on the non-critical path to instances of pre-characterized resources consisting of transistors of higher oxide thickness and nodes on the critical path to resources of lower-oxide thickness. The first alternative provides flexibility to the designer to provide time constraints. Whereas the second alternative converges to solutions faster as the time constraint is not stringent.

Xu designed a tool to synthesize defect-tolerant architectures for Microfluidic Biochips [37]. The synthesis algorithm (which includes mapping, scheduling, and placement) is based on parallel recombinative simulated annealing algorithm which is a combination of multi-objective simulated annealing and genetic algorithms. Mapping of a node to a resource (i.e., a microfluidic module) is done based on the node's gene value, scheduling is done using list scheduling algorithm, and a greedy algorithm is used for placing the microfluidic modules on the chip.

Sun presented two different exploration algorithms combining pipeline scheduling, module selection, and resource sharing during architecture synthesis [38]. Scheduling is done onto pre-pipelined library elements. The first alternative uses a recursive branch-and-bound algorithm based on ASAP scheduling. The second alternative uses backtracking (unscheduling), based on iterative modulo scheduling. The empirical computational complexities of these algorithms are $O(m^n)$ and $O(n^3\ ln(n))$ respectively, where m is the number of possible implementation options of a node and n is the number of nodes.

Wang proposed an iterative scheduling algorithm based on ant colony optimization [39]. In this algorithm, a collection of agents (ants) cooperate together to search for a solution. Ants generally follow the previous path the other ants have taken, but with a certain probability will pick a different path. Whichever time step has the most ants go through gives the time step the node will be scheduled at. Using experimental results, it was shown that this algorithm outperforms an SA-based TCS algorithm in average area savings. It was also shown that SA algorithm took three to four times more time than the proposed algorithm. However, the SA algorithm used generates a random neighbor solution that may not be valid.

Chapter 3
Hardware Architecture Template for Accelerating Pipelined Simulated Annealing Kernels

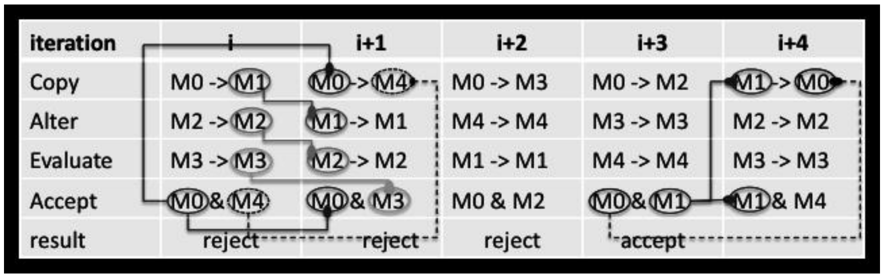
To provide clarity on the architecture exploration process, the hardware architecture template used for SAKs (originally designed by Jonathan Phillips) is briefly reviewed in this chapter. This chapter was leveraged from a section that Aravind Dasu and I wrote for a journal paper. A reasonable assumption is made that the algorithmic flow for SA techniques involves the generation of an initial solution, usually randomly, and evaluated for a score. This initial solution is designated as the current solution until a new one is generated and accepted. SA algorithms usually iterate several thousand times over the five following processes: *Copy*, *Alter*, *Evaluate*, *Accept*, and *AdjustTemperature*. In every iteration, the current solution is copied to a second buffer during the *Copy* process. This copied solution (designated as new solution) is then altered slightly during the *Alter* process. This new solution is then evaluated for a score during the *Evaluate* process. During the *Accept* process, the score of this new solution is then compared against the score of the current solution to determine whether to accept this new solution. A probability value (p), for minimization problems considered in this paper, is computed using eq. (3.1):

$$p = e^{\frac{\triangle S}{T}}, \triangle S = S_{new} - S_{current}, \tag{3.1}$$

where $\triangle S$ is the difference between the score of new solution (S_{new}) and current solution ($S_{current}$), and T represents temperature. The new solution is accepted if: (a) S_{new} is less than $S_{current}$, or (b) the outcome of a uniform random number generator (between 0 and 1.0) is less than p. When the temperature is high, suboptimal solutions are more likely to be accepted. This feature allows the algorithm to escape from local minima as it searches the solution space and attempts to zero in on a close approximation to the optimal solution. The last step in the loop decreases the temperature according to a pre-determined schedule during the $AdjustTemperature$ process. A typical method is to geometrically decrease the temperature by multiplying it with a cooling rate, which is generally a number such as 0.99 or 0.999. The closer the cooling rate is to 1.0, the more times the loop will execute. This results in longer program execution, but also improves the probability of finding the best solution (i.e., the solution that is the closest approximation to the optimal solution). This algorithmic flow allows for variations on how solutions are represented, scores are calculated, solutions are altered, and evaluated, which often are tailored to the problem being solved.

Since classic SA techniques are sequential in nature, pipelined SA (pSAK) versions require a storage system to hold the multiple solutions in the pipeline. This process is illustrated with fig. 3.1. During iteration i, a *Copy* process transfers contents of memory bank-0 (M0) into memory bank-1 (M1) and an *Alter* process performs a random (strictly pseudo-random) alteration of the solution in memory bank-2 (M2). Therefore, its source and destination banks are the same. An *Evaluate* process evaluates the solution in memory bank-3 (M3) over a cost function and stores the score back into M3. An *Accept* process makes a choice between the current solution residing in M0 and the new solution (i.e., it has passed through *Copy*, *Alter*, and *Evaluate* processes previously) residing in memory bank-4 (M4). If it rejects the new solution in M4, the solution in M0 continues to be the currently accepted solution and is used as the source by the *Copy* process in iteration $i + 1$.

Therefore the solution in M0 is copied into M4, overwriting the rejected solution of iteration i. The current solution in M0 is then used to compare with the new solution in



iteration	i	i+1	i+2	i+3	i+4
Copy	M0 -> M1	M0 -> M4	M0 -> M3	M0 -> M2	M1 -> M0
Alter	M2 -> M2	M1 -> M1	M4 -> M4	M3 -> M3	M2 -> M2
Evaluate	M3 -> M3	M2 -> M2	M1 -> M1	M4 -> M4	M3 -> M3
Accept	M0 & M4	M0 & M3	M0 & M2	M0 & M1	M1 & M4
result	reject	reject	reject	accept	

Fig. 3.1: Multiple solution storage approach to illustrate pipelining in pSAKs.

M3 during iteration $i + 1$. However, as shown with iteration $i + 3$, if the new solution in M1 is accepted over the current solution in M0, then in iteration $i + 4$, solution in M1 becomes the current solution and will be used to overwrite the contents of M0. From this illustration it can be observed that to obtain a pipelined behavior in a hardware architecture, it is necessary (conservatively speaking) to design five memory banks that can be accessed concurrently by sub-systems representing the four main processes: *Copy*, *Alter*, *Evaluate*, and *Accept*($CAEA$). The only process not represented by the illustration in fig. 3.1, because it does not use the memory banks, is the *Adjust Temperature* process, which is responsible for adjusting the temperature at the end of every iteration. By pipelining the behavior of a SAK, the quality of the final solution is comparable to that from a non-pipelined version as shown in the results section (see Table 5.4) for various test cases. If the memory banks were implemented as pipeline interlock registers, it would result in the need of the *Copy* process to occur in every process (i.e., you need to copy the entire input pipeline interlock register to the output pipeline interlock register). The advantage of having pointers to the memory banks is that during every iteration the pointers are updated which avoids the need to copy the entire contents every time; however, using pointers requires the use of multiplexers in FPGAs to achieve the required functionality. We use the pointer setup in the template for pipelined hardware accelerator architecture.

Based on this concept, the template for the pipelined hardware accelerator architecture (shown in fig. 3.2) is composed of (a) set of five memory banks, (b) memory multiplexing (data routing network), (c) five data-processing sub-systems ($CAEA$ and *Adjust Temperature* sub-systems), and (d) a kernel controller (responsible for data routing network, signaling new iterations to the sub-systems, and signaling the host processor when the kernel execution is complete). The data processing sub-systems represent a one-to-one mapping of equivalent processes in simulated annealing. The memory banks are used for the purpose of storing solutions to a SA problem.

Parameters that can vary among flavors of kernels are problem size, how the solution is represented, strategy for altering solutions, cost functions to evaluate solutions, rate of

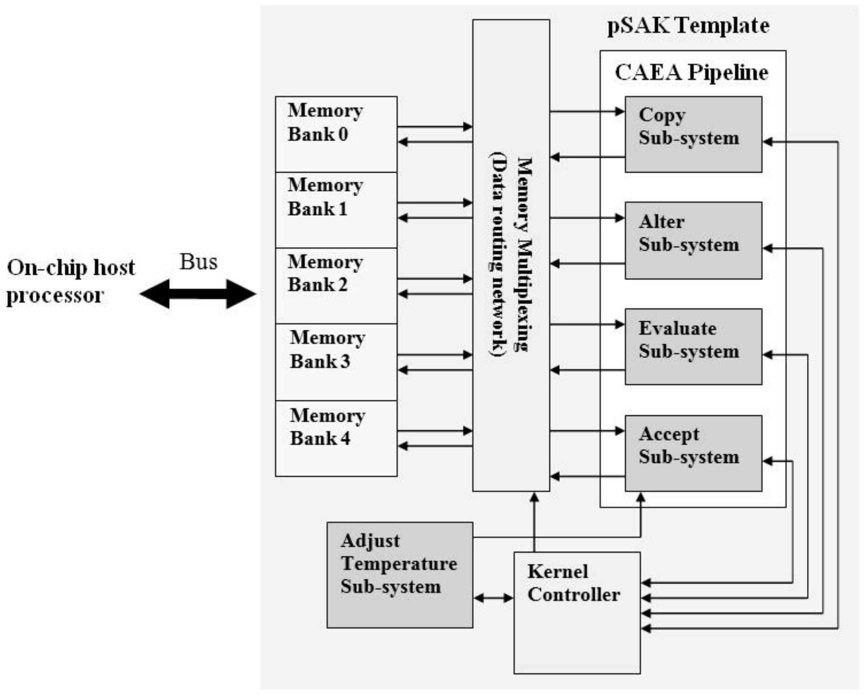


Fig. 3.2: Template for pSAK on an FPGA.

cooling, etc. Some of these features are used to modify parameters in the template used inside the memory banks and memory multiplexing. Other features are used in deriving the micro-architectures of the sub-systems.

The pSAK accelerators are assumed to interact with a host processor on the FPGA via On-chip Peripheral Bus or Processor Local Bus. These bus standards are widely used by the community when Xilinx FPGAs are considered. An assumption is made that a higher-level spacecraft software controller code will reside on and be executed by the host processor. For this paper, this processor is assumed to reside on the MicroBlaze soft-core processor.

The template of the *memory banks* and *memory multiplexing* are shown in figs. 3.3 and 3.4. They are represented as a set of parameterizable VHDL entities. The present version of the template assumes a simple configuration mode for a random access memory (RAM) module: one read and one read/write port. Considering this simple configuration mode for a RAM module, concurrently reading from N address locations requires $\frac{N}{2}$ copies of a single address space (representing a solution) stored in $\frac{N}{2}$ RAM modules. This template consists of five *memory banks*. Each bank can consist of $\frac{N}{2}$ read ports and $\frac{N}{2}$ read/write ports. However, since there are four data-processing sub-systems ($CAEA$) that can concurrently read from any of the five *memory banks* (based on illustration provided earlier with fig. 3.1), this template consists of a series of read-address (ra) multiplexers (four to one) $M_{ra}^{i,j}$, where i refers to the memory bank and j refers to a single RAM module inside the memory bank. These multiplexers in a given bank allow only one of the four data-processing sub-systems to read up to N data points from N address locations, in a given iteration. The data read out of the memory banks are then passed into a set of read-data (rd) multiplexers (five to one) M_{rd}^{i}, where i refers to the multiplexer index. A set of N such multiplexers are dedicated to each of the sub-systems in the $CAEA$ pipeline. The only exception is that two such sets are dedicated to the *Accept* sub-system (but not shown in fig. 3.3 for sake of clarity) to allow reading of solutions (current and new) from different *memory banks*. Note that there are two read-data (rd) ports coming out of each RAM module (where the second rd port comes from the read/write address (rwa)), but are not shown in the figure for sake of clarity.

Writing of data is carried out through the read/write multiplexers $M_{rw}^{i,j}$, where i refers to the memory bank and j refers to a single RAM module inside the memory bank. These multiplexers carry the read/write address (rwa), write enable (we), and write data (wd) lines. Note that all the multiplexers in fig. 3.3 are controlled by the *kernel controller*. To maintain coherency among the $\frac{N}{2}$ copies of a solution in a memory bank when writing, rwa, we, and wd must be the same for all the RAM modules within a memory bank. While this does not allow for writing disparate data into $\frac{N}{2}$ arbitrary addresses concurrently, a minor modification within each RAM module allows for a restricted form of writing to multiple (but contiguous) address locations. To support reading from/writing to multiple address locations, data in a RAM module is distributed across multiple RAM blocks as shown in the template of fig. 3.4. This is similar to how double data rate RAM groups multiple memory banks together to allow for wider reads/writes [40]. This template can currently support access to one, two, or four contiguous locations. The parameters of this template are as

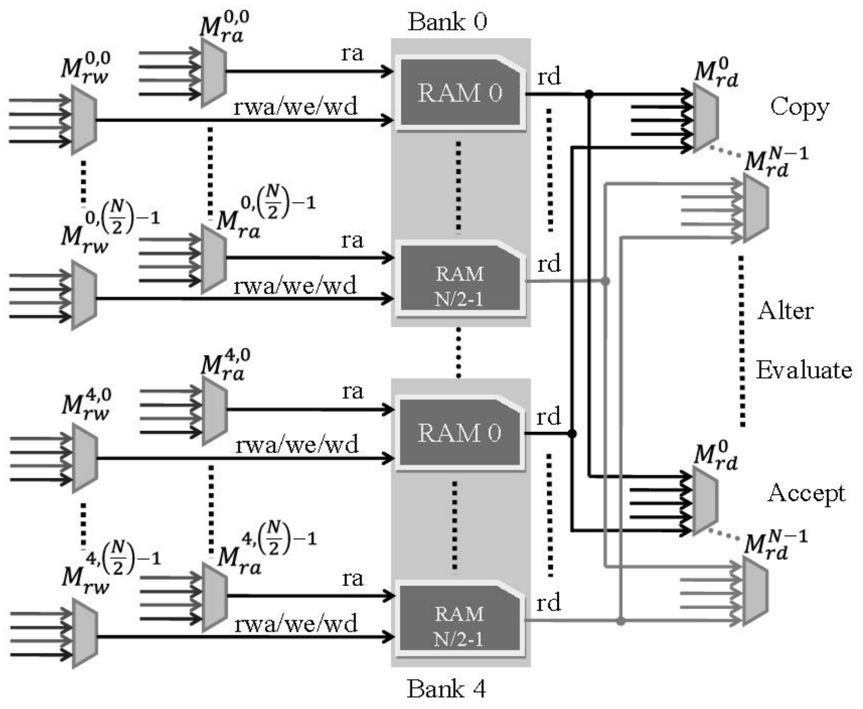


Fig. 3.3: Memory banks and memory multiplexing for pSAK on an FPGA.

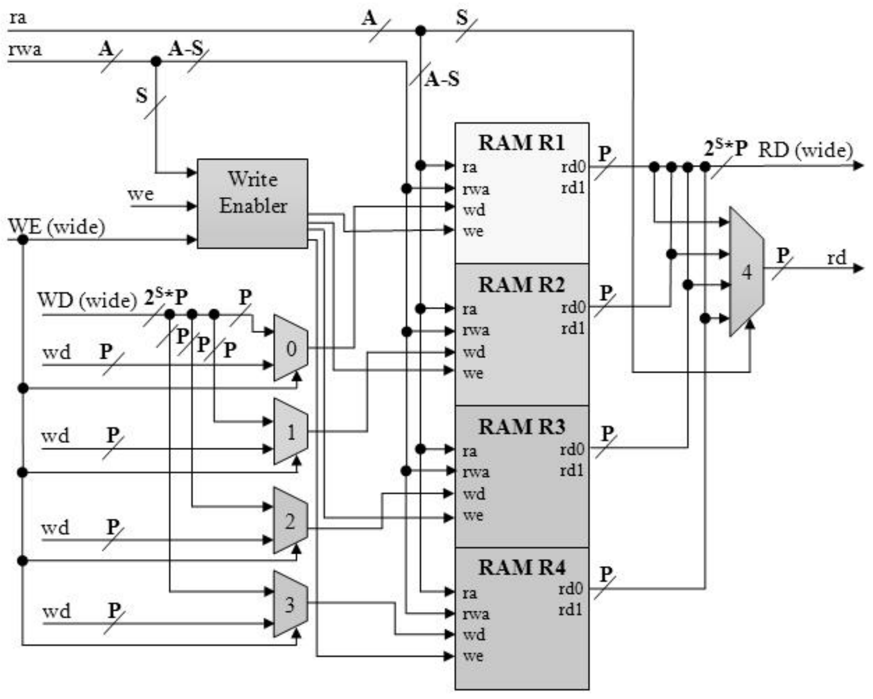


Fig. 3.4: Template of a RAM module to support reading from/writing to one, two, or four contiguous locations.

follows: A is the number of bits used to represent an address, P represents data width in bits (also referred to as a *word* henceforth), and S is the number of bits stripped from an address to enable single/wide word read/writes.

As the data is distributed across multiple RAM blocks (R1-R4), for single word reads, $rd0$ outputs of all RAM blocks are passed through a multiplexer ($multiplexer-4$) controlled by the S bits from ra. Another multiplexer (not shown in the figure for sake of clarity), controlled by the S bits of rwa, is used to select $rd1$ outputs of all RAM blocks when rwa port is used in read mode. However, for wide word reads, $rd0$ outputs of all RAM blocks are concatenated as $RD(wide)$. Similarly $rd1$ outputs are concatenated as another $RD(wide)$, not shown in the figure, when rwa port is used in read mode.

For single word read/writes, an instance of this template is created using only RAM block R1 with S being zero. To allow wide word reading from/writing to two consecutive locations, RAM block R2, $WriteEnabler$, $multiplexers - 0$, and 1 are added to the above instance, with S being one. RAM R1 contains all even address locations and RAM R2 contains all odd address locations. Also, $multiplexer - 4$ (only the first two inputs) is included in this instance to allow for single word reads, as explained earlier. Depending on the value of the S bits stripped from the address rwa, the $WriteEnabler$ module drives the we lines of RAM R1 and R2. On similar lines, to allow wide word reading from/writing to four consecutive locations, RAM blocks R3 and R4, $multiplexers - 2$, 3, and 4 (the last two inputs) are added to the above instance, with S being two. In this case, the $WriteEnabler$ drives the we lines of all the four RAM blocks.

Note that the $RD(wide)$ data lines (shown in fig. 3.4) from all RAM modules in each memory bank also pass through a set of N read-data multiplexers similar to rd data lines (shown in fig. 3.3) for each sub-system, but are not shown in fig. 3.3, for sake of clarity.

The template is designed to allow for various parameters, specific to a target pSAK, to drive the customization. For example, the data width P is determined by the maximum of the number of bits to represent an element in a solution and the number of bits to represent the cost of a solution. The number of address bits needed (A in fig. 3.4) is determined by taking the logarithm of the number of locations required to represent a solution plus one (to store the score of a solution). Other parameters are derived through the process of architecture exploration, specifically the iterative pipeline-latency-balancing (PLB) algorithm described later in Chapter 4. The number of read-data multiplexers (M^i_{rd} of fig. 3.3) required for each sub-system is determined by the number of simultaneous reads for that sub-system. The number of RAMs ($\frac{N}{2}$ in fig. 3.3) for each memory bank and conversely read-address multiplexers ($M^{i,j}_{ra}$ of fig. 3.3) are determined by the sub-system that requires the most simultaneous reads (one multiplexer per read and one RAM per two reads are required).

The templates for sub-systems in the $CAEA$ pipeline are shown in fig. 3.5. The input *step* signal (generated by the *kernel controller*) indicates to a sub-system when a new iteration begins. The output *done* signal (received by the *kernel controller*) is driven by the sub-system when it has completed the given task for that iteration. Figure 3.5(a) shows a common template for *Alter* and *Evaluate* sub-systems that interacts with one memory bank. Currently the tool is set for up to eight parallel reads. So there are eight read ports (ra/rd). The write signals are driven using the ports read/write address (rwa), write data (wd), and write enable (we). Figure 3.5(b) shows a common template for *Copy* and *Accept* sub-systems that interacts with two *memory banks*. Any output not used in a template is driven to ground. This allows the synthesis tool (Xilinx's XST) to optimize away or reduce any components that use the grounded output. This synthesis tool also optimizes or reduces any components that drive any inputs not used in the template. As a note, the *kernel controller* is very simple and does not change across pSAKs, its template consists of a parameterizable VHDL implementation of this controller with the cutoff temperature passed in as a parameter. The *Adjust Temperature* sub-system is similarly implemented as a parameterizable VHDL entity with the parameters being the cooling rate and initial temperature.

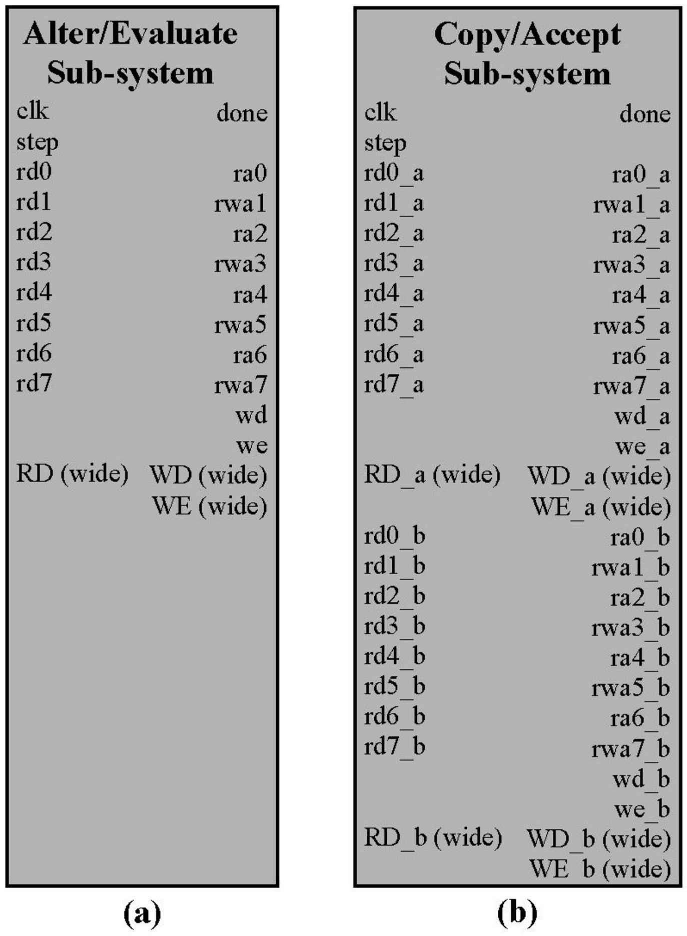


Fig. 3.5: Common templates for sub-systems of the *CAEA* pipeline for interacting with the memory banks.

Chapter 4
Architecture Derivation Methodology

The core contributation of this thesis, namely the architecture exploration methodology, is an enhancement of the prior doctoral disseration of Jonathan Phillips. The starting point for the architecture derivation methodology in our approach is a sequential SAK expressed as a C program constrained to a set of five processes (functions): *Copy*, *Alter*, *Evaluate*, *Accept*, and *Adjust Temperature*. Through the use of existing front-end compiler passes in the GNU C compiler (*gcc*), a CDFG is extracted for each process as well as the constants describing parameters of the kernel such as initial temperature, cooling rate, cut-off temperature, and size of the solution. Then each CDFG is converted into a DFG by unrolling the loops completely, converting conditional constructs (such as if) into predicative execution, etc. The DFGs corresponding to each sub-system in the $CAEA$ pipeline are then passed through an iterative PLB algorithm explained in sec. 4.1.

4.1 The PLB Algorithm

The pipeline-latency-balancing (PLB) algorithm (shown in Algorithm 4.1) first associates an initial latency for the *Copy* sub-system by adding one to the number of events to be scheduled in the kernel. An area cost was not considered because the *Copy* sub-system is just a state machine. The second step is to associate an initial latency and area for the *Accept* sub-system. It always assumes a fixed latency of 56 clock cycles because it is essentially a sequential set of operations supported by the following circuits: an integer subtractor, integer to floating-point convertor, floating-point divider, exponential look-up table, random number generator, and floating-point comparator. When these circuits are implemented using the Xilinx CORE generator (and our custom circuits), the overall latency adds up to 56 clock cycles. Computation of the initial latency (computed by the

Scheduler discussed later in sec. 4.3) and area (computed by the *Mapper* discussed later in sec. 4.4) for the *Alter* and *Evaluate* sub-systems is also done. The original implementations for the sub-systems is used by minimizing area required to implement the sub-system. The PLB algorithm minimizes the different latencies of the sub-systems by targeting the sub-system with the longest latency and allowing it to use more area on the chip to achieve a lower latency. This allows the minimal pipeline latency to be achieved among the different sub-systems while using only the minimal area required to do this.

Before *Loop*1 is executed, the current sub-system with the longest latency is given a token. The area available on the chip is also determined by the area taken by the different sub-systems and updated as the area of the sub-systems is updated. *Loop*1 is exited if a sub-system is given the token twice in a row or the *Accept* sub-system has the token (because it cannot be parallelized to decrease its latency). In *Loop*1 a check is carried out to see if *Copy* sub-system has the token, in which case, the current wide word read/write usage for the *Copy* sub-system is doubled. Wide word read/write usage here refers to reading from/writing to more than one location (i.e., contiguous locations) in one clock cycle by using the template shown in fig. 3.4. Then its latency is updated. However, if either the *Alter* or *Evaluate* sub-system has the token, the *Scheduler* and *Mapper* are invoked to try and reduce latency of the sub-system that has the token, to a value less than the second longest latency. If both the *Scheduler* and *Mapper* produce a valid solution, then the schedule, mapping, latency, and area of the sub-system is updated; otherwise the sub-system that currently has the token retains the token and eventually *Loop*1 is exited. After these checks, the token is passed to the sub-system with the longest (worst) latency. These steps are iterated until the exit condition of the loop is met.

Once *Loop*1 exits, a check is carried out to see either the *Alter* or *Evaluate* sub-system has the token. If yes, then *lowerBound* is initially set to the latency of the sub-system with second longest latency and *upperBound* is initially set to the latency of the sub-system that has the token. Then *Loop*2 is executed whose purpose is to achieve the lowest possible latency for the sub-system that has the token. The first step in *Loop*2 is to invoke the

Algorithm 4.1 PLB algorithm

```
Associate initial latency for *Copy* sub-system
Associate initial latency and area for *Accept* sub-system
Compute initial latency and area for *Alter*, *Evaluate* sub-systems
    [Invoke *Scheduler*, *Mapper*]
Identify sub-system with longest latency and give it a token
**LOOP1:** Do
    if(*Copy* sub-system has token) then
        Double the wide word read/write usage, if possible, and update its latency
    else if (*Alter* or *Evaluate* sub-system has the token) then
        Invoke *Scheduler* to reduce latency of sub-system having the token, to a value less
                than the 2nd longest latency
        if(*Scheduler* produces a valid solution) then
            Invoke *Mapper* with the new schedule from the *Scheduler*
            if(*Mapper* produces a valid solution) then
                Update schedule, mapping, latency, and area of the sub-system that has the
                    token
            End If
        End If
    End If
    Identify sub-system with longest latency and pass the token to it
    Exit Loop1 if new recipient of token is not different from previous recipient Or if
                *Accept* sub-system has the token
**END LOOP1**
if(*Alter* or *Evaluate* sub-system has the token) then
    set *lowerBound* to latency of the sub-system with 2nd longest latency
    set *upperBound* to latency of sub-system that has the token
    **LOOP2**: Do
        Invoke *Scheduler* (for sub-system that has the token) to produce a schedule with a
                latency no more than half-way between *lowerBound* and *upperBound*
        if(*Scheduler* produces a valid solution) then
            Invoke *Mapper* with the new schedule from the *Scheduler*
            if(*Mapper* produces a valid solution) then
                Update schedule, mapping, latency, and area of sub-system that has the
                    token
                Set *upperBound* to latency derived from *Scheduler*
            End If
        End If
        If(*Scheduler* or *Mapper* do not produced a valid solution) then
            Set *lowerBound* to mid-point between previous *lowerBound* and *upperBound*
        End If
        Exit Loop2 if *upperBound* and *lowerBound* have a difference of less than 2 clock
            cycles
    **END LOOP2**
End If
Merge *Alter* and *Copy* sub-systems if viable
```

Scheduler and *Mapper* for the sub-system that has the token, to produce a circuit with a desired latency of no more than halfway between *lowerBound* and *upperBound*.

If the *Scheduler* and *Mapper* produce a valid solution then *upperBound* is set to the latency derived from the *Scheduler*; otherwise, *lowerBound* is set to the mid-point between the previous *lowerBound* and *upperBound*. These steps are iterated until the difference of *lowerBound* and *upperBound* is less than two clock cycles.

The last step in the PLB algorithm is to see if *Alter* and *Copy* sub-systems should be merged. The criteria for merging, are as follows: (a) neither of these sub-systems should be in possession of the token, and (b) their combined latency should be less than that of either *Evaluate* or *Accept* sub-system. Since merging can sometimes result in an unfavorably large joint latency, the option of doubling the wide word read/write usage for the *Copy* sub-system to bring down the joint latency is explored.

Before explaining the *Scheduler* and *Mapper* algorithms, the resource estimation technique is introduced in sec. 4.2.

4.2 Resource Estimation

Since the design space exploration needs to evaluate large number (ex., just under 100,000,000 for one invocation of the *Scheduler* for the dgv500 test case used in Chapter 7) of circuits (through the PLB, *Scheduler* and *Mapper* algorithms), there was a need to quickly obtain approximate estimations of the area usage of circuits, without having to go through time consuming Electronic Design Automation (EDA) based synthesis, translate, map, place and route (P&R) tool flow. Therefore to facilitate this quick estimation process, the components used in the *CAEA* sub-systems are classified into two categories: (a) those invoked using Xilinx CORE Generator, and (b) those composed of basic building blocks (such as integer adders, modulo operators, comparators, multiplexers, registers, etc.). For circuits of type ‘a’, post P&R area estimates are obtained from Xilinx CORE Generator. However for circuits of type ‘b’, the following technique described next is used.

First, post P&R usage is obtained of basic building blocks in terms of device primitives (LUTs, FFs, and DSPs) for some data widths (usually at regular intervals of four). Curve fitting tools in MATLAB is used to obtain polynomial expressions (up to 5^{th} order) to interpolate an estimate of post P&R device primitive usage for other data widths. Validate of our estimation technique is done by estimating device primitive usage for various circuits (i.e., various data widths and combinations of basic building blocks) and compare with the actual post P&R values of the implemented circuits, as shown in Tables 4.1 and 4.2. This is similar to estimations done by Memik [41] and by Zaretsky [42]. However, unlike the peer publications, this technique is not extended to clock frequency estimation.

Table 4.1 shows the estimated and actual values for a circuit composed of an integer adder followed by an integer multiplier. The estimations are done by combining the individual resource estimations of the integer adder and integer multiplier. There is only an error of 15.4% for a data width of four for LUT usage. Table 4.2 shows the estimated and actual values for a circuit composed of an integer adder, integer multiplier, and three 2:1 multiplexers. The estimation of LUTs, FFs, and DSPs is computed by summing the individual resource estimations from each building block. There is only error as high as 16% for LUT usage for lower data widths. However, the kernels considered for testing required higher data widths (> 10), for which our estimation technique is observed to be fairly reliable with errors less than 5%.

The heterogeneity of modern FPGAs made it necessary for us to modify traditional techniques of resource estimation (similar to Bilavarn [43]), by considering search space options that can dynamically estimate costs of individual solutions in terms of device primitives such as LUTs, FFs, DSPs, and BRAMs, in the *Scheduler* and *Mapper* algorithms. Let us consider a sub-set of 16-bit integer components (adder, multiplier, and absolute value)

Table 4.1: Estimated vs. actual device primitive usage for a circuit composed of an integer adder followed by an integer multiplier.

Data	LUT Usage			FF Usage			DSP Usage		
Width	Estimate	Actual	% Error	Estimate	Actual	% Error	Estimate	Actual	% Error
4	11	13	15.4	8	8	0	0	0	0
8	8	8	0	8	8	0	1	1	0
12	12	12	0	12	12	0	1	1	0
16	16	16	0	16	16	0	1	1	0
24	24	24	0	48	48	0	3	3	0

Table 4.2: Estimated vs. actual device primitive usage for a circuit composed of an integer adder, integer multiplier, and three 2:1 multiplexers.

Data	LUT Usage			FF Usage			DSP Usage		
Width	Estimate	Actual	% Error	Estimate	Actual	% Error	Estimate	Actual	% Error
4	21	25	16	8	8	0	0	0	0
8	34	32	6.3	8	8	0	1	1	0
12	48	48	0	12	12	0	1	1	0
16	64	64	0	16	16	0	1	1	0
24	97	96	1	49	49	0	3	3	0

implemented on a Xilinx Virtex 4 FPGA. The area requirements of these three components (in terms of device primitives excluding BRAMs) are shown in Table 4.3.

For example, looking at which one of these components is cost effective in terms of area usage, for a given set of device primitives, it is not always clear. The integer adder (*iadd*_16*bit*) and absolute value (*iabs*_16*bit*) use only LUTs and FFs; whereas, the integer multiplier (*imul*_16*bit*) uses only a DSP. Therefore, a weighted sum of device primitives (WSDP) as a unified unit of currency is proposed to evaluate the area cost of components mapped onto FPGAs. WSDP for any component/resource (R) is computed using eq. (4.1):

$$R = \sum_i \left\{ \begin{array}{ll} \frac{n_i}{p_i} & if \quad n_i \le p_i \\ \infty & otherwise \end{array} \right\} \; \forall \; i \in \{LUT, \; FF, \; DSP, \; BRAM\}, \tag{4.1}$$

where n_i is number of device primitives of type i needed to implement a virtual resource, and p_i is number of device primitives available. Note that if adequate device primitives of any type are not available, the associated weight is taken as infinity (implying that a particular resource cannot be implemented). Table 4.4 re-expresses the three components of Table 4.3, for different sets of available device primitives, in terms of WSDPs. The lowest-cost implementation in each set is shown in bold. Note that, as *iabs*_16*bit* uses more number of LUTs and same number of FFs than *iadd*_16*bit*, it would not ever be chosen over

Table 4.3: Different integer components implemented on a Xilinx Virtex 4 FPGA.

Component	LUT Usage	FF Usage	DSP Usage
iadd_16bit	16	16	0
imul_16bit	0	0	1
iabs_16bit	30	16	0

Table 4.4: WSDPs for implementations of Table 4.3.

Component	LUT Usage	FF Usage	DSP Usage	WSDPs when available LUTs/FFs/DSPs are 1000/1000/20	100/100/5	16/16/1
iadd_16bit	16	16	0	**0.032**	0.32	2
imul_16bit	0	0	1	0.05	**0.2**	**1**
iabs_16bit	30	16	0	0.046	0.46	∞

the latter. However, it can be chosen over *imul_16bit*, for certain sets of device primitives (see column 5 of Table 4.4). Also, note that *iabs_16bit* cannot be implemented under one set of available device primitives (see column 7 of Table 4.4).

4.3 The Scheduler

The *Scheduler* algorithm (shown in Algorithm 4.2) takes as inputs, a DFG for the sub-system, desired latency for the schedule, available resources on the chip, and resource's area and latency required to implement the different arithmetic, logic, and supporting (ex., registers, multiplexers) components. The output of the *Scheduler* is a schedule (i.e., start times for all nodes in the DFG) and the achieved latency. The next step in the algorithm is to initialize the *temperature* (this is different than the temperature in the SAK being derived). This is set to the product of the number of nodes in the DFG and the desired latency, with the result of this product being raised to the power 0.8. This *temperature* was found to work well across different problem sizes. The ASAP and ALAP scheduling window for all nodes in the DFG is set next. The initial solution derived by the *Scheduler* is the ASAP schedule, thus ensuring that the algorithm begins with a schedule without any data graph violations (DGVs). A DGV occurs when a node's start time is scheduled at a time earlier than its predecessor's start time plus the latency of the predecessor's operation. The *currentScore* is initialized with the score of this initial solution. Derivation of this score is described later using eqs. (4.2), (4.3), and (4.4). Lastly, the best solution is set to be the initial solution and the *bestScore* to be the *currentScore*. Then the *Loop* is entered into now that the current solution, best solution, *currentScore*, and *bestScore* have been initialized where they will be used in comparisons and potentially updated.

The first step in the *Loop* is to randomly pick one node and assign a new random start

Algorithm 4.2 *Scheduler* algorithm

```
Input: DFG, desired latency, available resources on chip, resource area and latency
Output: operation schedule, achieved latency
Initialize temperature, ASAP, and ALAP for each node
Set the initial solution (ASAP schedule) and currentScore (score of initial solution)
Set the best solution (to initial solution) and bestScore (to currentScore)
Loop: Do
    Choose a random node and assign a new start time within its ASAP to ALAP window
    For each predecessor and successor of the random node
        If(data graph violation: DGV)
            Assign a new start time within its ASAP to ALAP window that will not incur
                a DGV
        End if
    End For
    Incrementally calculate the score (nextScore) of this solution
    $p = e^{\frac{currentScore - nextScore}{temperature}}$
    If ($nextScore \leq currentScore$) or (RandomFloat(0 to 1) $< p$))
        Accept all changes to start times of the nodes and update currentScore with
            nextScore
        If ($currentScore \leq bestScore$)
            Update the best solution with current solution and update bestScore with
                currentScore
        End If
    Else
        Do not accept changes to start times of nodes (i.e. undo changes)
    End If
    temperature = temperature * coolingRate
    Exit Loop if temperature is less than or equal to cutoff Temperature
End Loop
If best solution is not valid, do not return any solution
```

time within that node's ASAP to ALAP window. Similar to Nestor, the need to compute the score for DGVs is avoided by updating the start times (to a new time within the ASAP to ALAP window), as necessary, of the predecessors and successors of the random node until there are no DGVs. The next step is to evaluate this solution for a score (*nextScore*) [27].

A solution is evaluated according to eqs. (4.2), (4.3), and (4.4). Any time a gap of one or more cycles exists between completion of a parent node and commencement of a child node, registers are needed. Since mapping has not been carried out, the number of registers that are required (*#regs*) is estimated pessimistically (i.e., register sharing is not considered) according to eq. (4.2):

$$\#regs = \begin{cases} \sum_{i=0}^{n-1} \sum_{j=0}^{p_i-1} s_i - (s_j + L_j) & if \quad s_j + t_j < s_i \\ 0 & otherwise \end{cases}, \tag{4.2}$$

where n is the number of nodes in the graph, p_i is the number of parents of node i, s is the start time, and L_j is the latency of node j. Our tool currently uses only one implementation for every operation (i.e., different latencies or resource implementations are not considered). An estimate of the area of a solution (in WSDP units) is computed using eq. (4.3):

$$R_{cummulative} = \sum_{j=0}^{V_R-1} C_j R_j, \tag{4.3}$$

where V_R is the number of distinct virtual resource types (components). A virtual resource type here refers to basic arithmetic and logic components, such as integer adder, floating-point multiplier, floating-point comparator, etc., as well as registers. C_j is the maximum number of concurrent instances of each of the V_R virtual resource types (with the exception of registers, for which $C_j = \#regs$) and R_j is a weighted resource value in WSDPs from eq. (4.1) using the input area available on the chip and the area used by the different resources. In the *Scheduler* the area contribution by multiplexers required for resource sharing is not considered.

From the memory template described in fig. 3.3, each memory bank (with four read ports and four read/write ports i.e., N=8) has two limits depending on whether a write is enabled or not. The first limit is to allow for up to eight simultaneous reads and no writes in one clock cycle. The number of violations of this limit is accumulated in P_{rw}. The second limit on the memory bank is four simultaneous reads and one write in one clock cycle. The number of violations of this limit is accumulated and added into P_{rw}. Based on the estimated WSDPs for virtual resources ($R_{cummulative}$) and the number of violations (P_{rw}), a score for the schedule ($score_{schedule}$) is calculated using eq. (4.4):

$$score_{schedule} = R_{cummulative} + 100P_{rw}, \tag{4.4}$$

where P_{rw} is used to penalize the score with a factor of 100 in order to strongly discourage accepting the current solution. This is because the *Scheduler* algorithm is a minimization problem where the lower the score is better. The *nextScore* in the *Loop* is initialized using *currentScore*, and is then incrementally updated (as was done by Nestor) based on the changes in the schedule of the nodes. Incremental updating reduces the complexity of computing C_j of each of the different resource types from $O(n)$ to $O(L)$ where n is the number of nodes and L is the desired latency.

Once the score of the solution is calculated according to eq. (4.4), the next step in the *Scheduler* is to determine if these changes should be accepted. An acceptance probability (p) is first generated as shown in the *Scheduler* (Algorithm 4.2). There are two criteria for the changes to the schedule to be accepted: (a) the new solution's score (*nextScore*) has a score lower than the currently accepted solution's score (*currentScore*), or (b) the outcome of a random number generator (between 0 and 1.0) is less than the probability (p).

If the changes are accepted then the start times of all the nodes that were changed are updated. The *currentScore* is also updated with the value from *nextScore*. If the currently accepted solution's score (*currentScore*) is less than or equal to the best solution's score (*bestScore*), then the best solution and score (*bestScore*) are updated with the currently accepted solution and associated score (*currentScore*), respectively. If neither criterion was met, the changes to the solution are not accepted (undone).

The *temperature* is updated by multiplying it by a predefined cooling rate. The *Loop* is exited if the temperature is less than a predefined cutoff temperature. A final check is done to see if the best solution is valid (i.e., it does not use too many concurrent read/writes). If no valid solution is found, then no solution is returned. If a valid solution is found then the best solution is returned with its associated latency. Note that the associated latency of the solution can be less than (but never more than) the desired latency (as lower latency can save register usage compared to the desired latency).

4.4 The Mapper

The *Mapper* algorithm (shown in Algorithm 4.3) takes as input a DFG for the subsystem, the schedule of the nodes (output of the *Scheduler*), available resources on the chip, and resource's area and latency required to implement the different arithmetic, logic, and supporting (ex., registers, multiplexers) components. The output of the *Mapper* is the resource bindings to specific components (such as integer adder, floating-point multiplier, floating-point comparator, etc.) for each of the nodes in the DFG. The next step in the algorithm is to initialize *temperature*. The initial *temperature* is set to the number of nodes in the DFG raised to the power 0.6, because this *temperature* was found to work well across different problem sizes. In the next step the number of components of each operation type is set to the maximum number of concurrent instances required in any clock step for that operation type. A way to reduce the area for support units (i.e., multiplexers and registers) can sometimes be achieved by allowing more components than the worst case needed in any clock cycle; however, in our case a decrease in area performance was observed by doing so. This could partly be due to the fact that allowing more components increases the design search space which was not accounted for by giving our *Mapper* more time to run. Note that operation chaining (i.e., having two or more operations combined without having a register in between) is not considered. The initial solution is obtained by going through the nodes one by one and assigning a lowest numbered component that is not currently mapped in a given clock cycle. The *currentScore* is initialized with the score of this initial solution. Derivation of this score is described later using eq. (4.5). Lastly, the best solution is set to be the initial solution and the *bestScore* to be the *currentScore*.

The first step in the *Loop* is to randomly pick one node and bind it to a randomly selected component of the same operation type. If the randomly selected component was already bound to another node in the same control step, then the bindings of these two nodes are swapped in relation to the bindings before the random node got the new target component to bind to (i.e., the random node gets the new component binding and the previous node bound to that component gets the previous component binding of the random node). This avoids any component over-utilization (i.e., two nodes assigned to the same component in a given clock step) that can occur in the *Mapper*. The next step is to evaluate this solution for a score (*nextScore*).

The process of evaluating the score of a solution in the *Mapper* is more complex than that in the *Scheduler*. The *Mapper* must be cognizant of total circuit size (components and support units). Recall, the *Scheduler* estimated the number of support units (registers only) because the component binding information was not available. Now that this information is available, the *Mapper* calculates the exact number of support units (registers and multiplexers) required to allow the resource sharing on the component binding given by the solution. A multiplexer is required on an input for a component needs inputs from two different components during different time steps (ex., an adder needs the input from another adder at control step 3 and from a multiplier at control step 6). Multiplexers are also required if a component consumes an output from a component at different latencies (ex., an adder consumes during control step 3 from a multipler whose result is available at control step 2 (needs one delay register to hold output from multiplier before consumption), and also consumes during control step 6 from the same multiplier whose result is available at control step 4 (needs two delay registers to hold output from multiplier before consumption)). A register is used to implement the different delay registers needed to hold an output before it is consumed by the destination component from a source component. Note that wider multiplexers have an impact on maximum clock frequency and area usage. Therefore, currently the tool supports multiplexers up to 16 inputs. Based on the WSDPs

Algorithm 4.3 *Mapper* algorithm

```
Input: DFG, schedule of nodes, available resources on chip, resource area and latency
Output: resource binding for nodes, achieved area
Initialize temperature, and number of components of each operation type
Set initial solution (greedy assignment solution) and currentScore (score of initial solution)
Set the best solution (to initial solution) and bestScore (to currentScore)
Loop: Do
    Choose a random node and bind it to a new component within its operation type
    If(another node was already bound to the new component during the same control
        step)
        Swap the component bindings of the two nodes
    End if
    Incrementally calculate the score (nextScore) of this solution
    $p = e^{\frac{currentScore - nextScore}{temperature}}$
    If ($nextScore \leq currentScore$) or (RandomFloat(0 to 1) $< p$))
        Accept changes to component bindings of the random node
        If(swapping of component bindings of the two nodes occurred)
            Accept changes to component bindings of the swapped node
        End If
        Update currentScore with nextScore
        If ($currentScore \leq bestScore$)
            Update the best solution with current solution and update bestScore with
                currentScore
        End If
    Else
        Undo changes to component bindings of the random node
        If(swapping of component bindings of the two nodes occurred)
            Undo changes to component bindings of the swapped node
        End If
    End If
    temperature = temperature * coolingRate
    Exit Loop if temperature is less than or equal to cutoff Temperature
End Loop
If best solution is not valid, do not return any solution
```

for components ($R_{cummulative}$) and the total number of multiplexers requiring more than 16 inputs (P_m), a score for the mapping ($score_{map}$) is calculated using eq. (4.5):

$$score_{map} = R_{cummulative} + 100P_m, \tag{4.5}$$

where P_m is used to penalize the score with a factor of 100 in order to strongly discourage accepting the current solution. Note that $R_{cummulative}$ is from eq. (4.3), with the exception that this estimate includes the exact number of registers and multiplexers required.

The $nextScore$ in the *Loop* is initialized using $currentScore$, and is then incrementally updated based on the new binding(s). A data structure (shown in fig. 4.1) is used for incrementally updating the number of registers and multiplexers at the inputs of each component.

Each component has up to four input ports. Each port can require registers and multiplexers in order to process data from different components correctly (tracked by *Number Of Multiplexers* and *Number of Registers*). To accurately keep track of nodes that use the same bus between two components (i.e., this encourages that two edges in the DFG that go between two matching source and destination operation types with the same time delay before consumption would share the same bus, source, and destination components), additional information is stored in a component. A component (*Component*) tracks other components (*Source Component*) that are feeding data into any of its inputs (*Input Port*). A *Source Component* can have different number of the delay slots (i.e., registers) that are needed to hold the data before it is consumed by this component (*Component*). This is tracked by the *Source Component* in the *Delay Slots* array. Registers and multiplexers that follow an output port of component are already tracked by the input ports of the component(s) that consume that output.

Without using the above data structure and without incrementally updating the score, the complexity of computing the register and multiplexer area usage was $O(n\ logn)$ where n is the number of nodes in the DFG. By using the above data structure itself, the complexity was reduced to $O(n)$. The complexity was further reduced to $O(1)$ by using the above data

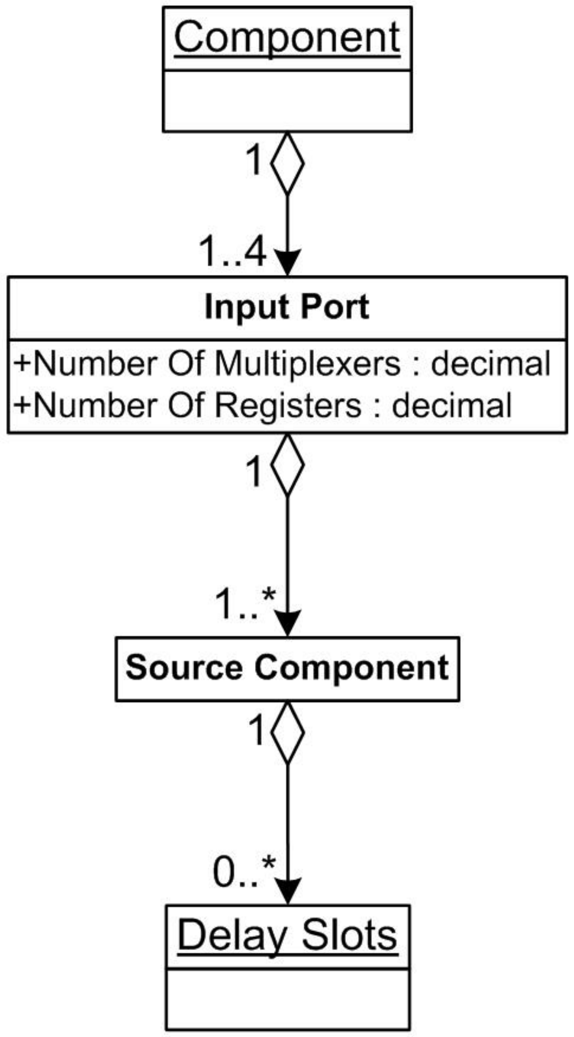

Component
1
1..4
Input Port
+Number Of Multiplexers : decimal
+Number Of Registers : decimal
1
1..*
Source Component
1
0..*
Delay Slots

Fig. 4.1: Data structure used for keeping track of component bindings.

structure and incrementally updating the score. Because this score calculation runs the number of iterations (I) of the *Loop*, in relation to the *Loop* the complexity was reduced from $O(I * n\ logn)$ to $O(I)$.

Once the score of the solution is calculated according to eq. (4.5), the next step is to determine if these changes should be accepted. An acceptance probability (p) is first generated as shown in the $Mapper$ (Algorithm 4.3). There are two criteria for the changes to the mapping to be accepted: (a) the new solution's score ($nextScore$) has a score lower than the currently accepted solution's score ($currentScore$), or (b) the outcome of a random number generator (between 0 and 1.0) is less than the probability (p).

If the changes are accepted then the changes in the resource bindings of the nodes are updated and the $currentScore$ is updated with $nextScore$. If the currently accepted solution's score ($currentScore$) is less than or equal to the best solution's score ($bestScore$), then the best Solution and score ($bestScore$) are updated with the currently accepted solution and its associated score ($currentScore$), respectively. If neither criterion was met, the changes to the solution are not accepted (undone).

The *temperature* is updated by multiplying it by a predefined cooling rate. The *Loop* is exited if the temperature is less than a predefined cutoff temperature. A final check is done to see if the best solution is valid (i.e., it does not use too many registers/multiplexers, or any multiplexers requiring more than 16 inputs). If no valid solution is found, then no solution is returned. If a valid solution is found then the best solution is returned with the associated area of that solution.

4.5 FPGA Architecture Generation

Once the PLB algorithm has finished, then micro-architectures for the *Alter* and *Evaluate* sub-systems in the $CAEA$ pipeline are generated in terms of a simple hardware intermediate format (HIF), which is essentially a structural representation of the micro-architectures. These HIF files are then translated using a tool into VHDL files. *Copy*, *Adjust Temperature*, *Accept* sub-systems, *kernel controller*, and *memory multiplexing* are converted to synthesizable problem specific VHDL files. Some components used in these

VHDL files are created using Xilinx CORE Generator (i.e., integer divider, floating-point comparator, floating-point divider, and integer to floating-point convertor).

A custom linear feedback shift register (LFSR) is used for random number generation. The LFSR is initialized with a seed value (any value except all bits being a '1'). LFSR works by shifting the register contents one bit position (either left or right) and inserting a new bit in the empty bit position. This new bit is generated using a linear function of the previous register contents. As an example, the feedback polynomial used for a 15-bit random number is shown in eq. (4.6):

$$x^{15} + x^{14} + 1, \tag{4.6}$$

where x^n is the bit value at the n^{th} bit position in the LFSR (indexed from 1 to n). As the target platforms for pSAKs are radiation hardened Virtex-4 FPGAs, which still are not completely immune to single event upsets (SEUs) [44], there is a need for invoking a reliable fault mitigation circuit design tool. Therefore, the Xilinx TMR tool (XTMR) is used to convert the design obtained into TMR (triple modular redundancy) form to offer protection against SEUs. This is a fairly straightforward process, and hence the discussion is out of scope of this paper.

Chapter 5
Results

In order to compare the performance of the FPGA designs generated using our approach against using microprocessors, software versions of several SAKs was ported onto a cycle accurate emulator of the PPC 750 [45]. This processor was chosen because it is architecturally equivalent to a state of the art space based microprocessor (BAE Systems RAD 750 [46]) that has a floating-point unit and runs at 200 MHz. Three types of SAKs for testing were choosen: data graph violations (dgv), graph coloring (gc), and traveling sales person (tsp). For each type of SAK, problem/event sizes of 100, 300, and 500 were done resulting in nine test cases. Note that all nine test cases are minimization problems (i.e., trying to achieve the smallest score possible).

Table 5.1 shows detailed results of these test cases on the FPGA and PPC 750. Power estimations for the FPGA were done using Xilinx's *XPower* tool. The BAE RAD 750 requires 5 Watts of power [47]. Energy was obtained by multiplying the power with the time required for the kernel to complete. Energy savings of 99% (on average) was obtained for all the FPGA designs. Not surprisingly, the FPGA designs (despite being clocked up to 115 MHz) also outperform the PPC 750 with a speedup of over 50x. This is due to the pipelined nature of the architectures generated (pSAKs) and the inherent parallelism offered by the FPGA. This makes FPGAs superb candidates for space-borne autonomous mission planning and scheduling.

The final results of the sub-systems after pipeline balancing are shown in Table 5.2. As the number of events to be scheduled increased, the gap between the latency of the *Accept* sub-system and the other sub-systems in the $CAEA$ is widened. Table 5.3 shows the time it took the tool that implements the proposed architecture derivation methodology and generate synthesizable VHDL files, the number of nodes that were being scheduled and

Table 5.1: Device primitive usage, time to complete, power, and energy required for the FPGA based pSAKs compared to PPC 750 based SAKs.

Problem	FPGA								PPC 750 (200 MHz, 5 Watts)		FPGA vs. PPC 750
	LUTs	FFs	DSPs	BRAMs	Max Freq. (MHz)	Time To Complete (seconds)	Power (Watts)	Energy Required (Joules)	Time To Complete (seconds)	Energy Required (Joules)	Speedup
dgv100	14787	10224	15	165	105.9	0.153	2.454	0.376	8.85	44.2	57.8
dgv300	17529	12258	15	165	104.9	0.420	2.693	1.13	31.96	159.8	76.2
dgv500	22545	16983	15	165	108.4	0.663	2.866	1.9	43.28	216.4	65.3
gc100	14244	9948	15	165	112.9	0.091	2.652	0.242	8.22	41.1	89.9
gc300	19872	10572	15	315	71.3	0.325	3.083	1.003	24.57	122.9	75.5
gc500	24786	17493	15	315	70.3	0.529	3.209	1.699	40.97	204.9	74.4
tsp100	14100	10125	15	165	115.4	0.099	2.527	0.25	8.71	43.6	88.0
tsp300	24468	27165	15	165	111.7	0.468	2.784	1.304	25.11	125.6	53.6
tsp500	21897	21111	15	315	64.3	0.760	3.224	2.449	41.58	207.9	54.7

Table 5.2: Final latencies (measured in number of clock cycles) of the different sub-systems for the nine test cases.

Problem	Copy	Alter	Copy/Alter Merged	Evaluate	Accept	Longest Latency
dgv100			88	88	56	88
dgv300			194	239	56	239
dgv500			292	390	56	390
gc100	51	43		55	56	56
gc300			118	126	56	126
gc500			168	202	56	202
tsp100	51	45		62	56	62
tsp300			197	284	56	284
tsp500			150	265	56	265

mapped for the *Alter* and *Evaluate* sub-systems, and the number of iterations of loops in the PLB algorithm (*Loop*1 and *Loop*2 in fig. 4.1). The tool ran on an AMD Athlon 64 X2 Dual Core Processor 5200+ (2.61GHz) with one gigabyte of RAM. The longest the tool took to complete was just under seven hours in the 'tsp500' problem.

Figure 5.1 visually illustrates the working of the PLB algorithm for the 'tsp100' problem. The latency is shown in the y-axis with a logarithmic scale. The x-axis markings separate out the different iterations inside *Loop*1 and *Loop*2 and the initial solution generations of the *Alter* and *Evaluate* sub-systems. Latencies of schedules are recorded and plotted (shown in between different x-axis marks ($X\#$) in fig. 5.1) at intervals of 10% of the total number of iterations of the *Scheduler*. There are times where the *Scheduler* deviates away from a lower latency. This is because the *Scheduler* optimizes for area and

Table 5.3: Time to run the tool, number of nodes for *Alter* and *Evaluate* sub-systems, and the number of iterations of loops in the PLB algorithm (*Loop*1 and *Loop*2 in fig. 4.1).

Problem	Time to run the Tool (minutes/hours)	Number of *Alter* nodes	Number of *Evaluate* nodes	Number of iterations (*Loop*1 + *Loop*2)
dgv100	13.76 m	7	2851	8
dgv300	1.28 h	7	8551	8
dgv500	6.38 h	7	14251	11
gc100	3.16 m	7	1651	4
gc300	27.34 m	7	4951	10
gc500	1.56 h	7	8251	11
tsp100	10.36 m	10	1783	9
tsp300	1.25 h	10	5383	10
tsp500	6.83 h	10	8983	10

not latency as discussed earlier in sec. 4.3. In reference to the PLB algorithm in fig. 4.1, x-axis marks $X0$ and $X1$ generate an initial solution for *Alter* and *Evaluate* sub-systems, $X2 - X4$ iterate through *Loop*1, $X5 - X9$ iterate through *Loop*2, and $X10$ shows the final latencies after the PLB algorithm is complete.

Figure 5.2 shows intermediate scores for the $Scheduler$ and $Mapper$ when invoked for the *Alter* and *Evaluate* sub-systems during x-axis marks $X0 - X2$, and $X4 - X9$ of fig. 5.1. Figure 5.2 also shows the maximum number of read ports used for the memory banks during the schedule process (i.e., when the maximum number of read ports is violated, a penalty of 100 is assigned as discussed earlier in sec. 4.3). $X3$ and $X10$ are not shown in fig. 5.2 because they do not use the $Scheduler$ or $Mapper$.

To avoid staying at the local minimum given by the original solution, the *Scheduler* and $Mapper$ algorithms probabilistically accept worse solutions at the initial iterations of the algorithms (which can be seen for the marks $X5$ and $X6$ from fig. 5.2). Slowly the *Scheduler* and $Mapper$ algorithms start moving towards the global minimum (i.e., worse solutions are not accepted as often) as can be seen by the decreases in the scores during each invocation. The $Mapper$ was called only once for the initial mapping during $X0$ because there was only one component of each operation type, and hence component swapping was not possible to generate more solutions. For x-axis marks $X4$, and $X7 - X9$, the *Scheduler*

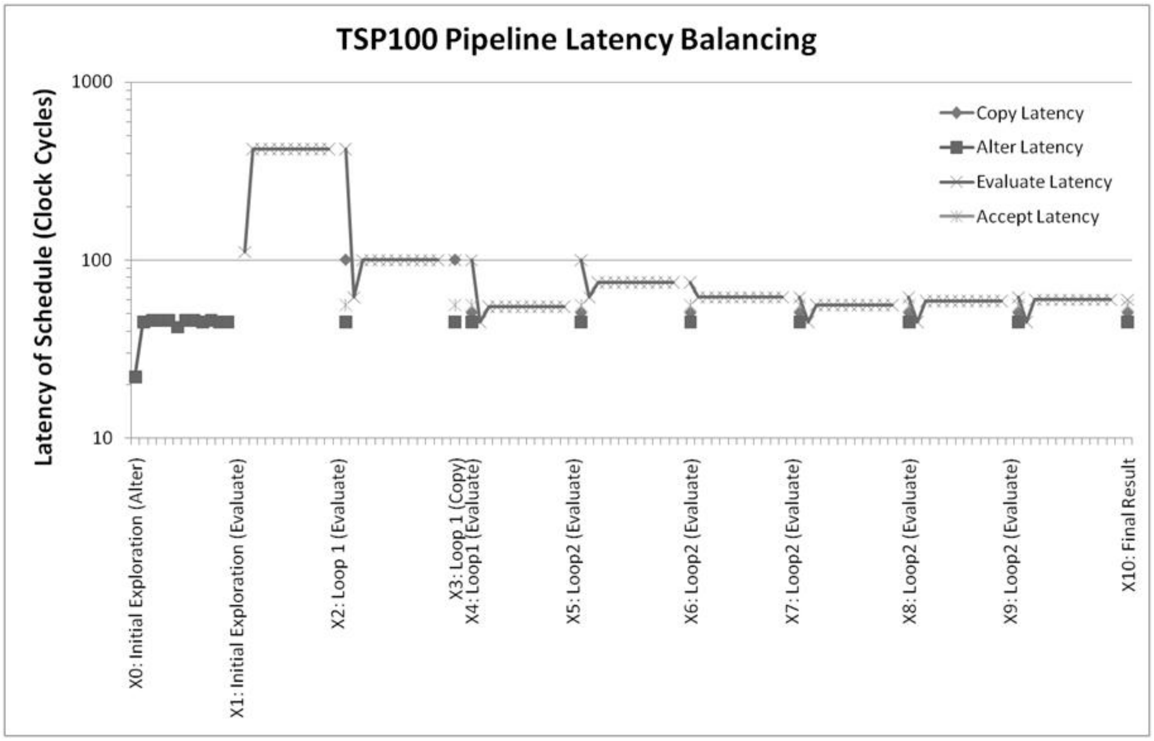


Fig. 5.1: Working of the PLB algorithm on the tsp100 problem.

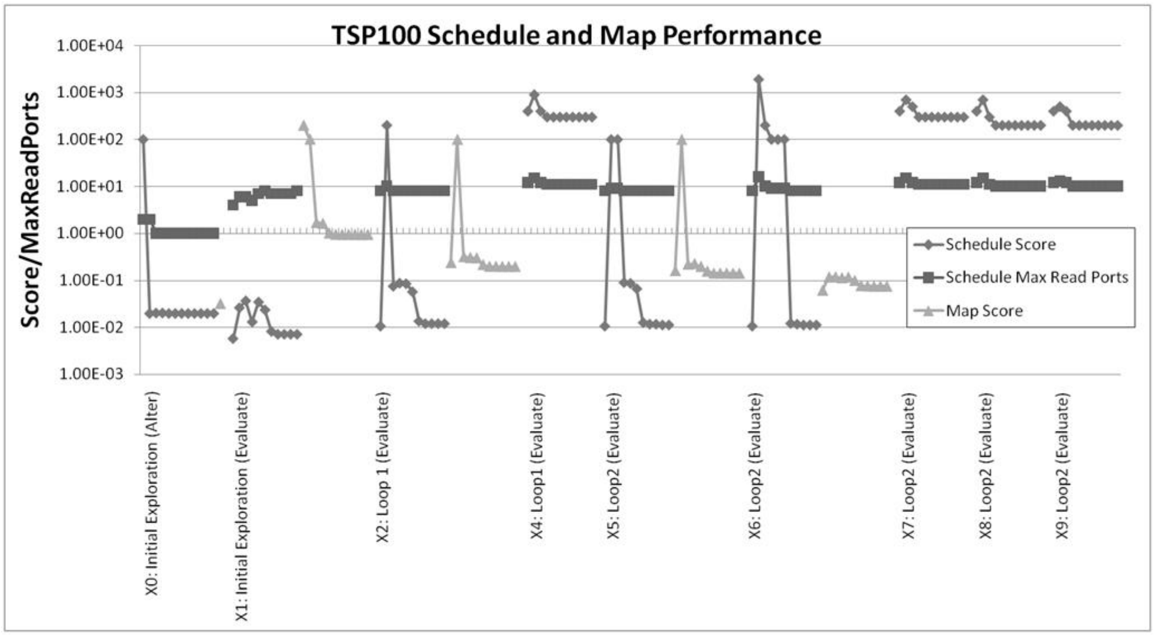


Fig. 5.2: Performance of the *Scheduler* and *Mapper* for the tsp100 problem.

did not return a solution; as such, the *Mapper* was not called.

To show that the quality of a solution by pipelining the behavior of a SAK is comparable to the non-pipelined (SAKs) version, a 100 runs for each test case was run and then averaged the score of the final solution for both versions (shown in Table 5.4). As the lab did not have sufficient resources for acquiring the targeted radiation hardened FPGA (XQR4VLX200) for testing, the functionality was emulated for the pipelined version in software to obtain the results. For the non-pipelined version, the code for the SAKs was ran on the desktop machine to get the results. The average error in the scores for these test cases was observed to be 4.23%, with a maximum of 15% for one test case. However, even for this exceptional test case, the solution provided by a pSAK is valid (but not as good). Therefore, pSAKs can be a good substitute for SAKs, given the small difference in final solutions for most test cases and the benefits of acceleration on FPGAs.

The salient improvements in the proposed algorithms over Jonathan Phillips' algorithms: 1) fixing *Loop*1 in the PLB algorithm to work correctly, 2) adding *Loop*2 in the PLB algorithm to reduce the latency of the worst case sub-system, 3) adding the option to merge Copy and Alter sub-system, 4) reducing the complexity when evaluating a solution in the *Scheduler* from $O(n)$ to $O(L)$ where n is the number of nodes and L is the desired latency, 5) using a different alteration technique in the *Scheduler* that eliminates the need to check for DGVs when evaluating a solution, 6) reducing the complexity when evaluating a solution in the *Mapper* from $O(n\ logn)$ to $O(1)$ where n is the number of nodes in the DFG, and 7) using a different alteration technique in the *Mapper* that eliminates the need to check for component over-utilization.

Table 5.4: Final scores and %Error averaged over 100 runs of the pipelined (pSAKs) vs. non-pipelined (SAKs) versions for the nine test cases.

Problem	Non-pipelined average score	Pipelined average score	%Error
dgv100	41.0	41.03	0.07
dgv300	72.8	74.74	2.66
dgv500	205.81	212.1	3.08
gc100	3.08	3.14	1.95
gc300	0.16	0.18	12.5
gc500	2.57	2.96	15.18
tsp100	2817.13	2844.81	0.95
tsp300	7500.32	7564.8	0.85
tsp500	9915.27	9995.62	0.8

Chapter 6

Conclusions

In this thesis a methodology to design FPGA circuits specifically to accelerate SAKs for space-borne applications is presented. This methodology uses a PLB algorithm that leverages the structure of a hardware architecture template and invokes a *Scheduler* and $Mapper$. A low complexity ($O(1)$) score calculation for the heuristic mapping algorithm ($Mapper$) was presented to accurately estimate area usage for support units (i.e., multiplexers and registers). A weighted sum of device primitives (WSDP) was used to calculate the relative weight of components implemented on heterogeneous FPGAs and aid in the process of area estimation during architecture exploration. The *Scheduler*, $Mapper$, and resource area estimation can be used for other applications besides the SAK.

Energy required and time to complete various FPGA based pipelined SAKs with non-pipelined SAKs implemented on a PPC 750 emulator (architecturally equivalent to the state of the art BAE RAD 750 processor used in Spacecraft systems) was compared. Average energy savings of 99% was observed with significant speedups of over 50x. This shows that FPGAs are superb candidates for space-borne autonomous mission planning and scheduling.

References

[1] D. E. Bernard, E. B. Gamble, N. F. Rouquette, B. Smith, and Y. Tung, "Remote agent experiment ds1 technology validation report," `http://nmp-techval-reports.jpl.nasa.gov/DS1/Remote_Integrated_Report.pdf`, 2000.

[2] Xilinx, "Xilinx," [http://www.xilinx.com/], 2009.

[3] Altera, "Altera," [http://www.altera.com/], 2009.

[4] Xilinx, "Basic fpga architecture," [http://www.xilinx.com/], 2009.

[5] Xilinx, "Partial reconfiguration," [http://xilinx.com/support/prealounge/protected/index.htm], 2009.

[6] B. Brumitt and A. Stentz, "Grammps: a generalized mission planner for multiple mobile robots in unstructured environments," in *International Conference on Robotics and Automation*, pp. 1564–1571, 1998.

[7] G. Sánchez-Ante, F. Ramos, and J. F. Solís, "Cooperative simulated annealing for path planning in multi-robot systems," in *MICAI '00: Proceedings of the Mexican International Conference on Artificial Intelligence*, pp. 148–157. London, UK: Springer-Verlag, 2000.

[8] T. Fayard, "Will schedulers be available on board in the next generation of robots?" `stinet.dtic.mil/dticrev/PDFs/ADA445130.pdf`, July 2005.

[9] S. Lee, R. Russell, W. Fink, R. Terrile, A. Petropoulos, and P. von Allmen, "Low-thrust mission trade studies with parallel, evolutionary computing," *Aerospace Conference, 2006 IEEE*, p. 12, 2006.

[10] W. T. Scherer and F. Rotman, "Combinatorial optimization techniques for spacecraft scheduling automation," *Annals of Operations Research*, vol. 50, no. 1, pp. 525–556, Dec. 1994.

[11] Xilinx, "Xcellence in aerospace and defense - taking designs to new heights with space-grade virtex-4qv fpgas," [http://www.xilinx.com/publications/xcellonline/xcell_65/xc_pdf/p22-26_65_XIAD.pdf], 2008.

[12] R. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, pp. 200–209, 1962.

[13] B. Pratt, M. Wirthlin, P. Graham, K. Morgan, and S. Shelley, "Improving fpga reliability in harsh environments using triple modular redundancy with more frequent voting," Military and Aerospace FPGA and Applications (MAFA) Meeting [http://nepp.nasa.gov/mafa/talks/MAFA07_18_Pratt.pdf], 2007.

[14] K. Morgan, D. McMurtrey, B. Pratt, and M. Wirthlin, "A comparison of tmr with alternative fault-tolerant design techniques for fpgas," *Nuclear Science, IEEE Transactions*, vol. 54, no. 6, pp. 2065–2072, Dec. 2007.

[15] R. Hentschke, F. Marques, F. Lima, L. Carro, A. Susin, and R. Reis, "Analyzing area and performance penalty of protecting different digital modules with hamming code and triple modular redundancy," in *Proceedings of the 15th Symposium on Integrated Circuits and Systems Design*, pp. 95–100, 2002.

[16] L. Jones, "Seu detection and correction using virtex-4 devices," [www.xilinx.com/ support/documentation/application_notes/xapp714.pdf], 2007.

[17] B. Mei, A. Lambrechts, J.-Y. Mignolet, D. Verkest, and R. Lauwereins, "Architecture exploration for a reconfigurable architecture template," *Design & Test of Computers, IEEE*, vol. 22, no. 2, pp. 90–101, March-April 2005.

[18] L. Jozwiak and S.-A. Ong, "Quality-driven template-based architecture synthesis for real-time embedded socs," in *Proceedings of the 9th EUROMICRO Conference on Digital System Design*, pp. 397–406. Washington, DC: IEEE Computer Society, 2006.

[19] P. Mishra, A. Kejariwal, and N. Dutt, "Rapid exploration of pipelined processors through automatic generation of synthesizable rtl models," in *Proceedings of the 14th IEEE International Workshop on Rapid Systems Prototyping*, pp. 226–232, June 2003.

[20] H. Ziegler, M. Hall, and B. So, "Search space properties for coarse-grained pipelined fpga applications," *16th Workshop Languages and Compilers for Parallel Computing*, pp. 149–155, 2003.

[21] B. A. Kienhuis, *Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools*. Ph.D. dissertation, Delft University of Technology, The Netherlands, 1999.

[22] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Communication of Association for Computing Machinery*, vol. 17, no. 12, pp. 685–690, 1974.

[23] P. Paulin and J. Knight, "Force-directed scheduling for the behavioral synthesis of asics," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions*, vol. 8, no. 6, pp. 661–679, June 1989.

[24] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors," *Parallel and Distributed Systems, IEEE Transactions*, vol. 7, no. 5, pp. 506–521, May 1996.

[25] S. Govindarajan and R. Vemuri, "Improving the schedule quality of static-list time-constrained scheduling," in *Proceedings at Design, Automation and Test in Europe Conference and Exhibition*, p. 749, 2000.

[26] T. Hagras and J. Janecek, "A high performance, low complexity algorithm for compile time job scheduling in homogeneous computing environments," in *Proceedings of the International Conference on Parallel Processing Workshops*, pp. 149–155, 2003.

[27] J. Nestor and G. Krishnamoorthy, "Salsa: a new approach to scheduling with timing constraints," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 8, pp. 1107–1122, Aug 1993.

[28] K. Purna and D. Bhatia, "Temporal partitioning and scheduling data flow graphs for reconfigurable computers," *IEEE Transactions on Computers*, vol. 48, no. 6, pp. 579–590, Jun 1999.

[29] M. Kaul and R. Vemuri, "Design-space exploration for block-processing based temporal partitioning of run-time reconfigurable systems," *Journal on Very Large Scale Integration Signal Processing Systems*, vol. 24, no. 2-3, pp. 181–209, 2000.

[30] B. So, M. W. Hall, and P. C. Diniz, "A compiler approach to fast hardware design space exploration in fpga-based systems," in *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pp. 165–176. New York, NY: Association for Computing Machinery, 2002.

[31] S. Chaudhuri, S. A. Blythe, and R. A. Walker, "A solution methodology for exact design space exploration in a three-dimensional design space," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 5, no. 1, pp. 69–81, 1997.

[32] Z. Gu, J. Wang, R. Dick, and H. Zhou, "Unified incremental physical-level and high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 9, pp. 1576–1588, Sept. 2007.

[33] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist, and M. Sivaraman, "Pico-npa: High-level synthesis of nonprogrammable hardware accelerators," *Journal on Very Large Scale Integration Signal Processing Systems*, vol. 31, no. 2, pp. 127–142, 2002.

[34] Y.-H. Wu, C.-J. Yu, and S.-D. Wang, "Heuristic algorithm for the resource constrained scheduling problem during high-level synthesis," *Computers & Digital Techniques, Institution of Engineering and Technology*, vol. 3, no. 1, pp. 43–51, Jan. 2009.

[35] G. Theodoridis, N. Vassiliadis, and S. Nikolaidis, "An integer linear programming model for mapping applications on hybrid systems," *Computers & Digital Techniques, Institution of Engineering and Technology*, vol. 3, no. 1, pp. 33–42, Jan. 2009.

[36] S. P. Mohanty, E. Kougianos, and D. Pradhan, "Simultaneous scheduling and binding for low gate leakage nano-complementary metal-oxide-semiconductor data path circuit behavioural synthesis," *Institution of Engineering and Technology Computers & Digital Techniques (CDT)*, vol. 2, no. 2, pp. 118–131, Mar. 2008 [Online]. Available: `http://www.cs.bris.ac.uk/Publications/Papers/2000840.pdf`.

[37] T. Xu, K. Chakrabarty, and F. Su, "Defect-aware high-level synthesis and module placement for microfluidic biochips," *Biomedical Circuits and Systems, IEEE Transactions*, vol. 2, no. 1, pp. 50–62, Mar. 2008.

[38] W. Sun, M. J. Wirthlin, and S. Neuendorffer, "Fpga pipeline synthesis design exploration using module selection and resource sharing," *Computer-Aided Design of*

Integrated Circuits and Systems, IEEE Transactions, vol. 26, no. 2, pp. 254–265, Feb. 2007.

[39] G. Wang, W. Gong, B. DeRenzi, and R. Kastner, "Ant colony optimizations for resource- and timing-constrained operation scheduling," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions*, vol. 26, no. 6, pp. 1010–1029, June 2007.

[40] B. Prince, "Application specific drams today," in *International Workshop on Memory Technology, Design and Testing*, pp. 7–13, July 2003.

[41] S. Memik, N. Bellas, and S. Mondal, "Presynthesis area estimation of reconfigurable streaming accelerators," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions*, vol. 27, no. 11, pp. 2027–2038, Nov. 2008.

[42] D. Zaretsky, G. Mittal, R. Dick, and P. Banerjee, "Balanced scheduling and operation chaining in high-level synthesis for fpga designs," *International Symposium on Quality Electronic Design*, pp. 595–601, Mar. 2007.

[43] S. Bilavarn, G. Gogniat, J.-L. Philippe, and L. Bossuet, "Design space pruning through early estimations of area/delay tradeoffs for fpga implementations," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions*, vol. 25, no. 10, pp. 1950–1968, Oct. 2006.

[44] C. Carmichael and C. Tseng, "Correcting single-event upsets in virtex-4 platform fpga configuration." `www.xilinx.com/support/documentation/applicationnotes/xapp988.pdf`, 2008.

[45] Virtutech, "Virtutech simics," `https://www.simics.net/`, 2009.

[46] T. Becker, W. Luk, and P. Cheung, "Enhancing relocatability of partial bitstreams for run-time reconfiguration," *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 35–44, Apr. 2007.

[47] A. Burcin, "Rad750 mrqw," `http://www.aero.org/conferences/mrqw/2002-papers/A_Burcin.pdf`, 2002.