Title
Energy Efficient Computing Using Static-Dynamic Co-optimizations

Author
Gururaj, Karthik

Publication Date
2013

Peer reviewed|Thesis/dissertation

University of California
Los Angeles

Energy Efficient Computing Using Static-Dynamic Co-optimizations

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Karthik Gururaj

2013

Abstract of the Dissertation

Energy Efficient Computing Using Static-Dynamic Co-optimizations

by

Karthik Gururaj

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2013

Professor Jason Cong, Chair

Energy consumption is a primary concern of current day computing systems – from handheld battery operated systems to servers in data centers connected to wall power. Research in academia as well as industry has focused on a variety of techniques for minimizing energy consumption while maintaining a good level of performance. The most effective techniques combine the best of static (or offline) and dynamic (or online/runtime) optimizations to obtain the best solution. Static optimizations can be more complex and can afford to take a global view of the application/computation being executed on the system – however, such optimizations have to be conservative in nature because they cannot anticipate all the different scenarios that can appear at runtime. Dynamic optimizations have more information about the application/computation for the given input – however, since such optimizations have to have low overhead, they can afford to have only a local view of the computation/ application and the complexity of the optimization has to be relatively low. An additional direction that needs to be considered is to determine whether dynamic optimizations should be implemented in software or hardware. In this thesis, I present a study of three aspects of energy efficient computing that use a combination of static and dynamic optimizations to minimize energy consumption. The first aspect is to consider variability in the

execution time of applications during scheduling for dynamic voltage frequency scaling (DVFS) capable systems to minimize energy consumption and yet maintain a desired level of performance. The main idea is to construct a schedule table offline and perform a simple table look-up at runtime. The second aspect is to consider application-level reliability for applications that can tolerate certain error in the outputs. I present the study of a profile-guided offline compilation strategy to identify critical instructions and a monitoring technique in software to handle corner cases at runtime. Finally, the third aspect of energy efficient computing I investigate is flexibility – customizing the instruction sets of processors to improve energy efficiency. I study the benefits of compiler directed optimizations for generating custom instructions which are executed within a modified processor pipeline and an architecture mechanism for detecting corner cases and to roll-back to a safe state. Additionally, I investigate the benefits of customizing the instruction set dynamically in hardware.

The dissertation of Karthik Gururaj is approved.

Yuval Tamir

Glenn Reinman

Lei He

Jason Cong, Committee Chair

University of California, Los Angeles

2013

To my family who have supported me at every step.

Table of Contents

List of Tables

List of Figures

Acknowledgments

Firstly, I would like to thank my adviser, Professor Jason Cong, for his insightful guidance and help during the full course of my study. If I was asked to state the most important thing that I learned from him as a student, it would be the method of approaching a research problem. Professor Cong has always emphasized the need to model research problems under a formal framework and to develop algorithms that provide some guarantee of global optimum. Under his approach, the simplifications to the framework and algorithm (to make it tractable) can come later. This methodology is quite different from some of the ad-hoc techniques that we see in many publications. Professor Cong has provided numerous research ideas and refinements during the course of the study. His suggestions, both technical and with respect to presentation, have been very helpful in polishing my rough ideas into a publishable work.

I would also like to thank Professor Glenn Reinman for the many discussions we have had during the architecture meetings, especially when he plays the role of the devil's advocate when trying to nail down the novelty of a proposed architecture/design.

I would like to thank several members of VLSI CADLAB for their help and the many discussions we have had. Guoling Han was practically a co-adviser in my early years as a graduate student when we worked on the synthesis of processor networks. Yi Zou has helped me several times while working with the high-level synthesis tool AutoPilot. Discussions with Sen Li have helped me improve/fix my code several times. I have had several meeting when dealing with architecture simulators with Mishali Naik, Mohammadali Ghodrat, Chunyue Liu and Adam Kaplan. Other students with whom I have worked with include Bo Yuan, Kirill Minkovich and Mike Gill.

I am thankful to my family – my parents and my brother for supporting me

through my program.

The research work in this thesis has been supported by the Gigascale Systems Research Center (GSRC) 2009-DT-2049 and the National Science Foundation under the Customizable Domain-Specific Computing (CDSC) center CCF-0926127.

Vita

2002–2006	B.Tech. Department of Computer Science and Engineering, Indian Institute of Technology Madras (IITM)
2006–present	Ph.D. student, Department of Computer Science, University of California Los Angeles

Publications

[1] J. Cong, K. Gururaj, G. Han, A. Kaplan, M. Naik and G. Reinman,"MC-Sim: An efficient simulation tool for MPSoC designs", Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2008), San Jose, CA, pp. 364-371, November 2008.

[2] J. Cong, K. Gururaj, and G. Han,"Synthesis of Reconfigurable High-Performance Multicore Systems", Proceedings of Field Programmable Gate Arrays, Monterey, California, pp. 201-208, February 2009.

[3] J. Cong and K. Gururaj,"Energy Efficient Multiprocessor Task Scheduling under Input-dependent Variation", Proceedings of Design, Automation and Test in Europe, pp. 411-416, April 2009.

[4] J. Cong, K. Gururaj, B. Liu, C. Liu, Z. Zhang, S. Zhou and Y. Zou,"Evaluation of Static Analysis Techniques for Fixed-Point Precision Optimization", Proceedings 17th Annual IEEE Symposium on Field-Programmable Custom Computing

Machines (FCCM 2009), Napa, California, pp. 231-234, April 2009.

[5] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, W. Hwu,"High-Performance CUDA Kernel Execution on FPGAs", ACM/SIGARCH 23rd International Conference on Supercomputing, Metro New York City Area, pp. 515-516, June 2009.

[6] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W. W. Hwu,"FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs", Symposium on Application Specific Processors , pp.35-42, July 2009. (Best Paper Award)

[7] J. Cong, K. Gururaj, W. Jiang, B. Liu, K. Minkovich, B. Yuan and Y. Zou,"Accelerating Monte Carlo based SSTA using FPGA", Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA 2010), Monterey, California, pp. 111-114, February 2010.

[8] A. Papakonstantinou, Y. Liang, J.A. Stratton, K. Gururaj, D. Chen, W.M. Hwu and J. Cong,"Multilevel Granularity Parallelism Synthesis on FPGAs", Proceedings of 19th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2011), Salt Lake City, UT, pp. 178-185, May 2011. (Best Paper Award)

[9] J. Cong, K. Gururaj, H. Huang, C. Liu, G. Reinman and Y. Zou ,"An Energy-Efficient Adaptive Hybrid Cache", Proceedings of International Symposium on Low Power Electronics and Design (ISLPED 2011), Fukuoka, Japan, pp. 67-72, August 2011.

[10] J. Cong, K. Gururaj, M. Huang, S. Li, B. Xiao and Y. Zou,"Domain-Specific Processor with 3D Integration for Medical Image Processing", Proceedings of the 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2011), Santa Monica, CA, pp. 247-250, September 2011.

[11] J. Cong and K. Gururaj,"Assuring Application-Level Correctness against Soft Errors", Proceedings of the 2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2011), San Jose, CA, pp. 150-157, November 2011.

[12] J. Cong and K. Gururaj,"Architecture Support for Memory Operations in Custom Instructions", Proceedings of the 21st ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2013), Monterey, California, pp. 231-234, February 2013.

CHAPTER 1

Introduction

Energy consumption is a primary concern of current day computing systems – from handheld battery operated systems to servers in data centers connected to wall power. The most effective techniques combine the best of static (or offline) and dynamic (or online/runtime) optimizations to obtain the best solution. Static optimizations by nature can be more complex and can afford to take a global view of the application/computation being executed on the system – however, such optimizations have to be conservative in nature because they cannot anticipate all the different scenarios that can appear at runtime. Dynamic optimizations have more information about the application/computation for the given input – however, since such optimizations have to have low overhead, they can afford to have only a local view of the computation/application and the complexity of the optimization has to be relatively low. A relatively well-known example is the optimization for exploiting memory locality – compilers may be able to reorder loops for better locality; however, they cannot anticipate exactly which data should be kept in on-chip caches. This where a dynamic and relatively simple block replacement policy such as LRU (least recently used) comes in.

In this thesis, I will investigate three aspects of energy efficient computing that exploit static and dynamic optimizations.

1.1 Energy efficient multiprocessor task scheduling under input-dependent variation

Modern processors are capable of switching their voltage/frequency at runtime to reduce dynamic energy consumption (DVFS). The goal is to provide an acceptable level of performance for applications while minimizing energy consumption. I describe a scheduling heuristic for exploiting dynamic variation in execution time of sub-tasks of an application to further reduce energy while still meeting the required performance which consists of an offline convex optimization algorithm and a simple greedy online algorithm.

1.2 Assuring application-level correctness in programs

As transistor sizes get smaller, the probability of a failure leading to erroneous computation increases. In this section, we target applications with "elastic" outputs – meaning that outputs with a certain amount of error can be tolerated. The goal is to provide an acceptable output solution with minimum overhead – both in terms of energy and performance. We present a profile-guided compilation strategy to identify critical instructions – instructions which can severely affect the quality of the output if erroneous – and a runtime monitoring technique to identify corner cases which the compiler could not anticipate.

1.3 Architecture support for custom instructions with memory operations

Accelerators are in general more energy-efficient when compared to conventional processors (running software). In this study, we propose an architecture for fine-grained integration of accelerators into the processor pipeline allowing the ac-

celerators to share the processor's resources. A compilation framework is also developed enable programs to exploit fine-grained accelerators. We show that such an approach with the compilation framework and architecture working together can produce more energy efficient solutions than when these techniques are applied in isolation.

1.4 Architecture support for dynamic instruction set customization

Offline instruction set customization has some limitations: (1) Large space requirements for storing the configuration bits for all custom instructions. (2) Inability use custom functional units in legacy binaries and precompiled library code. An architecture framework is developed to dynamically determine 'hotspots' in the program and insert custom instructions into the instruction stream. These custom instructions are executed on a reconfigurable functional unit with a very low overhead of reconfiguration.

CHAPTER 2

Energy Efficient Multiprocessor Task Scheduling under Input-dependent Variation

2.1 Introduction

Energy consumption is an important issue in designing battery operated embedded systems. The goal of designers is to create a system which consumes minimal energy and satisfies performance constraints at the same time. Multi-core chips are becoming increasingly popular for embedded systems. Examples include the ARM Cortex A-9 MPCore [2] and the MIPS 1074K [8]. One of the most effective design techniques for minimizing energy consumption of processors is dynamic voltage frequency scaling (DVFS), in which processor frequency and voltage can be adjusted depending on the workload of the processor. A primary problem of interest has been to minimize the energy consumption of an application running on a system subject to performance constraints. A variety of algorithms have been proposed to tackle the energy efficient scheduling and mapping of applications to a multi-core system subject to performance constraints – we examine the major classes of algorithms proposed in this area.

2.1.1 Workload agnostic techniques

Techniques such as those described in [54], do not assume any knowledge about the computation time of input tasks beforehand. Such techniques use simple, online algorithms to estimate computation workload in the next time-step based

on the workload in recent as well as the global history. Based on this estimation and the list of tasks that are ready to be processed, scheduling and task mapping is done. Power-reduction techniques are applied if the estimated workload can be completed subject to timing constraints. These techniques are pre-dominantly used in OS schedulers.

2.1.2 Workload aware techniques

Techniques that fall into this category typically assume knowledge of the application – the most common assumption is that the worst-case execution time and power consumption of each application or sub-task are known beforehand. The performance constraint specified is the maximum execution time of the application.

2.1.2.1 Theoretical results

The problem of scheduling a given set of tasks onto multiple processors to minimize an arbitrary cost function subject to latency constraint and resource constraints is NP-Hard [102]. In our case, the cost function is power. A 2-approximation [102] is given for the case in which all tasks are independent. For the case where precedence constraints exist among tasks, a $O(nlogm)$ approximation scheme is provided in [59] ere m is the number of processors available. If we drop the resource constraint and consider the problem where all processors are identical, with each processor capable of functioning at k discrete voltages $\{V_1, V_2, ..., V_k\}$, the problem of minimizing power consumption subject to latency constraint is still NP-Hard. A simple reduction from 0-1 Knapsack problem proves this. If we further drop the constraint that only a few discrete voltage levels are available and assume that every processor can switch dynamically to any voltage, then the problem of minimizing power can be solved in polynomial time [56]. In [56], the

problem of assigning the optimal voltages to each task is reduced to optimally assigning slack to each task. The solution is based on the assumption that in the optimal solution the lengths of all paths from a node to a primary output are the same. Based on this assumption, a LP is formulated, the dual of which is shown to be identical to a network flow problem. A heuristic is proposed in [43] which uses the LP solution to guide an iterative technique which assigns discrete voltages to each task. If the processor can be only be run at discrete voltages/frequencies, but is allowed to switch voltage/frequency in the midst of executing a task, then the scheduling problem can be solved optimally. The authors in [66] prove that at most one voltage switch in the middle of a task is sufficient to guarantee the optimal solution. The basic idea is to generate a solution similar to [56], determine the two discrete processor voltages between which the voltage for each task falls (as per the solution) and split the execution time of the task between the two processor voltage levels.

2.1.2.2 Heuristic techniques

List scheduling [58] is a popular technique for resource constrained scheduling. In [60], the authors extend list scheduling to take into account energy consumption. The priority function to guide the list scheduling algorithm is a weighted average of the energy saving of the current task and the amount of time available for the successor tasks lying in the critical path of the application.The technique is iterative in nature where initial weight assigned to energy savings is higher and gradually decreased to ensure that the performance constraint is satisfied. Force directed scheduling [89] is another general scheduling heuristic – the authors in [91] modify the force function to reflect the energy consumed by the system.

2.1.3 Exploiting variation in execution time

The techniques described in the previous section assume that the execution time of each sub-task of the application is always the worst-case execution time – they do not exploit variation in the execution time of the application (because of variation in the inputs). In this section, we describe techniques which exploit this variation to further reduce energy consumption.

2.1.4 Uni-processor Systems

The work in [61][72][118][117][14] models the execution time of a task as a random variable and minimizes expected energy consumption on a single processor system. A heuristic is provided in [61] for obtaining a low-energy schedule. In [72][118], exact solutions are provided using convex optimization techniques; however, many of their assumptions, such as the ability to change the voltage to any arbitrary value at any point during the execution of a task, are not valid for practical systems. Many of these issues are addressed in [117] for uni-processor systems. In [14], a mathematical formulation is presented to optimize the expected energy consumption (both dynamic and leakage) by using DVFS and Adaptive Body Biasing (ABB). However, a simple extension of the proposed method for multi-processor systems leads to an exponential increase in complexity.

2.1.5 Multiprocessor Systems

Dynamic slack reclamation based techniques: In [122][24], the authors propose techniques by which the dynamic slack is distributed among the remaining tasks. While the work in [24] does not consider precedence constraints among tasks, a list scheduling heuristic is used in [122] for tasks with precedence constraints. Such online techniques are constrained to be relatively simple and fast.

2.1.5.1 Schedule Table based

The idea of using heuristic to build a schedule table at design time was proposed in [115][101] for scheduling and voltage assignment for conditional task graphs (CTG). However, the proposed techniques are restricted to CTGs.

2.1.5.2 Expected energy minimization

A highly complex, non-linear integer programming based method is proposed in [78] for task mapping and scheduling in multiprocessor systems. In [116], the authors attempt to balance the expected energy consumption across processors by partitioning a set of independent tasks. In [93], a dynamic programming based method is used to minimize expected energy consumption. However, the proposed method is exponential in complexity for multiprocessor systems. The primary contributions of this paper can be stated as follows.

- We propose a mathematical programming formulation based technique for scheduling tasks on DVFS capable multiprocessor systems, which takes into account input-dependent variation in execution time of tasks to reduce average energy consumption subject to a specified latency constraint. Our technique is capable of handling precedence constraints among tasks.

- Our technique runs in polynomial time for multiprocessor systems; the solution is optimal for tree like task graphs. This is achieved by a novel pruning method during the formulation phase that avoids the exponential enumeration done in [14] [93].

- Our algorithm constructs a schedule table at design time to provide multiple scheduling options for each task. While complex algorithms can be used to build the schedule table at design time, the only extra processing that a system needs to perform at run-time is a table look-up.

The rest of the chapter is organized as follows: section 2.2 describes our task graph and processor models. Section 2.2.3 provides the problem statement. In section 2.3, our formulation is presented by which variation in execution-time can be exploited; Experiments performed on randomly generated task-graphs as well as real-world applications are described in section 2.5.

2.2 Preliminaries and problem statement

2.2.1 Processor Model

We assume that the number of processors is restricted to be no more than a certain number P and that the voltage of every processor in the system can be set independently to any value within a given range $[V_{lower}, V_{upper}]$ at run-time. The overhead for switching between voltages is assumed to be negligible compared to the execution time of individual tasks. To model the relation between energy, voltage and clock frequency, we use well known equations for CMOS logic [81]:

$$f = C_1 \frac{(V - V_{th})^\alpha}{V} \tag{2.1}$$

$$E = C_2 W V^2 \tag{2.2}$$

where f is the clock frequency, V is the supply voltage, W is the number of cycles taken by a task (or the workload). C_1 and $C_2$are constants. α is a constant between 1 and 2. In this paper, we approximate the frequency to be a linear function of supply voltage [62][30]. Thus, we can see that energy can be modeled as a non-increasing convex function of the clock period as shown in Equation 2.2. From this point onwards in the paper, we model the energy consumption as a function of clock period cp, instead of the clock frequency. Our method is applicable to any convex, non-increasing function of clock period.

$$E = C_3 W f^2 = \frac{C_3 W}{(cp)^2} \tag{2.3}$$

2.2.2 Application Model

We assume that an application is represented as a directed acyclic graph (DAG) $G(V, E)$ called the task graph. Each node $v \in V$ represents a task that has to be executed on a single processor without preemption. Moreover, every task is restricted to run at a single voltage. A directed edge $(u \rightarrow v) \in E$ represents a precedence constraint between the tasks represented by nodes u and v. Each precedence constraint is also associated with a certain communication delay between the two tasks. In our model, we assume that the execution time of the source task (of an edge) includes the time for communication of data from the predecessor to the successor. A latency constraint on G is a timing constraint which specifies the allotted time L within which the execution of G should be completed. We assume that G will be re-executed every L time units. We now define a set of parameters associated with each task v:

- $WorkLoad(v, I)$ of a task v is defined as the number of clock cycles taken by v to complete execution. Note that the workload is not the execution time because the execution time for a fixed workload varies with the clock frequency. Also, the $WorkLoad$ depends on input I.

- $WCW(v)$ (Worst-Case Workload) of a task v is defined as the maximum workload of the task v.

  Motivating example: Consider the task graph G shown in Figure 1. All four tasks are assumed to be identical and the $W - Vector$ for the tasks is shown in Figure 1. The worst-case workload, $WCW(v)$, is assumed to be 100 cycles for all the tasks. The probability that the workload of a task is no greater than 75 cycles is 0.7 and that the workload is between 75 and 100 cycles is 0.3. Suppose we are given a latency constraint of 450ns for this example on a 2 processor system $P1$ and $P2$. Using the model described in section 2.2, we can determine that when all the tasks run for 75 cycles, the

Table 2.1: Sample schedule table

Task	Entry 1	Entry 2	Entry 3
v_1	$< 0, 1.5, 0 >$	-	-
v_2	$< 75, 1.68, 112.5 >$	$< 100, 1.5, 150 >$	-
v_3	$< 75, 1.68, 112.5 >$	$< 100, 1.5, 150 >$	-
v_4	$< 150, 2.1, 238.5 >$	$< 175, 1.69, 280.5 >$	$< 200, 1.5, 300 >$

worst-case scheduling produces a solution which consumes 44% more energy than the optimal solution.

- *Start Cycles Elapsed (SCE(v))* for a task v is the number of clock cycles elapsed when all predecessors of v have completed. For tasks with only primary inputs, $CE(v)$ is 0. For other tasks, $SCE(v) = \max_u SCE(u) + Workload(u)$ where $u \in predecessors(v)$.

- *End Cycles Elapsed (ECE(v))* for a task v is the number of clock cycles elapsed when v finishes. $ECE(v) = SCE(v) + Workload(v)$.Also, note $SCE(v) = \max_u ECE(u)$ where $u \in predecessors(v)$

  In the above example, if v_1, v_2 and v_3 take 75 cycles each, $SCE(v_4)$ is 150 cycles and $ECE(v_4)$ is 225 cycles.

- A *schedule table* is a table that has for each task v, a vector of tuples $< scev, i, cpv, i, sv, i >$ where $(sce_{v,1}, sce_{v,2}, ...sce_{v,K})$ is the list of possible values of $SCE(v)$ sorted in increasing order, $cp_{v,i}$ is the clock period at which the task v is run and $s_{v,i}$ is the start time of v when the value of $SCE(v)$ is $ce_{v,i}$. Continuing to use our example in Figure 1, we show a sample schedule table in Table 2.1 when the latency constraint is set to 450ns. Suppose at run-time, $CE(v_4)$ is computed to be 175 cycles, then the schedule table matches this value to the entry ¡175, 1.69, 280.5¿ (Row 3, Column 2 in Table 2.1). Thus, the task v_4 is scheduled to start at time 280.5ns and run with a

clock period of 1.69 ns. This implies that task v_4 will complete at most by time 450ns because the worst case workload of v_4 is 100 cycles.

2.2.3 Problem Statement

At run-time, each task u propagates the value of $ECE(u)$ to its successors. Every task v computes the value of $SCE(v)$ and performs a table look-up to determine the start time of v and the clock period to use. This immediately implies that considering different values of $SCE(v)$ provides us with a way of exploiting workload variation to generate different scheduling solutions and clock period assignments for task v (seen in Table 2.1). With this in mind, we formally state the problem: Given a task-graph $G(V, E)$, the $WorkLoad$ distribution associated with each task v, and a latency constraint L, the goal of the scheduling algorithm is to construct a schedule table T such that the average energy consumption is minimized and the latency and precedence constraints are satisfied for any combination of workloads of the tasks of G.

2.3 VAR-TB – Variation-aware Time Budgeting

Our algorithm is divided into two phases – the first phase assigns tasks to processors and the second phase determines the start time and voltage assignment for each task.

2.3.1 Task assignment heuristic

The problem of resource-constrained energy minimization subject to latency constraints has been proved to be NP-Hard [24]. We use a priority based heuristic to assign the tasks to a set of P processors. The highest priority task is scheduled to run on the first processor that is ready to accept a new task. In our experiments,

we use the difference between the ALAP and ASAP time to decide the task priorities. These times are computed by assuming that all processors run at their highest frequency and all tasks run at their worst-case. After a task is assigned, the ASAP and ALAP times for the remaining tasks are re-computed. We insert pseudo edges between consecutive tasks running on the same processor to enforce the resource constraints during the scheduling step.

2.3.2 Task scheduling and voltage assignment

We present the mathematical formulation for the scheduling and voltage assignment problem. We first present the variable organization in our formulation. We contrast our approach with two existing works and explain how our formulation avoids enumeration of a large number of task workloads. In section 2.4.2, we prove why our approach can run in polynomial time and yet provide optimal solutions for certain kinds of task graphs.

2.3.3 Mathematical formulation of VAR-TB

In [67], the authors provide a mathematical formulation for the integer time-budgeting problem. However, they do not consider variation in workloads of the tasks. Based on the formulation in [67], we propose a novel method to handle variations in workloads. For each task v, we introduce a number of start time variables and clock period variables. With every value of $SCE(v)$ (given by $sce_{v,i}$), we associate a start time variable $s_{v,i}$ and clock period variable $cp_{v,i}$. Every predecessor u of v also has a set of finish time variables. Each finish time variable - $f_{u,j}$ is associated with a value of $ECE(u)$ (given by $ece_{u,j}$). Based on these variables, our formulation consists of the following constraints:

$$s_{v,i} \geq 0 \tag{2.4}$$

$$s_{v,i} + Workload_k(v) cp_{v,i} \leq f_{v,l} \iff sce_{v,i} + Workload_k(v) \leq ece_{v,l} \tag{2.5}$$

$$s_{v,i} \geq f_{u,j} \iff ece_{u,j} \leq sce_{v,i} \tag{2.6}$$

$$cp_{lower} \leq cp_{v,i} \leq cp_{upper} \tag{2.7}$$

$$fZ_{v,l} \leq L \forall v, l \tag{2.8}$$

The first constraint imposes non-negativity on the start time variables. Constraint 2.8 imposes latency constraint on the finish times of each task. The second constraint ensures that the finish time is greater than the sum of start time and execution time (expressed as a product of workload and clock period). Constraint 2.5 is repeated for every value of $WorkLoad(v)$. The third constraint imposes precedence between u and v. Constraint 2.7 imposes bounds on the clock period variable. Note that the only variables in the above problem are the start times, finish times and clock periods. We explain precedence constraints (Constraint 2.6) with the example from Figure 1. Task v_1 can complete after 75 cycles ($ece_{v1,1}$) or 100 cycles ($ece_{v1,2}$). Thus, task v_1 has 2 finish time variables – $f_{1,1}$ and $f_{1,2}$ associated with 75 and 100 cycles respectively. Task v_2 can start after 75 cycles ($sce_{v2,1}$) or 100 cycles ($sce_{v2,2}$). Hence, v_2 has 2 start time variables – $s_{2,1}$ and $s_{2,2}$. The precedence constraints are given by:

$$s_{2,1} \geq f_{1,1} \tag{2.9}$$

$$s_{2,2} \geq f_{1,2} \tag{2.10}$$

Note that *no constraints* exist between $s_{2,1}$ and $f_{1,2}$ because task v_2 starts at time $s_{2,1}$ only when task v_1 has completed within 75 cycles which ensures that

task v_1 would have completed by time $f_{1,1}$. When v_1 runs at the worst case (100 cycles), it will complete by $f_{1,2}$ and v_2 will start at time $s_{2,2}$. This ensures that the precedence constraints are always satisfied. For task v_2, $s_{2,2}$ is the worst-case start time. If v_1 completes by 75 cycles, v_2 can start at time $s_{2,1}$ (which could be less than $s_{2,2}$), thus allowing our method to exploit variation in $WorkLoad(v1)$. Such precedence relationships are exactly captured by constraint 2.6 which states that a particular start time ($s_{v,i}$) should be greater than a finish time of predecessor u ($f_{u,j}$) if and only if the corresponding values of $sce_{v,i}$ and $ece_{u,j}$ satisfy the relation – $ece_{u,j} \leq sce_{v,i}$.

We return to our example from Figure 1 to explain Constraint 2. Constraint 2 enforces a relationship between the start time, execution time and finish time of a task v. As explained previously, task v_2 has two values of $SCE(v_2)$ - 75 cycles ($sce_{v2,1}$) or 100 cycles ($sce_{v2,2}$). Task v_2 can have two values of $WorkLoad$ – 75 and 100 cycles. Thus, $ECE(v_2)$ can have 3 values – 150, 175 and 200, each of which is associated with the finish time variables $f_{2,1}$, $f_{2,2}$ and $f_{2,3}$ respectively. Consider the case when $SCE(v_2)$ is 75 cycles ($sce_{v2,1}$) and $WorkLoad(v_2)$ is 100 cycles. We observe that $ECE(v_2)$ is 175, the start time of v_2 is given by $s_{2,1}$ (see previous paragraph), the finish time for v_2 is $f_{2,2}$ and the clock period variable is given by $cp_{2,1}$ (because $SCE(v_2)$ is $sce_{2,1}$). Hence, we impose the following constraint: $s_{2,1} + 75cp_{2,1} \leq f_{2,2}$.

Such a relationship is captured by constraint 2.5 which states that the sum of a start time ($s_{v,i}$) and execution time ($WorkLoad_k(v) * cp_{v,i}$) should be less than the finish time ($f_{v,l}$) if and only if the sum of associated value of $SCE(v)$ ($sce_{v,i}$) and $WorkLoad_k(v)$ is less than or equal to the associated value of $ECE(v, l)$ ($ece_{v,l}$).

We make the following observations about our method:

- Feasible schedule: The proposed constraints are safe because for a task v, for every value of $SCE(v)$, there always exists a start time that is greater than

the finish time of its predecessors and satisfies latency constraint (assuming a feasible schedule exists).

- Exploits variation in workload of tasks.
- Avoids enumeration of combination of workloads of all tasks: Note that in our method, the only variables that are considered while specifying the precedence constraints are the start times of the task v and the finish time variables associated with its predecessors. The approaches in [14][93] begin by fixing a start time for each task with multiple predecessors. The optimization pass is executed following which the start time of one of the tasks is changed. The methods terminate when all possible combinations of start times have been enumerated. As an example, consider a task graph with 10 tasks, each of which can have 4 different start times. The methods in [14][93] will require 410 optimization passes each involving approximately 10 variables and m constraints (where m is the number of edges). Our method on the other hand requires a single optimization pass with approximately 40 variables and $16m$ constraints (section 2.4.2). Note that the approaches in [14][93] work well for single processor systems because the task graph is restricted to a chain and the start time of only a single task needs to be fixed for each optimization phase. However, for multiprocessor systems, this is not true.

2.3.3.1 Objective Function

The objective function represents the average energy consumption and is given by equation 2.11. $prob_{v,i,n}$ is the probability that the value of $SCE(v)$ is $sce_{v,i}$ and $WorkLoad(v)$ is $WorkLoad_n(v)$, $cp_{v,i}$ is the clock period for task v when $SCE(v)$

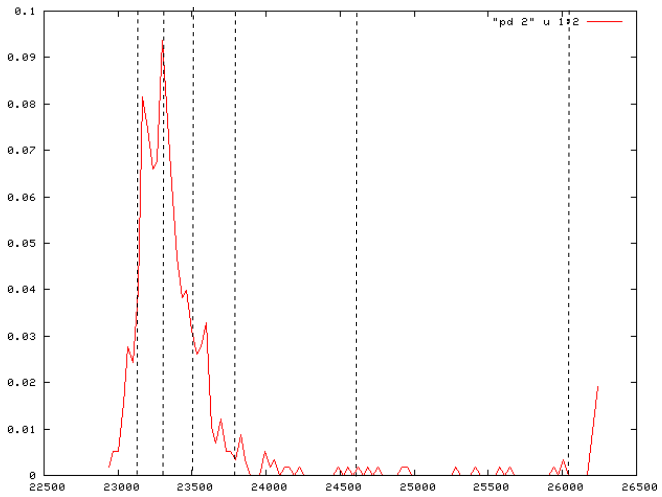

Figure 2.1: Fixing SCE values

is $ece_{v,i}$ and C_v is a constant for a given task.

$$\sum_{v \in V} \sum_{i=1}^{K} \sum_{n=1}^{M} \frac{C_v * prob_{v,i,n} * Workload_n(v)}{{cp_{v,i}}^2} \tag{2.11}$$

We observe that the only variable in the objective function is the clock period $cp_{v,i}$. Also, the objective function is convex separable and non-increasing. The probability information is obtained by profiling the application.

2.4 Improving the scheduling algorithm

2.4.1 Restricting the number of $SCE(v)$ entries per task

The number of values of distinct $SCE(v)$ can be very large even for small task graphs. To avoid this, we limit the number of values of $SCE(v)$ per task to be less than a constant K. In our approach, we select the K values so that the area under the probability distribution curve for $SCE(v)$ is split into K equal regions as shown in Figure 2.1.

2.4.2 Time complexity

Since the number of values of $SCE(v)$ per task is always less than a constant K, we ensure that the number of variables associated with each task is no more than $O(K)$ and the number of constraints associated with each precedence edge (Constraint 2.6) is no more than $O(K^2)$. Thus, the number of variables is linear with respect to the number of tasks ($O(K * n)$) and the number of constraints is linear with respect to the number of edges ($O(K^2 * m)$).

Lemma: Since all the constraints are linear and the objective function is separable convex, algorithm VAR-TB produces a schedule table to minimize average energy consumption subject to latency and precedence constraints in polynomial time [67]. The proof for the lemma follows from the fact that a minimization problem with a convex objective function and linear constraints can be optimally solved in polynomial time.

We state a theorem which proves why our method of associating start (finish) time variables with cycles elapsed is optimal for certain kinds of task graphs. The main assumption is that dynamic energy is a convex, non-increasing function of clock period.

THEOREM: Given a chain of tasks $(T_1, ..., T_i)$ to be executed sequentially and a latency constraint L, the value of the optimal energy consumption for a single run is dependent only on the total number of cycles consumed during the run and independent of the distribution of cycles consumed by the individual tasks.

The proof for this theorem follows directly from the fact that a chain of tasks can be viewed as a single task whose workload is the sum of the cycles of all the tasks in a chain. The energy consumption depends only on the number of cycles elapsed.

From the above theorem, we observe that for a given task chain, in the optimal configuration, the finish time of a task chain depends only on the cycles elapsed

and not on the workloads of the individual tasks. This is identical to the way we organize start (finish) time variables in our formulation. Such task chains are very common in pipelined applications.

2.4.3 Online algorithm

We introduce a simple online phase which does greedy slack reclamation. When a task starts, it consumes all dynamic slack available from its predecessors and completes within its assigned finish time. Thus, the complexity of the online algorithm is $O(1)$. The idea is the schedule table provides a good scheduling solution in the "global" sense while the greedy, online heuristic exploits slack in a local manner.

2.4.4 Voltage switching overhead

Real processors incur an overhead both in terms of time and energy to make a voltage (frequency) switch. We use the technique described in [14] to model this overhead. In this technique, the overhead is proportional to the magnitude of change of frequency switch. This can be expressed mathematically as shown below.

$$t_{over} = C_{latency} * |cp_2 - cp_1| \tag{2.12}$$

$$E_{over} = C_{energy} * |cp_2 - cp_1| \tag{2.13}$$

$C_{latency}$ and C_{energy} are constants and a frequency switch is made from the frequency corresponding to clock period cp_1 to the frequency corresponding to cp_2. Note that under this model, the constraints are still linear. However, for many real processors the switching overhead is a constant irrespective of the magnitude of the frequency switch [29]. This breaks the continuous nature of our formulation (both objective function and timing constraint) and cannot be currently incorporated into our framework.

2.4.5 Discrete voltages

Real processors operate only at a fixed set of discrete supply voltages (and frequencies). If each task is required to run at a single voltage only, the problem of minimizing energy consumption subject to latency constraints when the processor can run at discrete supply voltages is NP-Hard [25]. For this case, we propose a heuristic – for each clock period variable we round the value of the frequency to the next higher frequency value. This creates some slack because all the tasks now run faster. We then use a priority based approach to determine whether certain tasks can be slowed. The priority of a task is given by the energy savings that can be achieved if the frequency at which the task is to be executed is slowed to the next available frequency subject to latency constraints. Tasks are sorted in decreasing order and the highest priority task is slowed down. The process is repeated till no tasks can be further slowed down.

However, if a processor is allowed to make a voltage switch while executing a task, the authors in [66] propose a method by which the schedule to minimize energy consumption can be obtained efficiently even when the processor can only run at a discrete set of voltages. We briefly explain their approach in this section. Suppose task v is scheduled to run at frequency f_{ideal} after VAR-TB completes and

$$f_1 \leq f_{ideal} \leq f_2$$

where f_1 and f_2 are the frequencies that the processor can run at that are closest to f_{ideal}. The authors in [66] prove that it is sufficient to run task v partly at f_1 and partly at f_2. Using this result, we formulate a second problem after VAR-TB has completed. Let the number of cycles that task v runs at frequency f_1 by given by x_1 and at f_2 be x_2. We modify the execution time and energy consumption for task v as shown below.

$$x_1 + x_2 = Workload(v) \tag{2.14}$$

Table 2.2: Processor characteristics

Processor	Single issue, 5 stage pipeline, FPU
Registers	64 (32-bit)
Technology	90nm
L1 cache	16K I and D caches – 4 way set-associative
L2 cache 1	MB L2 cache – 16 way set associative

$$Execution\_time = x_1 * cp_1 + x_2 * cp_2 \tag{2.15}$$

$$Energy\_consumed = C_v * (\frac{x_1}{{cp_1}^2} + \frac{x_2}{{cp_2}^2}) \tag{2.16}$$

Here cp_1 and cp_2 are the clock periods associated with frequencies f_1 and f_2 and x_1 and x_2 are the variables in the formulation. The remainder of the formulation is the same as before. Note that the constraints are linear and the objective function is convex. By solving this second formulation, for a task v we can determine the number of cycles to run at each of the two possible frequencies. Although not shown here, we also consider the overhead associated with the frequency switch as described in section 2.4.4.

2.5 Experimental Results

We compare the results of our algorithm VAR-TB with a DVS algorithm which considers the worst-case only (WC-DVS), a worst-case DVS algorithm which performs dynamic slack reclamation [29] (WC-Reclaim), the method proposed in [93] (DynOpt) and a hypothetical method that can accurately determine the workloads of each task before hand and performs optimal scheduling (Oracle). WC-Reclaim allocates dynamic slack proportional to worst-case WorkLoad(v).

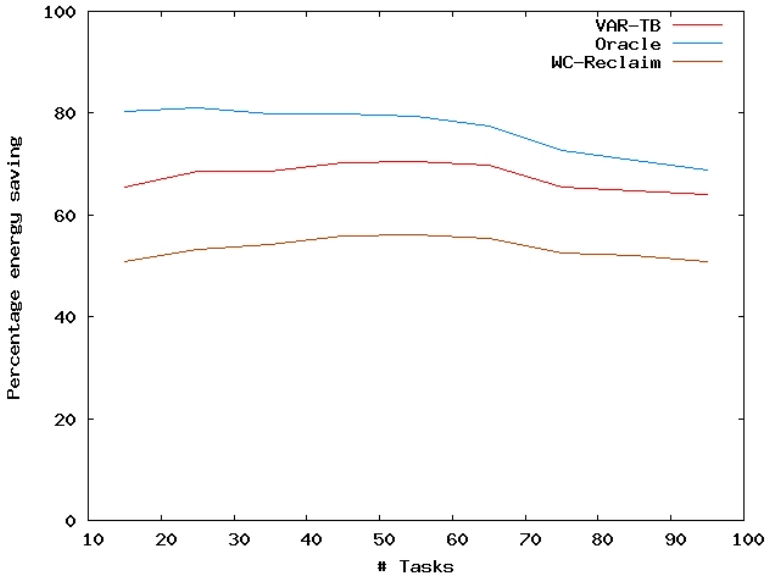

Figure 2.2: Energy savings over WC-DVS

2.5.1 Random task-graphs

We run our scheduling algorithm on several random task graphs generated from TGFF [47] with a resource constraint of 4 processors. We add the probability distribution of the task workloads as task attributes in the TGFF description. We obtain the probability distribution of $SCE(v)$ every task by performing a number of Monte-Carlo simulations (10,000 in our experiments). After the optimization step, we use Monte-Carlo simulations to compute the energy consumption and determine the average energy consumption. We compute the energy savings obtained by each algorithm and plot the savings as a percentage of energy consumption of the worst-case algorithm in Figure 2.2. The plot depicts the energy savings over algorithm WC-DVS for different task graphs. As can be seen, the simple algorithm WC-Reclaim performs much better than WC-DVS suggesting that the task graphs have significant variation in workloads to exploit. Our algorithm VAR-TB performs significantly better than algorithm WC-Reclaim; on an average the solutions provided by VAR-TB are 20-25% better than algorithm WC-Reclaim.

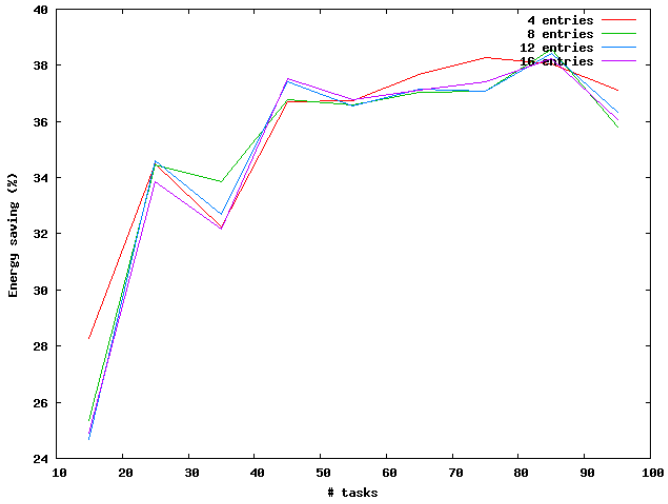

Figure 2.3: Varying the number of cores

Finally, we analyze the scheduling solution obtained by algorithm Oracle. As we can see, VAR-TB performs well compared to Oracle with the maximum deviation being 20% and the average deviation being 15%. Moreover, the quality of solutions provided by VAR-TB does not degrade with increase in the number of tasks in the task graph. Moreover, VAR-TB completes within 90 seconds for all task graphs. In the experiments presented, the number of $SCE(v)$ entries per task has been set to 16.

We have not plotted the results of algorithm *DynOpt*. We discovered that the pruning step for DynOpt proposed in [93] causes scheduling solutions that are inferior locally but optimal globally to be discarded. Such local pruning techniques cannot generate optimal solutions for scheduling problems. We found that DynOpt performs worse than WC-Reclaim in some cases. Moreover, DynOpt and [14] will require up to 520 optimization passes for the medium sized task graphs.

Next, we examine the effect of varying the number of $SCE(v)$ entries per task. The energy savings over WC-Reclaim are shown in Figure 2.3 for 4, 8, 12 and 16 entries per task. As we can see, our method of choosing the entries is effective – having 4 entries per task leads to a energy loss of only about 5% over having 16

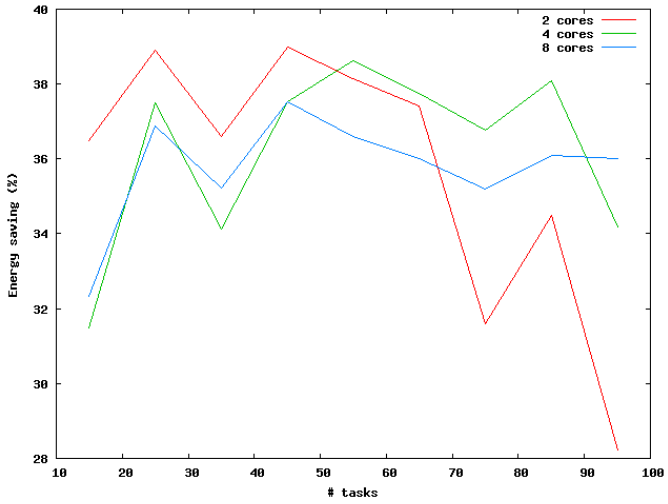

Figure 2.4: Varying the number of *SCE* values

entries per task. Finally, we examine the effect of varying the number of cores. We perform experiments for 2, 4 and 8 cores with 8 $SCE(v)$ entries per task. The results are shown in Figure 2.4. Our technique performs appreciably in all cases. The quality of the solution produced by our technique seems to improve when the number of cores is increased from 2 to 4 and then stabilizes.

2.5.2 Real-world Benchmarks

We perform experiments on two real-world applications – MPEG-4 decoder and Motion-JPEG (MJPEG) encoder. For these benchmarks, we apply dynamic slack reclamation after every algorithm. We evaluate two different schemes – W-Aware in which the workload of a task can be predicted from its input values and W-Unaware where the workload of the task cannot be predicted from its input values. Processor Architecture The processor cores in our experiments are modeled after the Intel XScale processor has 7 voltage levels as given in [29]. All processors share a 1 MB on-chip L2 cache through a common bus and implement a MESI cache-coherence protocol. Table 2.2 lists the relevant parameters. We use SESC [9] to simulate our multi-processor system and obtain profiling information (prob-

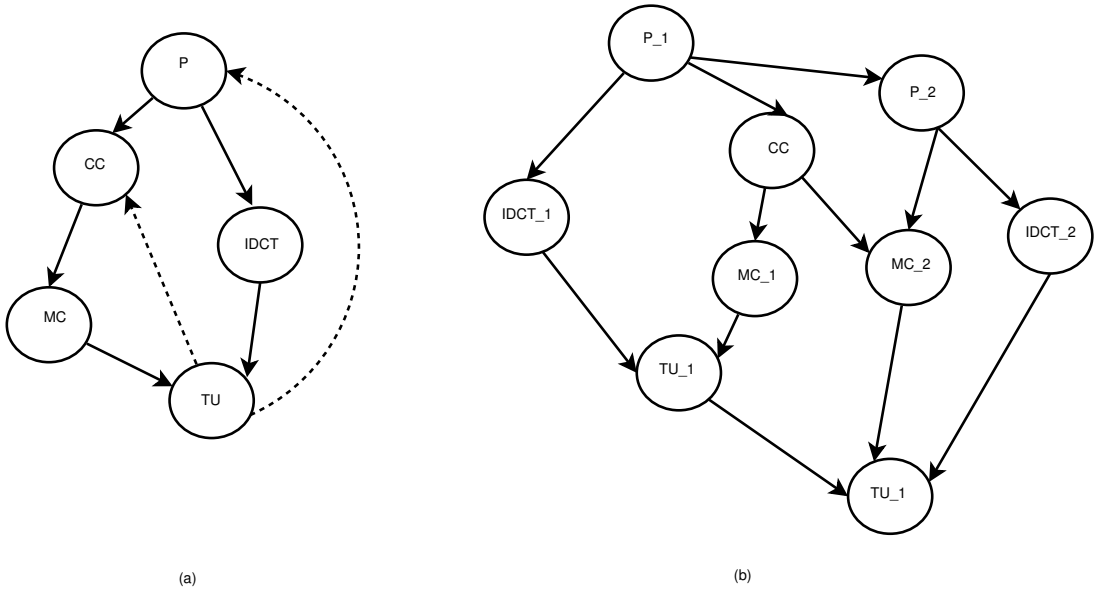

Figure 2.5: Task graph for MPEG-4 decoder (a) single iteration (b) two iterations

abilities and WorkLoad values).

For measuring dynamic energy consumption of on-chip components, we use Wattch [22] (integrated with SESC). Energy values for read-write operations for caches and SRAM-based array structures (TLB, ROB) are obtained from Cacti [82] for 90nm technology. For other processor components (such as ALU, decoder etc), energy numbers are obtained from Wattch for 180nm technology and scaling factors are applied as in [111]. Inter-processor communication is carried out through blocking FIFOs that are similar to the Fast-Simplex Link (FSL) [96] provided by Xilinx. We set the bandwidth of the FIFOs to be 300MB/s [96].

2.5.2.1 MPEG-4 decoder

A simplified task graph for the MPEG-4 decoder provided by Xilinx [100] is shown in Figure 2.5(a). The main components of the decoder are the Parser (P), Copy-Controller (CC), Inverse-DCT (IDCT), Motion Compensation (MC) and Texture

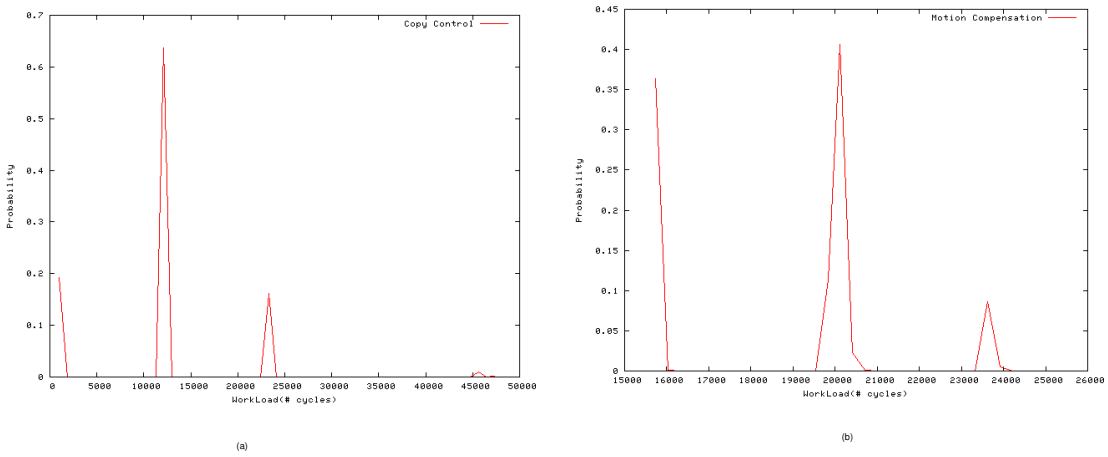

Figure 2.6: Probability distribution of workload (a) Copy control (b) Motion compensation

Update (TU). While IDCT does not show significant variation across different runs, the P, CC, MC and TU modules exhibit significant variation (Figure 2.6). By unrolling the loop for one macroblock and performing loop pipelining (Figure 2.5(b)), a parallel version of the decoder was implemented on a 7 processor system. A performance requirement of 20 frames/second was imposed on the decoder leading to a latency constraint of 500us per macroblock. We measure the energy reduction that our algorithm provides over the WC-Reclaim algorithm. Moreover, we measure how the quality of the solution is affected by the number of $SCE(v)$ values per task. The results are summarized in Figure 2.7. The two curves represent the energy consumption of the schedule produced by the two schemes – W-Aware (red curve) and W-Unaware (blue curve). From the plot, it is clear that our algorithm can achieve significant savings over the WC-Reclaim – up to 40% for W-Aware and up to 22% for W-Unaware. What is interesting is the fact that for modules such as IDCT and Motion-Compensation, the workload can be predicted very accurately from its input values primarily because of a nested if-else construct that depends on control signals that are inputs to the modules.

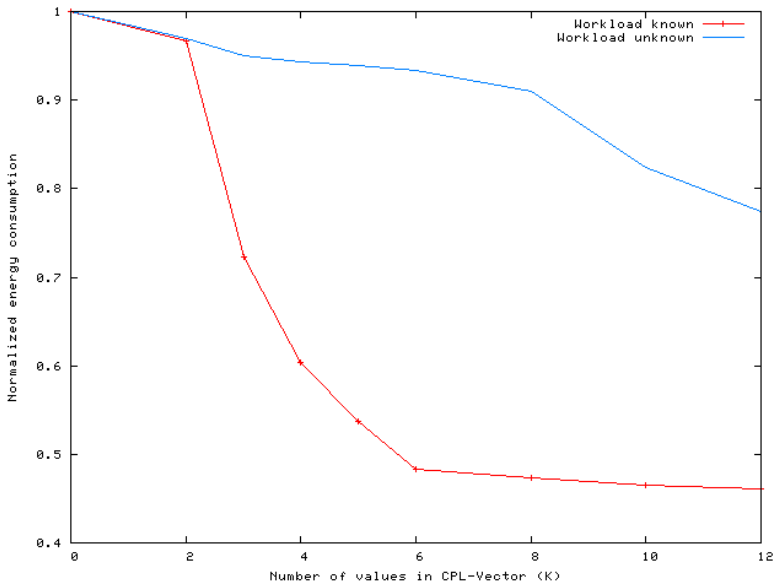

Figure 2.7: Normalized energy consumption for W-Aware and W-Unaware schemes for MPEG-4 decoder

2.5.2.2 MJPEG encoder

We apply our algorithm to the MJPEG encoder [45] for which the task graph is shown in Figure 2.8(a). The encoder is constrained to process Minimum Coded Units (MCU) of the incoming data stream in sequence. However, for processing a single MCU, we implement a pipelined version of the encoder using a four processor system where each task in Figure 2.8(a) is assigned to a processor. Of the four tasks, only the Huffman encoding task shows significant variation (Figure 2.9). We perform loop unrolling to obtain the task graph in Figure 2.8(b), on which we apply our We apply our algorithm to the MJPEG encoder [45] for which the task graph is shown in Figure 2.8(a). The encoder is constrained to process Minimum Coded Units (MCU) of the incoming data stream in sequence. However, for processing a single MCU, we implement a pipelined version of the encoder using a four processor system where each task in Figure 2.8(a) is assigned to a processor. Of the four tasks, only the Huffman encoding task shows significant variation (Figure 2.9). We perform loop unrolling to obtain the task graph in

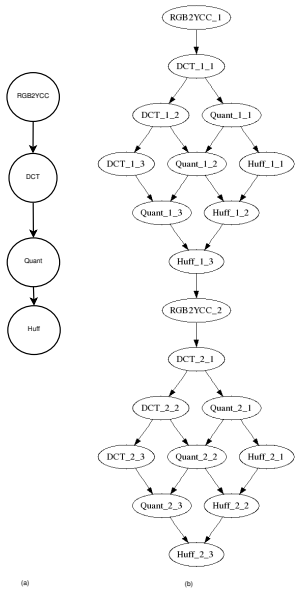

Figure 2.8: Task graph for MJPEG encoder (a) single iteration (b) unrolled

Figure 2.8(b), on which we apply our scheduling algorithm.

We compare the energy savings obtained from our algorithm against the WC-Reclaim algorithm. As explained before, we consider two cases – W-Aware and W-Unaware and vary the number of $SCE(v)$ entries. We show the energy consumption of the two schemes normalized to the energy consumption obtained by the WC-Reclaim algorithm in Figure 2.10. For the W-Aware scheme (red curve), we see that we can obtain up to 14% savings in energy whereas for the W-Unaware scheme (blue curve), the maximum savings we obtain is 4%. The small savings is because of the low variation seen in this benchmark.

2.6 Conclusions

We present a mathematical formulation which can exploit variation in workloads of tasks in applications to provide a low-energy scheduling solution. Our algorithm is capable of handling precedence constraints and multiple processors. We show that the schedule table can be generated in polynomial time and is optimal for trees.

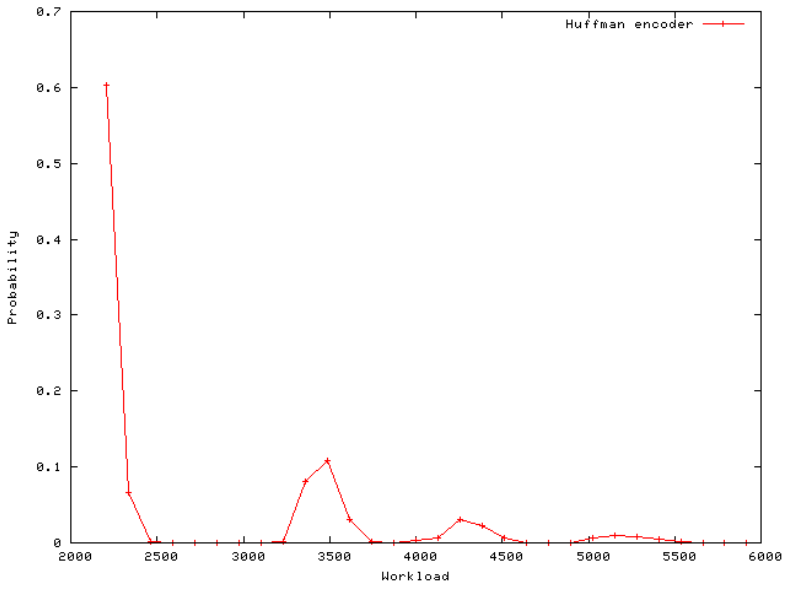

Figure 2.9: Workload variation of Huffman encoding module

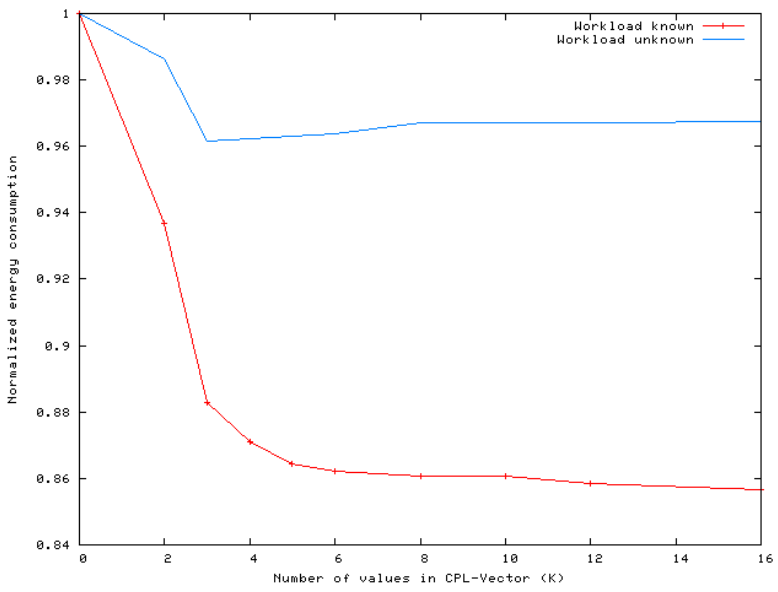

Figure 2.10: Normalized energy consumption for W-Aware and W-Unaware schemes for MJPEG encoder

Finally, our experiments show that significant energy savings can be obtained by our scheduling algorithm over worst-case only scheduling algorithm.

CHAPTER 3

Assuring Application-level Correctness Against Soft Errors

3.1 Introduction

While the previous chapter dealt with energy efficient computing in the presence of application-level variability, another important aspect of efficient computing is application-level reliability. With decreasing transistor sizes and supply voltages, CMOS devices face an increasingly higher probability of suffering from Single Event Upsets (SEUs) [4]. In theory, to ensure correct execution of programs, every operation needs to be duplicated and verified at runtime. However, the authors in [76] observe that the notion of correctness in programs can be defined in multiple ways. Quoting the authors from [76] – "*traditionally, a program's execution is said to be correct only if its architectural state is numerically perfect on a cycle-by-cycle basis. A similar (though slightly looser) notion of correctness requires a program's visible architectural state, i.e. its output state, to be numerically perfect. In both cases, correctness requires precise numerical integrity at the architecture level, which is a very strict requirement. We refer to the traditional notion of correctness as architecture-level correctness. Such strict assumptions regarding program output has led to conventional error detection and recovery techniques (such as triple (or dual) modular hardware/software redundancy (TMR, DMR)) that suffer from large performance and energy overheads. In many applications, even if execution is not 100% numerically correct, the program can still appear to*

execute correctly from the user's perspective [76][120]. Although such numerically faulty executions do not pass the test of architecture-level correctness, they may be completely acceptable at the user or application level. We refer to such a notion of correctness as application-level correctness. Good examples of such error tolerant applications are programs from the multimedia domain, where a few bits errors in an image or video stream may still be acceptable."

However, even applications that exhibit a high degree of error tolerance contain certain instructions (and program segments) that are required to be numerically correct for the program output to be acceptable. Such instructions are called application-level critical instructions, or critical instructions in short, in this paper. For example, in a MPEG-4 decoder, the finite state machine module, that determines control flow in the program is crucial for ensuring that the decoded video is of acceptable quality. Since real-world programs are typically far too complex for detailed manual analysis to determine critical instructions, automatic analysis techniques are required.

In this chapter, we focus on determining critical program segments which, when erroneous, will affect application level correctness. We assume that our application source code and the target hardware are correct, but transient errors (such as single event upsets due to high-energy particle strikes) are the sources of errors during run-time. Identifying critical instructions allows us to selectively replicate and verify critical program segments while executing a single, unchecked copy of the remainder of the program, thus greatly reducing overhead (in terms of time and energy) for transient error detection and recovery than the widely used TMR or DMR schemes.

3.2 Related work and our contributions

Several researchers have proposed techniques to distinguish between critical and non-critical instructions in programs. We briefly survey the existing techniques in this section.

3.2.1 Monte-Carlo based techniques:

The approaches proposed in [76][113][53][73][98] are based on extensive random, fault injection into program code and then observing their effect on program execution. In [76], based on these results, the program stack, register file contents and the PC, along with certain manually identified application specific data structures, are marked as part of critical state,. A detailed analysis of the frequency and type of abnormal program symptoms that are caused by errors is presented in [73]. The authors in [73][113][53] identify such symptom generating instructions and show that the probability of an error showing up as a symptom within a relatively small instruction window is high. In [98], likely program invariants are detected using Monte-Carlo simulation and checks are inserted to verify these invariants during program execution. Fault injections and analysis are performed at register transfer level by [112]. The advantage of Monte-Carlo techniques is that they are general and can be applied to any application; the downside is the high running time and the possibility of missing some critical instructions.

3.2.2 Program analysis techniques

In [53], the authors mark instructions that affect global variables and arguments to functions as high-value. Approaches proposed in [105][103] provide a simple static analysis technique wherein instructions that affect control flow and memory address computation are tagged critical and are marked for protection. However, such an approach might not be safe as some data flow critical instructions might

be missed. In [20], the authors analyze multimedia workloads that can tolerate errors, and propose exploiting this to address manufacturing defects. In [86][88], the authors use dynamic dependence graphs (DDG) to identify critical instructions and perform static analysis to determine all instructions that affect such instructions. However, as with Monte-Carlo based techniques, none of the above techniques can guarantee to identify all critical instructions. In [87], the authors use formal methods with symbolic expressions to obtain exhaustive error propagation and coverage metrics. But results are reported for small programs and it is unclear whether this technique can scale to large programs.

3.2.3 Using program invariants and patterns

The compiler research community has proposed several approaches for detecting errors in programs based on static analysis [44][52] using approximated fault models. Runtime error detectors are specified by the designer using rule-based templates in [65]. Daikon [50] and DIDUCE [63] are systems which dynamically detects program invariants. It is unclear whether all critical data and instructions can be protected by using such invariants. In [49], the authors try to learn common program patterns from the source code. Deviations from these patterns are tagged with a warning for possible errors. However, it would be very time consuming, if not impossible, for the user to specify all possible invariants for all critical instructions in a program.

In failure oblivious computing platforms such as in [95], the target platform is modified so that the faulty application can recover from a checkpoint even in the case of an instruction that causes fatal error (such as program termination). However, if the faulty instruction is critical in terms of application-level correctness, the output error can be arbitrarily large.

In contrast to previous work, our contribution is a highly efficient, profiling-

guided static program analysis technique and runtime monitoring approach that is guaranteed to identify all critical instructions in a program. In particular, our approach includes:

- A scalable program analysis phase that conservatively classifies instructions into 2 sets – *static critical (SC)* and *static non-critical (SNC)* based on the number of output that each instruction affects. The analysis is conservative because it might classify certain non-critical instructions as critical.

- A profiling phase that further divides the instructions in set SC into two subsets – *likely critical (LC)* (typically a small subset) and *likely non-critical (LNC)* based on the results of profiling.

- A lightweight run-time monitoring mechanism that tracks instructions that were classified as LNC by the static analysis to ensure that corrective actions are taken if these instructions become critical at run-time (in corner cases, non-profiled input sequences etc).

- We use the results of our analysis to ensure application-level correctness in the presence of soft errors. Instructions belonging to the set LC are duplicated and checked using the approach proposed in [83][94], instructions in LNC are monitored at runtime while instructions in set SNC are neither duplicated nor monitored. Together this ensures that all critical instructions are detected and verified at runtime.

Put together, our approach is shown to provide 21% reduction in energy consumption at run-time. Our approach is different from Monte Carlo based techniques, because profiling is used only to identify instructions that are likely non-critical (LNC), but cannot be proven by static analysis to be non-critical. Runtime monitoring is used to detect when such instructions might become critical (in rare cases) and to take corrective actions if necessary. This ensures that application-level correctness is ensured at run-time.

The remainder of this chapter is organized as follows. Section 3.3 describes certain preliminaries associated with our method. Section 3.4 provides a brief overview of our technique. Section 3.5 describes our static dependence analysis technique. Section 3.6 describes our lightweight runtime monitoring technique to detect if any likely non-critical instruction became critical. Section 3.7 describes experiments and results and finally we conclude the chapter in Section 3.8.

3.3 Program Representation

3.3.1 Preliminaries

Programs can exhibit enhanced error resilience at the application level when *multiple valid outputs* are permitted. In this paper, we say such programs have "elastic outputs". Elastic outputs commonly occur in programs computing results that are interpreted qualitatively by the user, such as multimedia applications and heuristic-based algorithms (such as genetic algorithms, loopy belief propagation and support vector machines) that attempt to solve complex problems for which absolute optimal solutions are too costly to compute. Programs with elastic outputs have application-dependent fidelity metrics associated with them to mathematically characterize the quality of the solution. Examples of fidelity metrics include PSNR (peak signal to noise ratio) for the multimedia applications, bit error rate for error correcting codes, etc. Application-level correctness can be defined in terms of the value of such fidelity metrics. Intuitively, these metrics estimate the overall quality of solution in the intended application domains.

3.3.1.1 Application-level correctness

Given an application A with:

- A vector of elastic outputs O.

- A set of outputs O_c that require numerical correctness.
- A fidelity metric $FA(I, O)$ associated with its input I and the corresponding elastic output O.
- A user-specified threshold T.

An output instance (O O_c) obtained by executing application A with input instance I in the presence of soft-errors is defined to be application-level correct if $FA(I, O) \leq T$ and O_c is numerically correct.

In general, the fidelity metric considers both the magnitude of error in each element of O and the number of elements that are erroneous. In our paper, we focus on the second part, as the impact of an soft error to an instruction output can be unbounded. More precisely, we assume that if an error in instruction x can propagate to output element o_i then the magnitude of error of o_i can be arbitrarily large and hence, the number of incorrect outputs determines whether the program execution satisfies application-level correctness. Previously proposed techniques in numerical analysis, such as those in [41], can be used to estimate the sensitivity of the error of o_i with respect to the error magnitude in x; however, there is one major difference from dealing with soft errors – in numerical analysis the inputs are assumed to have small error (precision error, which might be amplified by subsequent computations) and errors in subsequent computations are computed according to the assumed error propagation model. For soft errors, the error in any instruction can be arbitrarily large (depending on which bits were flipped). We also introduce the follow definitions that are used in the rest of the chapter.

- N_{min}: Given the maximum possible value of error for one output element (E_{max}), N_{min} is the minimum number of output elements that must be erroneous (each with error E_{max}) so that $FA(I, O)$ falls below the specified threshold T.

```
1 X=sqrt(Y);
2 for(i=1;i<N;++i)
3 {
4   C[i] = C[i-1] + i;
5   output[i] = C[i] + X;
6 }
```

Figure 3.1: Running example

- *Basic block*: A basic block in a program is a sequence of consecutive instructions that has one entry point, one exit point and no other branch instructions in between.

- *Instruction instance*: Instance of an instruction x in the program refers to a dynamic execution of the instruction at run-time.

- α-AFFECTER: An instruction x is said to be an $\alpha - AFFECTER$ of instruction y if an error in one instance of x can propagate to at least α instances of y.

- *Static instruction id (SID)*: Each static instruction is given a unique id – SID – at compile time.

Figure 3.1 shows a simple example where the array *output* can tolerate some errors. Assume that the each element of output is 10 bits wide and hence each element can have error no greater than 1024. Let the fidelity metric be the average error over all the elements of output. If the threshold value is T, then the value of N_{min} can be computed as $\frac{T*N}{1024}$. If N is 106 and T is 0.1, N_{min} is approximately 100. Figure 3.2 shows the LLVM IR for the above example. Intuitively, we can see that 100 *add* instructions in line 8 (Figure 3.2) would need to be erroneous to cause large degradation in output quality. On the other hand, a single error in the

```
1  entry:
2  X = call sqrt(Y);
3  bb:
4  i = phi [entry, 1] [ bb, i_inc]
5  c_i_1 = load &(C[i-1])
6  add_C = c_i_1 + i
7  store add_C, &(C[i])
8  c_i = load &(C[i])
9  out_i = add c_i, X
10 store out_i, &(output[i])
11 i_inc = add i, 1
12 cond = cmp i_lt, i_inc, N
13 br cond bb, exit
```

Figure 3.2: LLVM IR of the running example

computation of variable i (line 3) might cause significant degradation (especially if the error occurs in early iterations).

3.3.2 Program representation

We implement all our analysis techniques in the LLVM compiler framework [10] – however, our technique is not restricted to LLVM and can be applied easily to other compiler infrastructures. The LLVM intermediate representation (IR) is a static single assignment (SSA) based representation that essentially models a RISC processor with infinite registers where accesses to pointers (and arrays) are only through load/store instructions while all other instructions operate on register operands. Starting from the LLVM IR, we construct a weighted *program dependence graph* (PDG) $G(V, E, W)$ as follows:

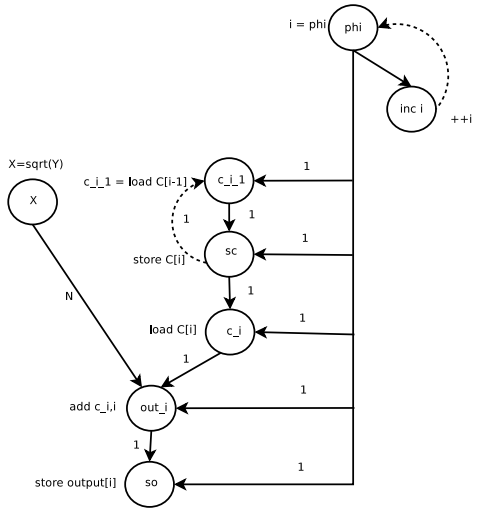

Figure 3.3: PDG of the running example

- Vertices V correspond to the instructions in the program, with one node for each instruction in the program. $I(u)$ represents the instruction corresponding to node u.

- Edges E are used to represent dependence among instructions in the program. An edge $e(u, v)$ is created between nodes u and v in the PDG if:

  - There exists a true data dependence between $I(u)$ and $I(v)$, or

  - $I(u)$ is a branch instruction and there exists a control edge between the basic blocks containing $I(u)$ and $I(v)$.

- Weight $w(u, v)$ for each edge represents the maximum number of instances of $I(v)$ that are affected by one instance of $I(u)$. Essentially, if an error occurs in one instance of $I(u)$ at run-time, $w(u, v)$ represents the maximum number of instances of $I(v)$ to which the error propagates to.

3.4 Overview of the proposed method

Our proposed method, named *CIAP (Critical Instruction Analysis and Protection)*, consists of the following steps

- Construction of the weighted PDG from the LLVM IR (section 3.4.2).
- Using the PDG to compute $N_{min} - AFFECTER$ for all outputs. As per the definition given above, the weighted PDG estimates the effect of an erroneous instruction on its immediate successors. Section 3.5 describes how this information is propagated through the PDG to estimate the effect on instructions that write to the final outputs. This step classifies instructions into 2 sets: *static critical (SC)* and *static non-critical (SNC)*. The classification is conservative in the sense that certain non-critical instructions might be marked critical.
- Profiling using given sample inputs is used to further refine the instructions in set SC. Instructions that are frequently seen to be critical are classified as *likely critical (LC)* while others are classified as *likely non-critical (LNC)*.
- Runtime monitoring is used to detect whether any instructions in set LNC become critical at runtime. This is described in section 3.6.
- Reliable execution is achieved by duplicating the instructions that are in set LC after step 3. Checks are inserted in the program for error detection and recovery (section 3.6.3). Together with the runtime monitoring system, errors (if any) in critical instructions are detected and corrected by re-execution.

3.4.1 Constructing PDG and computing edge weights

We introduce some terminology that we use in this section:

- Loop nest vector $L_{vec}(v)$ $(L_1, L_2, ..., L_k)$ for node v represents the loop nest inside which node v resides. The outermost loop is the first element of the loop vector.

- Trip count of loop L, $TC(L)$ is the number of iterations executed by loop L. If this value cannot be determined statically, then it is assumed to be a large value TC_{max}.

3.4.2 Constructing PDG

The construction of the PDG, apart from edges connecting store and load operations, is trivial given a SSA-based intermediate representation. For each LLVM instruction v, an edge e is added from every operand u to v. For load-store instructions, we use LLVMs alias analysis to determine whether the address for the load and store instructions can overlap. If the alias analysis returns true and if the object accessed is an array and the array index is an affine function of loop index variables, we further check for dependence using the Omega library. The Omega library [11] is a tool capable of fast analysis of affine integer constraints. We use the Omega library to determine loop carried and loop independent memory dependences as follows: let the memory addresses accessed by a store and a load instruction be given by the affine expressions $a_1^\top * I_1 + b_1$ and $a_2^\top * I_2 + b_2$ respectively (where I_1 and I_2 are the iteration vectors of the loop nest for the two accesses). For loop carried dependences, we set the constraint that iteration vector I_2 should be lexicographically greater than I_1 ($I_2 \succeq I_1$) and $a_1^\top * I_1 + b_1 = a_2^\top * I_2 + b_2$. Together these two constraints imply that the load should be executed after the store in the loop nest and the address accessed by both the instructions is identical. We also add constraints on the loop bounds of the iteration vectors (if loop bounds are constants). The Omega library can determine if the mathematical relation exists between I_1 and I_2 such that the above mentioned constraints are satisfied. If the

Omega library returns a feasible solution, it implies that loop-carried dependence exists for the store-load pair. Hence, we add a new edge in the PDG between the nodes corresponding to the load-store pair. Similarly, loop independent dependences are computed using the Omega library as well. If the array access is not an affine expression of the loop index variables or the object being accessed is not an array (generic pointer), then we make a conservative assumption and assume both loop carried and loop independent dependence between the load-store pair over all common loops in $L_{vec}(s)$ and $L_{vec}(l)$.

Figure 3.3 shows the partial weighted PDG for the example in Figure 3.2. The dotted edge indicates that it is a loop carried dependence edge between the store (node 3) and load instruction (node 2), and the solid edges represent loop independent dependences.

3.4.3 Computing edge weights - Static method

We describe how we compute the edge weights of the PDG based on a number of different cases. First, we introduce the definition of Max_live_reads:

$Max\_live\_reads(s, l)$ is the maximum number of instances of load instruction l that read the live value produced by store instruction s. Essentially, this number estimates the number of instances of l that are impacted by an error in store instruction s. In general, this is a difficult quantity to compute - we describe two scenarios in which this quantity can be efficiently computed. If neither scenario occurs, we conservatively set this value to a large number MLR_{max}.

- Scenario 1: instructions s and l are in the same basic block, have identical address operands and s is before l in the basic block. In this case, the $Max\_live\_reads(s, l)$ is 1 because a new instance of s is always executed before l and both access the same address. For the example in Figure 3.3, for one instance of node sc (store C[i]), there will be one instance of node c_i

(load C[i]) which reads the value of the store in the same iteration. Hence, $Max\_live\_reads(sc, c\_i)$ is set to 1.

- Scenario 2: instructions s and l are in the same basic block, access the same array object and the array access indexes are affine functions of the loop indexes (given by $a_1^\top * I_1 + b_1$ and $a_2^\top * I_2 + b_2$). A polytope can be constructed using equality constraint on the array access indexes ($a_1^\top * I_1 + b_1 = a_2^\top * I_2 + b_2$) and lexicographic ordering constraints on the iteration vectors ($I_2 \succeq I_1$). In this case, the Barvinok library [109] can be used to get an upper bound of the number of integer points within this polytope, each of which is a legal solution to the above constraints. By assuming unbounded loops, for a given value of $I1$, this number is an upper bound on the number of instances of l that will read the value generated by one instance of s. For the example in Figure 3.3, for one instance of node sc (store C[i]), there will be one instance of node c_i_1 (load C[i-1]) in the next iteration which reads the value of the store. Hence, $Max\_live\_reads(sc, c\_i\_1)$ is set to 1 and $e(sc, c\_i\_1)$ is a loop-carried dependence edge.

The LLVM IR contains three kinds of nodes that need to be considered specially while assigning weights to dependence edges – SSA phi node, load and store instructions (that provide read and write access to pointers and arrays). We identify three cases for edge $e(u, v)$:

- Case 1: v is not a load or phi node. Let $L_{vec}(u)$ and $L_{vec}(v)$ be the loop nest vectors for the two nodes, such that $L_{vec}(u)$ and $L_{vec}(v)$ are identical up to the i^{th} entry and are different after i. Then, $w(e)$ is given by $\prod_{k=i+1}^{m_v} TC(L_k)$ which is the product of the trip counts of the loops of which v is part of but u is not. Note that if v is not part of any loop nest or is part of an outer loop relative to u then this value evaluates to 1. For the example in Figure 3.3, node X is outside the loop nest that contains node out_i (add X, c_i).

Hence, $w(X, out\_i))$ is set to N because any error in the computation of X will affect N instances of the add instruction (out_i = add X, c_i).

- Case 2: v is a phi node and e is the dependence edge corresponding to control edge b for v. If b is not a back edge, then $w(e)$ is computed in the same manner as Case 1. If b is a back edge corresponding to loop L_q (of which v is also a part) and $L_{vec}(u)$ and $L_{vec}(v)$ are loop vectors for nodes u and v that are identical up to the i^{th} entry and are different after i, then $w(e)$ is set to $\prod_{k=i+1}^{q} TC(L_k)$. Essentially, this value computes an upper bound on the number of times the control-edge b is executed for one instance of node u. Case 3: v is a load instruction. If u is the instruction that computes the address from which data is loaded, edge e is handled in the same way as Case 1. If u is a store instruction, which may alias with the address of the load, then the $w(e)$ is set to $Max\_live\_reads(u, l)$.

3.5 Computing α-AFFECTER from weighted PDG

In this section, we present an algorithm *Propagation* to determine whether an instruction x is a $\alpha - AFFECTER$ of instruction y given a weighted PDG G. We first assume that G is a directed acyclic graph (DAG) and then relax this assumption.

3.5.1 Acyclic PDG

Algorithm *Propagation* iterates over the nodes in topological order. The value $propagate(root \rightarrow v)$ represents the maximum number of instances of node v to which an error in a single instance of $root$ can propagate to. At first, all values of propagate are set to 0. $propagate(root \rightarrow root)$ is set to 1 (because in a DAG 1 instance of a node affects only 1 instance of itself). For a node u, the

value of $propagate(root \rightarrow v)$ has been computed for all predecessors v (since we traverse G in topological order). By definition, $propagate(root \rightarrow v)$ represents the number of instances of v to which an error from a single instance of *root* can propagate to and $w(v, u)$ is the number of instances of u to which an error from a single instance of v can propagate to. Hence, the maximum number of instances of u to which an error from a single instance of *root* can propagate to through *nodev* is simply the product $w(v, u) * propagate(root \rightarrow v)$. We take the maximum over all predecessors v to get an upper bound on the number of instances of u to which an error in a single instance of *root* can propagate to. More precisely:$propagate(root \rightarrow v) = max_{u \in predecessors(v)} propagate(root \rightarrow u * w(u, v)$

3.5.2 PDG with cycles

Our goal is to modify the PDG by adding new edges and updating some edge weights so that all cycles can be removed. Existence of cycle C implies that an error propagated to any single node instance in C will propagate to all node instances in C (in the worst case). Hence, any node that can affect one node of the cycle must affect all the nodes of the cycle. This information must be reflected in the modified PDG. Since the PDG has cycles, the original program must have loops. This implies that for each cycle C in the PDG G, there exists an edge b_e which corresponds to the back-edge of the loop L associated with the cycle C. We first add edges to all nodes in C from nodes which are not part of C but affect at least one node in C. More precisely, $\forall u \in S = \{x | \exists\ v\ s.t.\ e(x \rightarrow v) \in E \wedge v \in C \wedge x \notin C\}$, create edges $e\prime(u \rightarrow v)$ for all and set $w(e\prime)$ to $TC(L) * w(b_e)$. This ensures that any error from a node outside the cycle will be propagated to all nodes in the cycle and the number of instances affected is upper bound by the product of trip count of the loop L and weight of the back-edge. We can then delete b_e. By repeating this process for all cycles, we finally end up with a DAG, for which we can reuse algorithm *Propagation*. The only difference is the

initial value of $propagate(root \rightarrow u)$ for nodes that were part of the cycle C, the initial value is set to $TC(L) * w(b_e)$. This implies that any error in one instance of *root* will propagate to all instances of u in the cycle of which both u and *root* are part of (in the worst case). For the PDG shown in Figure 3.3, the loop carried dependence between nodes 2 and 3 makes the value of $propagate(3 \rightarrow 3)$ and $propagate(2 \rightarrow 3)$ to become N. Thus, nodes 2 and 3 are marked critical - however, node 1 does not become critical as $propagate(1 \rightarrow 5)$ is simply 1.

3.5.3 Identification of critical instructions

Let $NC = (i_1, i_2, ..i_n)$ be the instructions that directly write to outputs that are required to be numerically correct and $J = (j_1, j_2, ..j_m)$ be those that directly write to the vector of elastic outputs. An instruction x is marked critical if:

- Rule 1: x is 1-affecter for any instruction in NC i.e. $\sum_{k=1}^{n} propagate(x \rightarrow i_k) \geq 1$ OR
- Rule 2: x is $N_{min}-affecter$ for any instruction in J i.e. $\sum_{k=1}^{m} propagate(x \rightarrow j_k) \geq 1$ OR
- Rule 3: x is $1-affecter$ for any instruction marked critical by the above 2 rules.

3.5.4 Control flow optimization

The procedure described in the previous sections marks certain non-critical control flow statements conservatively as critical. We define a critical basic block as a basic block that contains at least one critical instruction. Consider the control-flow graphs (CFG) shown in Figure 3.4 where each block represents one basic block. Critical basic blocks are shaded. In Figure 3.4(a), assume that block D contains an instruction X that has been marked critical according to rule 1 or

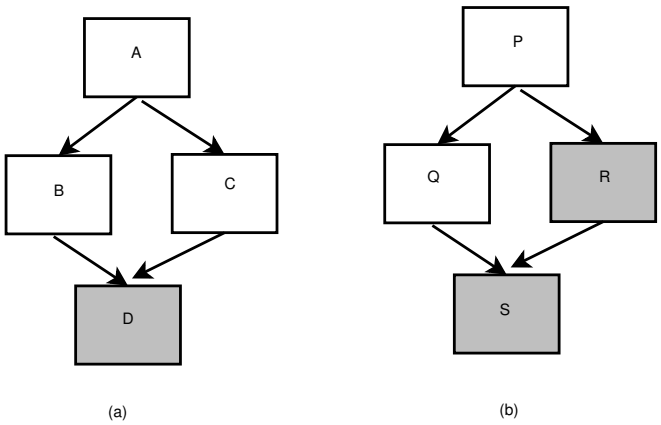

Figure 3.4: PDG of the running example

rule 2 in section 3.5.3. Let us also assume that blocks B and C do not have any instructions that are marked critical according to Rules 1 or 2. Rule 3 would lead to the branch instructions in B, C and A to also be marked critical (since they are $1 - affecters$ of the critical instruction X in D). However, the control flow from block A reaches critical block D irrespective of the direction taken by the branch instruction at the end of A. Since blocks B and C do not really have any critical instructions, an error in As branch instruction will not cause an error at the application-level. Thus, As branch has been conservatively marked critical. In contrast, in Figure 3.4(b), block R is critical and hence block Ps branch becomes critical (along with all instructions that affect Ps branch). We now present an analysis technique for further identifying non-critical control flow code segments. To detect non-critical control flow, we first unmark branch instructions that have been marked critical only by Rule 3 in section 3.5.3. We use the concept of post-dominator from control-flow analysis. A basic block B is said to strictly *post-dominate* block A if and only if all control paths from A to the end must pass through B and B is not equal to A. The immediate post-dominator of a block A ($ipdom(A)$) is the unique block B that strictly post-dominates A but does not strictly post-dominate any other block that strictly post-dominates A. The main idea behind our technique is to determine whether from a given basic block A,

control flow can reach a critical basic block before it reaches $ipdom(A)$. If not, As branch is not critical because no matter what path the control flow takes from A, it does not reach a critical basic block before it reaches $ipdom(A)$.

Figure 3.5 shows the outline of our analysis technique $C-Opt$. In the Initialize step (line 1), basic blocks are marked critical based on the conditions:

- All basic blocks containing at least one critical instruction (section 3.5.3) are marked critical.
- All basic blocks that exit from loops that contain at least one critical basic block are marked critical.
- All basic blocks that contain an exit instruction (exit function, return etc.) are marked critical.

Our algorithm then iterates over all basic blocks (lines 4-6) and for each block A tries to find whether control flow from A can reach a critical basic block before $ipostdom(A)$ (line 7). This step is a simple depth-first search in the CFG starting at A. If yes, then A is marked as a critical block and As branch instruction (A_{br}) is also marked critical (lines 9-11). All instructions that are 1-affecters of A_{br} (denoted $DS(A_{br})$) are also marked critical as are the basic blocks that contain these instructions (line 12). The algorithm terminates when no more basic blocks are marked critical (line 15). The algorithm is guaranteed to terminate because in the worst case all blocks will be marked critical.

```
    Initialize
2:  while new critical blocks found do
      for i = 1 → num_blocks do
4:      A ← block[i];
        if reachesCriticalBeforeIPDom(A) then
6:        Mark A as critical basic block;
          A_br ← As branch
8:        Mark A_br as critical;
          Mark DS(A_br) as critical;
10:     end if
      end for
12: end while
```

Figure 3.5: Control-flow optimization algorithm - C-opt

3.6 Assuring application-level correctness - profiling and runtime monitoring

3.6.1 Profiling edge weights

As is true for any static analysis, the results of the static dependence analysis might be overly pessimistic. We identify two main sources of conservatism that might lead to high edge weights resulting in several instructions being marked critical conservatively.

- Conservative edge weights – Conservative alias and control flow analysis techniques might set certain edge weights much higher than they occur during program execution.
- Correlated edge weights – It is possible that two dependence edges are never active simultaneously during program execution due to control flow, data

dependent address computation etc.

In our approach, we track edge weights that are overestimated by static analysis. Determining correlations between edge weights requires path tracking which we do not investigate in our work. Profiling is used to estimate edge weights of two kinds of edges – edges between a store and load instruction and control dependent edges with a phi node as their end-point. We compare weights of edges obtained during profiling with those estimated by static analysis. Any edge whose weight obtained from profiling is 25% of the value estimated by static analysis is identified as a conservative edge. We then substitute the new values of edge weights obtained from profiling for conservative edges and re-run the *Propagate* technique described in section 3.5. It is possible that certain instructions which were marked critical earlier by static analysis are now no longer critical. We mark these instructions as *likely non-critical (LNC)* to be monitored at runtime.

3.6.2 Runtime monitoring of edge weights

At runtime certain edges may have higher weights for certain corner input cases, which would make the conclusions of the profiling-guided criticality analysis invalid. Hence, we propose the use of a lightweight runtime scheme that can monitor edge weights and trigger a signal when a edge weight increases above a threshold (which might make certain instructions critical). We explain how we track edge weights for dependences between a store and load instruction. Each memory location has a tracker structure associated with it – the tracker has two fields a *src* field to store the *static instruction id (SID)* of the instruction writing to the address and a counter field to determine the number of times the memory location is read before being overwritten by a new value. When a store instruction writes to an address, it sets the *src* field of the tracker to its SID value and resets the counter to 0. When a load instruction reads from an address, it reads the *src*

field associated with the tracker and increments the value of the counter. The *src* id along with the load instructions SID describes the dependence edge. The *src* SID and the load instructions SID together uniquely determine the dependence edge and the value of the counter represents the edge weight. For each edge being monitored at runtime, a threshold value is assigned. As long as the edge weight is below the threshold, the results of the profiling guided analysis hold and all instructions in set LNC are still non-critical. If the edge weight crosses the threshold value, then there is a chance that certain instructions in the set LNC may become critical. The threshold value is computed at compile time – it is set to 80% of the edge weight obtained during profiling.

Accessing the counter location given the source and destination SIDs at runtime needs to be efficient. We use a perfect hashing scheme [90] to achieve this – a perfect hash is feasible since all the possible likely source ids given the destination id is already known after profiling.

In our method two versions of the code are compiled – version $V1$ which assumes that all instructions are critical and version V2 which assumes that only instructions classified as *likely critical (LC)* after the profiling phase are critical. Version $V2$ also contains runtime monitors to track dependence edges associated with *likely non-critical (LNC)* instructions. To begin with, version V2 is executed. If an unexpected dependence edge is seen during program execution, then we switch to version V1 as we can no longer guarantee non-criticality of instructions in version V2 due to the unexpected dependence. We observe two common patterns of accesses to array/pointer locations that would benefit from monitoring. The first pattern involves array/pointer objects that are accessed in a streaming fashion. The full input stream might be very large but the block of data (in the array/pointer) being processed in one stage is relatively small. For such arrays, it is important to keep track of accesses to individual words, because each word is read and overwritten frequently – the tracker counters associated with each word

need to be reset to 0 on every store instance (that writes to the particular word). The second pattern involves accesses to large sized pointer/array objects whose read counts are uniformly distributed across all the individual elements and read counts per element are rarely large enough to make store operations to these locations critical. For objects with such access patterns, maintaining a single counter for a contiguous region of memory is expected to provide good estimates of edge weights. Offline profiling can be used to estimate whether accesses to a particular array/pointer fit into one pattern or the other.

3.6.3 Ensuring application-level correctness

We use the results of our analysis to obtain application-level correctness using the approach in [83]. All instructions in the set LC are duplicated and their results are compared before writing to memory. If any errors are detected, then execution is rolled back to the start of the basic block and instructions are re-executed. Runtime monitors are inserted to detect whether any instructions in set LNC become critical. Finally, the instructions in set NC are not checked (since any errors in NC will not affect application-level correctness).

3.7 Experiments and Results

We assume our program will run on a commercial off the shelf (COTS) processor platform where the memory structures (caches, external memory) are protected by ECC. We measure the energy overhead for achieving reliable execution by simulation using Winsconsin's GEMS simulation infrastructure [80] and McPAT [74] for power estimation. We simulate a 4-issue processor with 64 KB L1 caches and 4MB L2 cache. Some of the relevant configuration parameters of the processor system which we simulated are given in Table 3.1.

Table 3.1: Simulation parameters

Parameter name	Value
Issue-width	2-issue superscalar processor
Physical registers	224
L1 cache	64 KB, 4-way set associative, 64 byte block
L2 cache	1 MB, 4-way set associative, 64 byte block

3.7.1 Error injection methodology

For each instruction in IR-level representation, based on the given error probability (0.01% in our case) and the number of times the instruction is executed (obtained using a profiling run), we use a pseudo-random number generator to determine whether that particular instruction should be impacted by a single-event upset. If yes, a new function call is inserted into the program – the new function call invokes a pseudo-random number generator and determines whether to inject an error or not at run-time. Using real soft error rates for applications would take a long time for errors to appear; hence we use 0.01% as a sample value. For actual soft-error rates, we refer the reader to [75]. For all applications, the maximum allowed fall in quality of solution is set to 5%.

3.7.2 Illegal memory accesses

In [73], the authors showed that when an error at the RTL-level for a processor is not masked, it appears in software primarily as an illegal memory address leading to a program crash. However, since many instructions are non-critical, an application-level correctness aware system would ignore memory accesses with faulty address values. Fortunately, current processor ISAs such as SPARC v9 [12] provide non-faulting loads (for optimized compilation) – load instructions which will execute correctly when the memory address is legal and will be ignored

when illegal. Such non-faulting loads have no overhead relative to normal load instructions when the address is legal.

We investigate the effectiveness of our technique on multimedia benchmarks from MiBench [7]. DCT, Huffman coding and ycc2rgb are important kernels in the JPEG decoder. Susan is an image recognition program with kernels for detecting edges and corners in images. G721 is a voice compression application while GSM implements a decoder for the GSM communications standard. LDPC [5] (low density parity check) is a linear error correcting code. H264 is a video decoder from the MediaBench suite [6] while libmad is an open source mp3 decoder.

Table 3.2 shows the results obtained by static analysis and profiling. Columns 1 and 2 show the total number of instructions and basic blocks in the program. Columns 3 and 4 show the number of instructions marked critical during the static analysis and control flow analysis phase respectively. Column 5 shows the number of basic blocks marked critical during our control flow analysis phase. Column 6 shows the number of instructions which became LNC after profiling phase. The primary reason for the large number of instructions marked critical during static analysis is the conservative nature of alias analysis. Profiling helps reduce weights of certain edges which leads to many instructions being marked as LNC, particularly for the larger benchmarks.

Table 3.3: Comparison with [83]: columns under 'Error-free' and 'Errors inserted' show the # instructions executed without and with errors at runtime respectively, column titled 'Energy reduction' shows the energy reduction relative to [83]

Benchmark	Error-free		Errors inserted		Energy reduction (%)	
	CIAP	Previous [83]	CIAP	Previous [83]	Static only	Static +monitor
IDCT	1.036	2.00	1.037	2.001	95	95
Huffman	1.761	2.00	1.78	2.02	32	32

Table 3.3: Comparison with [83]: columns under 'Error-free' and 'Errors inserted' show the # instructions executed without and with errors at runtime respectively, column titled 'Energy reduction' shows the energy reduction relative to [83]

Benchmark	Error-free		Errors inserted		Energy reduction (%)	
	CIAP	Previous [83]	CIAP	Previous [83]	Static only	Static +monitor
ycc2rgb	1.15	2.00	1.16	2.00	98	98
GSM	1.343	2.00	1.352	2.02	104	104
G721	1.665	2.00	1.687	2.00	63	63
Susan-edges	1.015	1.99	1.016	2.001	107	107
Susan-corners	1.045	2.00	1.046	2.03	110	110
LDPC (LBP)	1.098	1.99	1.1	2.007	103	103
Rician denoise	1.57	2.00	1.62	2.08	8.23	42
H264	1.63	2.00	1.68	2.00	7.10	51
Libmad (mp3)	1.41	1.99	1.49	2.002	4.56	71
Average	1.34	2.00	1.36	2.01	66.53	79.63

Table 3.3 shows the overhead (in terms of number of instructions executed at run-time and energy) of duplicating critical instructions to achieve application-level correctness with respect to the approach proposed in [83]. The approach in [83] protects all instructions in the program and guarantees architectural correctness. Columns 2 and 3 show the number of instructions (relative to the single correct copy) executed at run-time when no errors are inserted into the application for our approach and the previous work [83]. Columns 4 and 5 show the

Table 3.2: Static instruction classification

Benchmark	Instructions	Basic blocks	Critical data flow	Critical control flow	Critical blocks	LNC
IDCT	224	4	0	8	4	-
Huffman	443	76	0	309	66	-
ycc2rgb	85	8	27	17	7	-
GSM	2533	451	2	923	151	-
G721	1077	215	52	554	121	-
Susan -edges	1299	44	7	34	14	-
Susan -corners	993	106	14	30	15	-
LDPC (LBP)	200	29	145	19	9	122
Rician	832	87	72	12	17	64
H264	50431	7731	18654	24321	5845	14312
Libmad (mp3 decoder)	12349	2335	6086	3129	1945	4045

data for the case when errors are injected into the application. As can be seen, for the random error injections, the overhead of recovery is small primarily because of the relatively small number of critical instructions and the relatively small number spent in re-execution. Columns 6 and 7 show the energy savings achieved by our method compared to the approach in [83] – column 6 shows the impact of static analysis only while column 7 considers both static analysis and run-time monitoring. On the average, our technique provides 79.63% energy reduction compared to [83] while guaranteeing application level correctness.

Table 3.4: Comparison with [103]: columns under 'Error-free' and 'Errors inserted' show the # instructions executed without and with errors at runtime respectively, column titled 'Energy reduction' shows the energy reduction relative to [103]

Benchmark	Error-free		Errors inserted		Energy reduction (%)	
	CIAP	Previous [103]	CIAP	Previous [103]	Static only	Static +monitor
IDCT	1.036	1.036	1.037	1.037	0	0
Huffman	1.761	1.82	1.78	1.83	3	3
ycc2rgb	1.15	1.12	1.16	1.12	-1.7	-1.7
GSM	1.343	1.99	1.352	2.01	34	34
G721	1.665	1.96	1.687	1.99	17	17
Susan-edges	1.015	1.997	1.016	2.030	63	63
Susan-corners	1.045	1.984	1.046	2.012	63.3	63.3
LDPC (LBP)	1.098	1.63	1.1	1.64	5	24.7
Rician denoise	1.57	1.92	1.62	2.1	0	19.3
H264	1.63	1.91	1.68	1.98	0	32

Table 3.4: Comparison with [103]: columns under 'Error-free' and 'Errors inserted' show the # instructions executed without and with errors at runtime respectively, column titled 'Energy reduction' shows the energy reduction relative to [103]

Benchmark	Error-free		Errors inserted		Energy reduction (%)	
	CIAP	Previous [103]	CIAP	Previous [103]	Static only	Static +monitor
Libmad (mp3)	1.41	1.62	1.49	1.65	0	21
Average	1.34	1.72	1.36	1.76	16.69	25.05

Table 3.4 shows the overhead (in terms of number of instructions executed at run-time and energy) of duplicating critical instructions to achieve application-level correctness with respect to the approach proposed in [103]. The approach in [103] protects only control flow and memory address computations and does not guarantee application-level correctness. Columns 2 and 3 show the number of instructions (relative to the single correct copy) executed at run-time when no errors are inserted into the application for our approach and the previous work [103]. Columns 4 and 5 show the data for the case when errors are injected into the application. As can be seen, for the random error injections, the overhead of recovery is small primarily because of the relatively small number of critical instructions and the relatively small number spent in re-execution. Columns 6 and 7 show the energy savings achieved by our method compared to the approach in [103] – column 6 shows the impact of static analysis only while column 7 considers both static analysis and run-time monitoring. On the average, our technique provides 25% energy reduction compared to [103] while guaranteeing application level correctness.

3.7.3 Analysis of results

For applications such as DCT and YCC2RGB in JPEG, the data flow is non-critical and only loops form the critical program segments. These applications have affine accesses to arrays and hence static analysis is able to perform well. GSM and G721 operate on small chunks of data (16-130 bytes) and almost all dependences which are not well analyzed by static techniques affect the control flow. A few control statements are marked non-critical by our control-flow analysis technique. The kernels in Susan have affine accesses with a large number of instructions affecting control flow – however, each control decision affects only one output element and each decision is independent. Thus, our control flow analysis can save a significant number of critical instructions. Static analysis performs adequately for these applications. Static analysis performs poorly for the remaining benchmarks. LDPC is pointer-heavy with linked lists being used for representing a sparse matrix. Denoising on the other hand is an iterative algorithm with a check for convergence. Although it shows strong loop-dependence, the number of iterations, before convergence, is rarely large enough (even in presence of errors) for an error to propagate to a large number of elements. H264 and libmad (mp3 decoder) are streaming applications; however, static analysis is not able to perform effective alias analysis due to the complex nature of the code involved. For these applications, the runtime monitoring scheme is useful as it helps to overcome the overly conservative nature of static analysis. Table 3.5 shows the overhead associated with runtime monitoring (for the applications that benefited from runtime monitoring). The main source of overhead is the write and read of the source and destination SIDs in memory. Hashing is relatively fast. For our set of applications, we did not any edge weights crossing their threshold values. This might be a characteristic of media benchmarks – other applications such as simulated annealing might behave differently.

Table 3.5: Overhead associated with runtime monitoring

Benchmark	Overhead (# dynamic instructions)	Performance overhead
LDPC	9%	11%
Denoising	6%	9%
H264	13%	16%
MP3	8%	9.2%

3.8 Conclusion

In this chapter, we presented a hybrid analysis technique which analyzes programs with elastic outputs to determine critical instructions. Our approach consists of a static analysis phase which can identify critical instructions and mark those which are identified as critical because of the conservative nature of static analysis. This is followed by a profiling phase that determines whether instructions conservatively marked critical by static analysis do become critical at run-time – if not, these instructions are monitored by a lightweight run-time system. We use the results of our analysis to reduce the number of replicated instructions and show significant benefits over previous approach while guaranteeing application-level correctness.

CHAPTER 4

Architecture support for custom instructions with memory operations

4.1 Introduction

The previous two chapters focused on energy efficient computing in the context of application level variability and reliability. However, the target platform on which the applications were executed was a general purpose processor based system (chip multiprocessor or CMP). When we talk of energy efficient computing, hardware specialization has been a very powerful tool. Fixed-function accelerators are custom processing units which are highly efficient (in terms of energy) and fast at performing tasks for which they are designed [40] (compared to software running on conventional processor cores). However, typical coarse-grained accelerators are not very flexible – they are unable to handle computations apart from those for which their data-path and control flow are designed for. To bridge the gap between coarse-grained accelerators and software, instruction-set customization has been proposed in [21][32][1][13][99][36][15]. Custom instructions (CIs) enhance the conventional processor pipeline by allowing certain patterns of operations to be executed efficiently on custom functional units (CFUs) added to the processor pipeline. CFUs are typically small compared to coarse-grained accelerators – most CFUs comprise of a few operations and operate on a few inputs (< 10) compared to coarse-grained accelerators which operate on hundreds of input data elements and typically consist of hundreds of operations. This makes it possible

for compilers to automatically extract computation patterns from programs and map them to the CFUs. In our work, we target systems where the CFUs can be implemented in reconfigurable fabric that is tightly integrated with the processor pipeline.

Integration of CFUs with a superscalar pipeline provides additional opportunities – typical superscalar processors have hardware for speculatively executing instructions and rolling back and recovering to a correct state when there is mis-speculation. Speculation may involve branch prediction or load speculation – by load speculation we mean the scenario where a load instruction can read memory before the address of a preceding store instruction is even computed (operating under the assumption that the store and load will not access the same address). Using a content-addressable memory structure (CAM) called load-store queue (LSQ), the processor can detect conflicts and recover in case of mis-speculation. In our work, we propose an architecture for integrating CFUs with the superscalar pipeline to exploit the processor's hardware for speculation and recovery. Several previous works have proposed architectures for supporting processors with CFUs – in the following section, we review related work and discuss their limitations to motivate our work.

4.2 Related work and our contributions

4.2.1 ALU-like CFUs

In such systems, the CFUs read (write) their inputs (outputs) directly from (to) the register file of the processor and cannot access memory – hence the term ALU-like CFUS. Such an architecture has been used in several previous works [15][36][32]. This architecture makes it simple to integrate the CFU with a superscalar pipeline – the processor micro-architecture can perform register renaming, branch prediction and control speculation in a manner transparent to the CFU.

However, since the CFU cannot access memory, the size of the computation pattern that can be implemented on a CFU is constrained by the number of ports in the register file. A variety of architectures have been proposed to overcome this bottleneck – shadow register files [38][119], clustered register files [68] and spreading the register file reads (writes) across multiple cycles [92][48].

4.2.2 CFUs with memory operations

The primary problem with allowing CFUs to launch memory operations is to make sure that memory is updated and read in a consistent fashion with respect to other in-flight memory instructions in the superscalar pipeline. In the GARP [64] processor, the reconfigurable array can launch memory operations – however, the assumption is that all memory operations within the processor pipeline have completed before launching the CI and the processor pipeline cannot launch any memory operations while the CFU is computing. In the VEAL design [31], the authors assume that the compiler and hardware together take care of synchronization between the CFU and the processor pipeline and memory consistency is maintained using the system's cache coherence protocol. In the OneChip [23] processor, the compiler needs to insert special instructions in the program to mark all the memory blocks which may be read or updated by the CFU – the memory controller uses this information to ensure the correct ordering of operations within the pipeline. The limitation of this architecture is that the addresses of all possible memory blocks that may be touched by a CI needs to be computed before launching the CI itself.

4.2.3 CFUs with Architecturally Visible Storage (AVS)

The idea of AVS is to make a memory block inside the CFU visible to the processor micro-architecture so that data can be transferred to/from the memory block

from/to the processor's memory hierarchy in a consistent fashion. The CFU can then launch memory operations which read/write from/to the AVS block. In [17], the compiler inserts special instructions to explicitly transfer data from/to the CFU's AVS – thus, the compiler needs to identify which memory blocks are read by a CFU, which software instructions update the memory blocks before the CFU is launched and insert the transfer instructions after the update instructions have completed. Coherence was addressed by the authors in [70] who introduced a MESI coherence mechanism between the AVS and the processor cache. In an extension to that work, the authors in [71] propose minimizing the overhead of the coherence protocol between the AVS and cache by allocating some cache lines for the AVS. However, in all these approaches, it is the responsibility of the compiler to ensure that synchronization instructions are inserted in the program between the CI and inserting transfer instructions at the correct position.

4.2.4 Context-full CFUs

The authors in [114] proposed an architecture by which the CFUs can possess context/state (to minimize the number of transfers between the CFU and processor memory/register file) and can be launched speculatively. The proposed hardware contained support for storing CFU context and recovering it when there was a mis-speculation. However, no other details of the interaction between the CFU and the processor pipeline were provided.

4.2.5 Our contributions

In our paper, our goal is to design an architecture such that CIs with memory operations can execute seamlessly along with other instructions in the processor pipeline without any special synchronization operations. Our goal is to introduce minimal changes to a conventional superscalar core architecture and re-use most

of its component. More precisely,we present an architecture for integrating CFUs with the processor pipeline with the following properties:

1. CFUs can launch memory operations to directly access the L1 D-cache of the processor – our CFUs have no internal memory or context which removes the need for coherence between the CFU and D-cache.

2. CFUs do *not* have to specify which memory addresses they access before executing the CIs. This is specially useful for pointer heavy applications [106] where the address of memory accesses cannot be determined ahead of time.

3. *No synchronization instructions need to be inserted before /after the CI* – this greatly reduces the burden on the compiler specially for applications with operations whose memory address cannot be determined beforehand. Our architecture uses the components of the superscalar pipeline to ensure the correct ordering among the different memory operations.

4. The effect of doing away with synchronization instructions is that more aggressive speculative pipelining of CIs is possible for 'hot' loops in a program. CIs no longer need to wait for earlier CIs to complete.

5. Multiple CFUs can issue memory operations simultaneously in a speculative fashion – again, our architecture leverages and extends the superscalar pipeline components to determine conflicts and the processor's rollback and recovery mechanism to restart from a safe state in case of mis-speculation.

Our goal is to add CFUs to a relatively *small superscalar core (for example, a 2-issue wide core)* and achieve the *performance of a larger core (for example, a 4-issue wide)* with *significant savings in energy consumption.* Additionally, we show that our architecture can be used for *applications with complex memory*

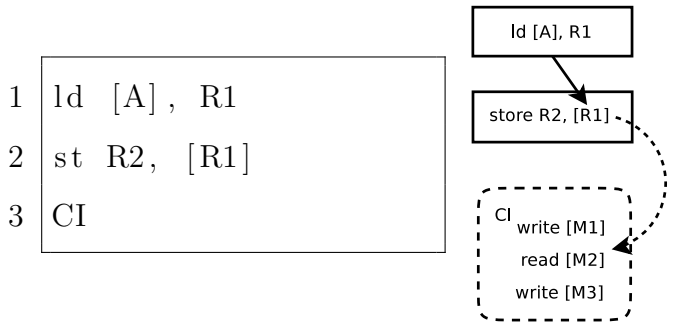

Figure 4.1: Memory ordering example

access patterns and achieve good performance gains while previous approaches do not perform well for such applications.

The rest of the paper is organized as follows – section 4.3 describes our main contribution in supporting memory operations in CIs. Section 4.4 describes some implementation details in our flow, section 4.5 describes the architecture of our processor which is tightly integrated with the CFUs. Section 4.6 describes our custom compilation flow for extracting CIs from computation-intensive loops of benchmarks so that they can be efficiently executed on the processor system with CFUs. Results are presented in section 4.7.

4.3 Challenges and our proposed solution for supporting memory operations in CFUs

Execution of ALU-like CIs, which only read from/write to the core's registers, is well understood – the conventional register renaming mechanism takes care of dependences between the CI and other instructions in the pipeline. In this section, we explain the issues associated when CFUs connected to an out-of-order (OoO) core are allowed to launch memory operations directly. As we explain the issues, we propose modifications in the compilation flow as well as the underlying micro-architecture of the core to support such CFUs. A more detailed view of the

architecture is described in section 4.5.

Basic working: A CI is essentially a set of operations grouped together by the compiler to be executed as a single instruction. The primary inputs for a CI always come from the registers of the processor – all input registers must be ready before a CI is ready to execute. Once a CI starts executing, it can issue a series of memory operations into the processor's pipeline. The addresses of these memory operations need not be known before the CI starts executing – the CI can supply the addresses as it executes. Outputs of the CI can be writes to the processor's registers or write operations to memory.

4.3.1 Issue 1: Maintaining program order for memory operations

Typically, when CIs are inserted inside a sequential code block, the user expects that memory is updated in program order. Consider the simple example in Figure 4.1 – the CI to be executed launches 3 memory operations, a read from location $M2$ and writes to $M1$ and $M3$. Note that the preceding store instruction could possibly write to the same location as $M2$ (shown as a dotted arrow in the dependence graph). For previous work [23][70], the CI needs to wait at least until the addresses of *all* preceding memory operations are computed before proceeding. However, the address of the preceding store instruction is provided by the preceding load instruction – in case of a cache miss, the load instruction could take several cycles to complete. Since the core is OoO and the CI may begin executing before the store instruction's address has been computed, the only way to achieve program correctness is through compiler inserted synchronization instruction(s) before the CI. This would force the CI to wait till the address of the store has been computed, even if the CI does not really depend on the store instruction. Note that even if the compiler determines address $M2$ to be a constant, the synchronization instruction is still needed since the compiler cannot determine the address of the store instruction. This could lead to idle resources in the pipeline

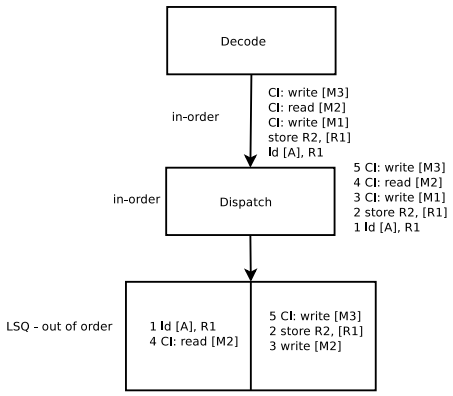

Figure 4.2: Ordering of instructions in the pipeline for the example in Figure 4.1

while the preceding addresses are being computed. Using a coherence protocol [70] reduces the burden on the compiler in transferring data between the CFU and L1 cache, but by itself does not guarantee correct program order ([70] uses synchronization instructions provided by the compiler) .

We overcome this limitation by modifying the core micro-architecture – the key difference of our approach is to make the memory operations launched by a CI ***visible to the OoO processor's pipeline in program order***. In the decode stage of the pipeline (see Figure 4.2), in which program order is still maintained, the CI in Figure 4.1 would launch 3 memory operations (which we call *mem-rops*) into the dispatch stage. In the dispatch stage, each operation is assigned a tag representing its program order. Since the store instruction would have been decoded/dispatched before the CI, it would have been assigned a preceding tag. The OoO pipeline would assign 1 entry for the store and 3 entries for the CI in the LSQ – the LSQ would contain the address, a valid bit to determine whether the address is known and the instruction tag. If the addresses of the CI memory operations can be computed (within the CI) before the store instruction's address, they are allowed to execute. Once the store instruction's address is determined, the LSQ will check whether any successive operations have an address conflict –

in case of a conflict, the OoO pipeline would ensure that the CI is squashed and a pipeline flush occurs. If not, the instructions wait for retirement.

Note that because we made the ordering of memory operations launched by the CI explicit to the OoO pipeline, the hardware is able to determine when a data hazard occurs without any information from the compiler (no explicit synchronization needed). Also, the CI can execute before the preceding address values are known.

4.3.2 Issue 2: Ordering of memory operations within a CI

For the CI in Figure 4.1, assume that the first write operation to address $M1$ and the read operation reading from $M2$ overlap/conflict. In case of normal memory instructions, if the read executes before the write, the read would be squashed and re-executed in the OoO pipeline. However, the instruction stream of the program contains only the CI and not the individual memory operations. Re-execution would need to begin from the CI and the same conflict would occur again. To overcome this problem, we place a constraint during the CI compilation phase – the compiler cannot cluster a memory operation in a CI if there is a preceding memory write operation within the CI which may cause a conflict. The compiler uses alias analysis (possibly in a conservative way) to satisfy this constraint. Memory dependences between different CIs are handled in the same way as described in section 4.3.1.

4.3.3 Issue 3: Possible partial commit to memory

For the CI in Figure 4.1, assume that the first write operation to address $M1$ commits and updates memory. However, after this commit, it is determined that the write operation to $M3$ fails because of a TLB translation fault. For a normal instruction, the pipeline would be squashed and the TLB translation fault would

be serviced by a page walk. However, for an operation launched by a CI, re-execution would begin from the CI again. This would leave the memory in an inconsistent state since the write to address $M1$ was committed. We overcome this problem by delaying the commit of all write memory operations launched by the CI till the successful completion of the CI (no TLB faults).

4.3.4 Issue 4: Handling TLB faults

Memory operations launched by a CI can cause TLB faults – we handle this from the compiler side. The compiler assigns an additional output register for the CI. When there are no faults, the CI writes 0 to this register, else it writes the logical address which caused the TLB translation fault. All memory operations launched by the CI are squashed (no update to memory). The compiler inserts four additional instructions – a comparison to see whether the output register contains a non-zero value, a branch if-not zero instruction, a load instruction for the logical address which caused the fault and an unconditional branch back to the CI. Basically, the program checks whether a TLB fault occurred (given by the non-zero test of the output register), performs a software read where the TLB fault is handled and re-executes the CI.

4.3.5 Issue 5: Handling variable number of memory operations

Since CIs can span multiple basic blocks in the program, the number of memory operations launched by a CI could vary across executions and need not be deterministic before the CI starts executing. This issue is solved by launching the maximum number of memory operations that a CI can execute during the decode stage. These memory operations wait in the processor pipeline till the CI supplies valid addresses. In case a particular memory operation is not executed, the CI supplies a 'dummy' address for these operations effectively turning them

Table 4.1: Alias information for benchmarks – columns titled 0-10 show the fraction (as percentages) of memory dependences sorted by iteration distance. The last column shows the ratio of dynamic/static memory dependences (expressed as percentage).

Benchmark	0	1	2	3	4	5	6	7	8	9	≥10	Dynamic /Static (%)
bzip2	8.71	9.8	4.2	0	0	1.54	0	3.23	0	0	72.52	14.34
libquantum	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	100.0	5.74
mcf	0.00	17.51	10.46	7.21	4.33	3.81	1.76	1.55	0.86	0.99	51.52	0.16
gobmk	72.40	13.77	3.24	1.60	0.94	0.58	0.45	0.36	0.31	0.23	6.12	17.86
h264	88.75	7.91	0.36	0.21	0.15	0.12	0.18	0.11	0.11	0.11	2.00	23.45
sjeng	55.87	21.74	2.67	1.76	1.50	1.09	1.48	0.70	0.66	0.62	11.93	11.78
hmmer	14.34	7.8	2.3	3.7	0	0	0	0	0	0	71.86	16.49

into nops. To bound the maximum number of memops that a CI can execute, our compiler ensures that control flow within a CI never follows a loop backedge.

4.3.6 Scenarios where our architecture would beat a system with compiler inserted synchronization

Our architecture would be beneficial if memory dependences predicted by the compiler are very conservative with respect to actual dynamic memory dependences in the program. In Table 4.1, we analyze integer benchmarks from the SPEC 2006 suite and show the distribution of dynamic memory dependences across loop-iteration distances for the 10 'hottest' loops in each program. For example, the number in column titled 0 for h264 implies that for 88% of all memory dependences (read after write) for the top 10 loops, the write and read operations were executed in the same iteration of the loop (hence iteration distance 0). The last column shows the ratio of the number of memory dependences observed at

runtime to the number of memory dependences predicted by the compiler (alias analysis). The relatively low values in the last column show that the alias analysis of the compiler is conservative when it comes to estimating RAW hazards for memory operations. If the compiler was expected to figure out all the memory dependences while generating custom instructions (like in OneChip [23]), then a large number of 'synchronization' instructions would need to be inserted into the code. In our experiments, we use the the full alias analysis from LLVM to estimate dependence at compile time.

The distribution of memory dependences across iteration distances in loops, shows us that there is significant opportunity for speedup using loop pipelining with CFUs – if a large percentage of memory dependences had a small dependence distance (1 to 3), then loop pipelining would not be efficient since too many iterations would depend on near previous iterations. Note that memory dependences with iteration distance 0 are not an issue with respect to loop pipelining. Our results are consistent with those reported in [106] regarding speculative pipelining.

4.3.7 Difference with CISC ISAs

Our method of using *mem-rops* is similar to *micro-ops* used by x86 based processors. CISCinstructions (like x86) are typically split into relatively simple *micro-ops* – typically an address computation *micro-op*, a memory read/write *micro-op* and possibly an arithmetic *micro-op*. The CIs in our system differ from CISC instructions in the following ways: (1) A CI can launch an arbitrary number of *mem-rops* which may complete in any arbitrary order. (2) Memory read *rops* launched by a single CI can have dependence between them as well (for example $ptr \rightarrow next \rightarrow X$). (3) A CI can span multiple basic blocks in the program and it is possible that the actual number of *mem-rops* executed varies from one execution of the CI to another. The reason we used *rops* is because we want to minimize the changes needed in the core for supporting memory operations from

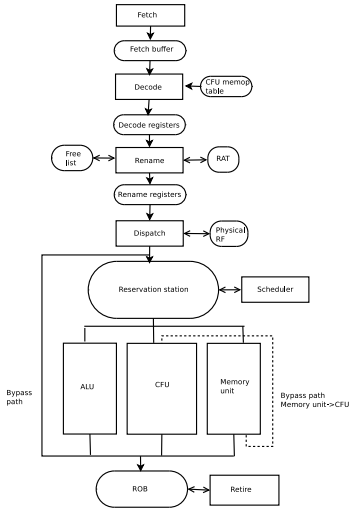

Figure 4.3: Layout of the processor pipeline with tightly integrated CFU within a CI.

4.4 Custom instruction operation and representation

In our system, a reconfigurable datapath is integrated tightly within the processor pipeline – the logical layout is shown in Figure 4.3. Physically, our system is a tiled chip multiprocessor (CMP). Individual tiles may be processor cores, L2 cache banks or tiles of reconfigurable fabric (see Figure 4.4). A core can implement custom instructions in an adjacent tile containing reconfigurable fabric. The baseline core in our system is a 2 GHz 2-issue superscalar processor. Consider the Xilinx Virtex-6 XC6VSX315T FPGA which has around 50K slices and 1300 DSP blocks in a 23mm by 23 mm die. In our system, we estimate the area occupied by a core (using McPAT [74]) and compute the number of slices/DSPs that could fit in this area as per the density of the above mentioned Virtex-6 FPGA die – this turned out to around 2000 slices and 150 DSP blocks. From McPAT [74], we determine that it would take 5 processor cycles to communicate between the reconfigurable fabric and core pipeline to transfer values from/to the FPGA in

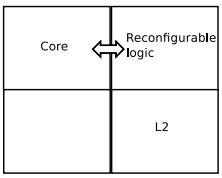

Figure 4.4: Tiled CMP with reconfigurable logic

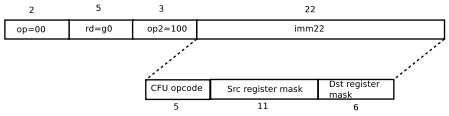

Figure 4.5: Custom instruction format

a pipelined fashion. More details of the communication are provided in section 4.5.6.

At the start of an application, the datapath is re-configured to implement the CFUs which execute the CIs selected by the compiler for that particular application (section 4.6). Reconfiguration time for modern FPGAs is around 1 ms [77] and is small enough to be ignored if performed once per application. We assume that the processor clock and the CFU clock arise from the same master clock and the processor frequency is an integer multiple of the CFU frequency. This may reduce the frequency at which CFUs are able to run, but makes the interface between the clock domains simpler [110] – essentially all that is needed is a register which can accept new data from the processor at a rate equal to the integer multiple. More details of the interaction between the CFU and processor pipeline are described in section 4.5.

Since memory operations launched by the CI can take arbitrary time to complete (cache misses etc), each CFU is not a simple deterministic datapath – the synthesized circuit contains a finite-state machine with dedicated states in which the CFU stalls while the launched memory operations are completed.

Each CI has a code associated with it which tells the hardware which CFU

to run the CI on. This code is part of the binary representation of the CI. In our setup, we use the SPARC v9 ISA and use the nop instruction "*sethi* $< imm\_value >$, *%g0*" provided to implement CIs – see Figure 4.5 for the instruction format of *sethi* (the numbers above the rectangle specify the number of bits) . This instruction provides 22 bits for representing the immediate value – we use 5 bits for the code of the CI, 11 bits as a bit vector to determine which registers are inputs to the CI and 6 bits as another bit vector for determining which registers are outputs of the CI (Figure 4.5 lower rectangle). For example, in the 11 bit source register bit-mask, if bit number 8 is set to 1 (numbering assumed to start from 0), then this implies that register $r8$ is one of the inputs to the CI. Note that the SPARC ISA has 32 registers – however, under our setup, CIs can only use 11 of them as inputs and 6 as outputs (in our experiments, we use registers 8 to 18 for inputs and 19-24 for outputs). This imposes a hard limit on the number of register inputs/outputs that custom instructions can have in our architecture.

4.5 Details of proposed architecture

Our baseline processor is a superscalar processor with support for branch prediction (and rollback-recovery) and out of order execution of memory operations. Figure 4.3 shows the basic layout of how our reconfigurable CFU units interact with the rest of the processor pipeline – the rest of the section briefly explains each component of this interaction. There are two main issues we need to consider in our architecture: (1) Each CI can have more than 2 source registers and more than 1 destination register (unlike conventional instructions). (2) Each CI can launch 1 or more memory operations. For the purpose of illustration, we assume that our baseline processor is a 2-issue wide superscalar processor.

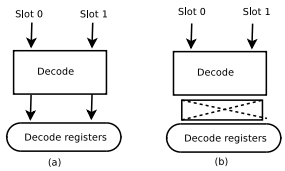

Figure 4.6: Decoder modification (a)baseline(b)modified

4.5.1 Fetch stage

Our baseline processor can fetch upto two instructions every cycle from the I-cache (assuming cache hit), perform branch prediction, update the PC and store the fetched instruction in the fetch buffer. The fetch buffer is a circular FIFO – the head (updated by the decode unit) points to the next instruction to be decoded while the tail (updated by the fetch unit) points to the last instruction fetched. The fetch unit ensures that the head of the fetch buffer is not overwritten when the decode stage is stalled. We make no changes to the fetch stage.

4.5.2 Decode stage

Our baseline processor's decoder has two decode slots and hence, can decode two instructions every cycle from the fetch buffer (moving the head pointer of the fetch buffer by two every cycle). The decoded instructions are placed into two *decode registers* which are read by the rename logic in the subsequent cycle. The rename logic checks for dependencies between the two instructions and renames the source registers by accessing the *register alias table (RAT)* which stores the mappings between architectural and physical registers. The destination registers of each instruction are assigned a physical register from the *free list buffer* (which is maintained as a circular FIFO). Since we are using the SPARC ISA, an instruction can have at most 3 source registers (store integer instruction) and 1 destination register. We assume that a 2-issue wide processor can rename upto 2x3 source registers per cycle and can assign upto 2x1 destination physical registers per cycle.

Since CIs may contain upto 11 source registers, 6 destination registers and multiple memory operations and the number of ports of the register file (and other components in the micro-architecture of a superscalar processor such as the RAT, free list, reservation station) is limited, additional changes are needed in the micro-architecture to support CIs. Essentially, we split a complex CI into multiple simple operations which we call *rops* (for RISC operations). This approach is similar to decoding performed in x86 processors where a complex instruction is split into multiple simple *micro-ops (uops)* [3]. However, the difference is that for our CIs the micro-ops are used only for reading from/writing to registers and memory while the computation is performed by customized logic on the FPGA.

When the decode stage encounters a CI, it determines the number of register operands from the binary representation of the instruction (see Figure 4.5) . The opcode of the CI (see section 4.4) is used to determine the number and type (read/write) of memory operations that the CI will launch. This operation is essentially a table lookup – we assume a 1-cycle latency for this operation (bit vector to denote the type of operation). We introduce a SRAM table called *CFU memop table* to store the mapping between the CI opcode and the number of memory operations launched by the CI – this is a programmable table which is filled up when a new application is launched.

Consider an example CI with 8 register inputs, 3 register outputs and 4 memory operations on a 2-issue wide baseline processor. Our baseline ISA is SPARC v9 where an instruction can have at most three source registers and one destination register. For a fair comparison, we use the same constraints and hence, a *rop* can have at most three source registers and one destination register. Hence, the decode logic creates 3 *sreg-rops (source register rops)*, 4 *mem-rops* (read operations first followed by write operations), 3 *dreg-rops (destination register rops)*. For a 2-issue processor, assuming the CI is decoded in decode slot 0, upto 2 rops (and/or instructions) can be processed every cycle. Hence, for our example CI, it would

take 5 cycles to insert all the *rops* into the pipeline. Also, we introduce two additional bits in the decode registers to mark the first and last *rops* of a CI.

A CI can appear at any of the two inputs of the decoder in a 2-wide processor while the *rops* of the CI can be output at any of the two outputs of the decoder. This implies that we need a crossbar at the end of the decode stage because the rop issued by any decode unit should be able to enter any of the two decode slots available (see Figure 4.6).

Using a circular FIFO for the fetch buffer allows the decode stage to throttle the fetch stage when the buffer is full. However, if the pipeline design is similar to the Alpha-21264 [69] with a single wide register to store the fetched instructions every cycle, our design needs another modification. An additional wide register is introduced to copy the fetched instructions every cycle – if the fetched instructions contain a CI, the fetch unit is stalled and the decoder will use the duplicate register in subsequent cycles till all the instructions in the duplicate register are decoded fully.

4.5.3 Rename stage

The *sreg-rops* and *dreg-rops* are renamed in the same way as normal instructions. *Mem-rops* are ignored completely by the rename stage.The source operands for these *mem-rops* (address and/or data) are provided by the CI after partial execution. Making sure that the *mem-rops*' source operands are passed correctly is handled by the dispatch stage and the scheduler.

4.5.4 Dispatch stage

The dispatch stage is the last stage in the pipeline where instructions are processed in order (apart from the commit stage) – each instruction is assigned a *sequence id* or *SID* which represents the program order. The *SID* is typically the index in

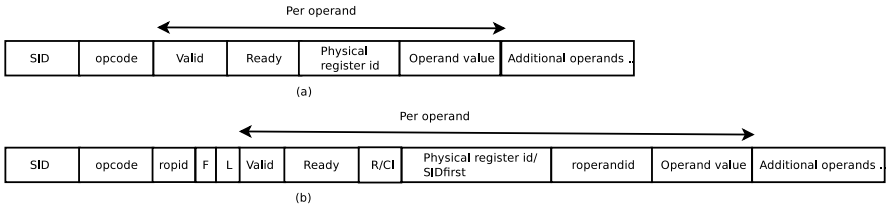

Figure 4.7: Reservation station entry format (a)baseline(b)modified

the ROB in which the instruction is stored.

The main job of the dispatch stage is to perform resource allocation – create entries in the ROB (and assigning *SIDs*) for each instruction/*rop*, the reservation station and in the LSQ for memory instructions/*rops*. Since *mem-rops* get their address (and data) operands directly from the CIs (and not register file), each entry in the reservation table needs to be expanded to accommodate this information. We explain with an example: consider a modern superscalar processor with 256 physical registers and a 128-entry ROB. A typical reservation station (RS) contains entries whose format looks similar to the one shown in Figure 4.7(a) – each entry has 3 fields for the source operands, each field stores the physical register id and value of the source operands (and/or immediate operand), a valid bit and a ready bit (labeled V and R respectively). For our example processor, the width of the physical register id field would be 8 bits.

In our modified ISA, the number of destination registers has been limited to 6 and source registers to 11 (see section 4.4). From section 4.4, each CI can have at most 16 *mem-rops*. Thus, each CI can have at most 4 *sreg-rops*, 6 *dreg-rops* and 16 *mem-rops*. Each *mem-rop* can have two source operands – address and data (for memory write operations). Thus, to correctly address the source operand of a *mem-rop*, the following information is needed:

- *SID* of the CI which provides the source operands: We use the *SID* of the first *rop* to represent the *SID* of a CI. We denote this by SID_{first}. Since an *SID* is the index of the *rop* in the ROB, the bitwidth for *SID* in our example

system is $log_2(128)$ bits.

For normal instructions, each source operand is simply the physical register id – to maintain compatibility with the field for *SID* is expanded to $1 + log_2(max(256, 128))$ bits, with one additional bit to determine whether the source operand is a physical register or is provided by a CI.

- Index of the *rop* inside the CI – we call this the *ropid*. In our example, the CI could provide data to upto 16 *mem-rops*, 4 *dreg-rops* or the 6 *dreg-rops*. Thus, $ceil(log_2(26))$ bits are needed to represent the *ropid*.

- Operand index within a single *rop*: For example, a *mem-rop* has upto two source operands (address and data) and hence, 1 bit is needed to mark the operand index within a single rop.

This brings the bitwidth of each source operand id to 15 bits while baseline processor (without CIs) would use only 8 bits for representing source ids (physical register ids).

For each *rop* that passes data to the CI, the *rop* needs to know the *SID* of the CI i.e. SID_{first}. This information is used to correctly route the results of *sreg-rops* and *mem-rops* over the bypass path to the CFU executing the CI. Additionally, since a CI may launch multiple *sreg-rops* and *mem-rops*, each *rop* must know its own index within a CI. Since a CI can have at most 26 *rops*, we would need 5 bits to represent the index of the *rop* within a CI or *ropid* as described in the previous paragraph. Normal instructions in the baseline processor have one field to represent the destination physical register id. We extend this field to also include the additional information required for CIs. In our baseline example processor, the destination register id field is $log_2(256)$ bits wide. In our system, this field is extended to $1 + log_2(max(256, 128)) + 5$ bits – the one additional bit is 1 if the destination is a CI, 0 otherwise.

From McPAT [74], we observe that the area of the reservation station in the baseline processor would be 0.17 mm^2 while the modified reservation station would occupy 0.21 mm^2 at the 45 nm node.

4.5.5 Scheduler and execute stage

When a particular *sreg-rop* becomes ready (all its source register operands become ready), assuming the CFU corresponding to the CI is idle or the CI corresponding to the *sreg-rop* has already been assigned to the CFU, the scheduler may decide to assign the CFU to that particular CI. Communication to the CFU occurs over the 5-cycle link mentioned in section 4.4. The correspondence between an *sreg-rop* and the issuing CI is obtained by reading the SID_{first} value from the destination field as described in subsection 4.5.4.

Preventing deadlocks: Consider an example, where two CIs C_1 and C_2 that runs on the same CFU enter the pipeline in-order and launch one *sreg-rop* each, given by r_1 and r_2 respectively. It is possible that r_2 is ready before r_1 (cache miss etc) and the scheduler reserves the CFU for C_2. However, if there is a dependence between C_1 and C_2, then C_2 would never complete execution on the CFU and C_1 would never execute (as C_2 has reserved the CFU) leading to a deadlock. The scheduler makes sure that such a deadlock never happens by reserving a CFU for a CI only when all the *sreg-rops* for the same CI are ready.

Once the CFU has obtained all its register operands, it begins execution. When a memory operation is encountered, the CFU computes the address (and data) for the operation and sends it over the bypass path to forward it to the *mem-rop* waiting in the RS. The address is tagged with the SID_{first} of the CI and the *ropid* so that the correct *mem-rop* reads the address. The CFU may have to stall till the *mem-rop* is completed – if the *mem-rop* is a read operation and its value is needed for further operations in the CFU.

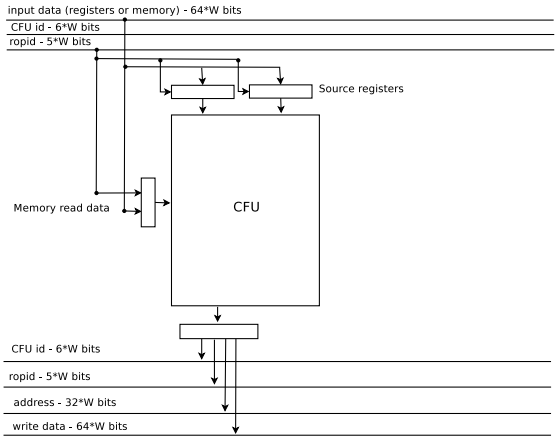

Figure 4.8: Communication links – W is the number of CFUs that can communicate in parallel

The *mem-rop*, using the address obtained from the CFU, proceeds with the memory operation in a manner identical to load/store instructions. Assuming there are no conflicts in the LSQ or TLB misses, the memory operation completes and waits in the ROB for retirement. If the *mem-rop* is a memory read operation, the read value is forwarded to the CFU over the bypass paths (again using SID_{first} to route the value to the correct CFU).The value is also tagged with the *ropid* (as described in section 4.5.4).

The CFU completes execution and forwards the results to *dreg-rops* waiting in the RS after which the *dreg-rops* wait in the ROB for retirement.

4.5.6 Communication with CFUs

Figure 4.8 shows the communication mechanism between the processor core and CFUs. As described in subsections 4.5.4 and 4.5.5, the value of *ropid* is used to pass data either from *sreg-rops* to the primary inputs of the CFU or from *mem-rops* to the CFU. The value of *ropid* is used to pass address and data values to *dreg-rops* and *mem-rops*. The parameter W determines the number of CFUs that can communicate in parallel; in our experiments W is set to 1.

The communication links and source registers for the CFUs are pipelined and operate at the core frequency. The CFU reads from the source registers at the FPGA frequency which is typically much lower. The difference between the frequency of the communication links and the CFUs allows communication to proceed efficiently even when there are several CFUs connected to the core. In our experiments, we determined that increasing the width of the communication links did not provide any performance improvements (the frequency difference between the primary reason).

4.5.7 Retire stage

The *sreg-rops* and *mem-rops* which only read from memory are retired like any other instructions – except that they do not update any registers. Unlike store instructions, write memory *mem-rops* launched by CIs are allowed to update memory only when all the *rops* launched by the CI have retired because of reasons explained in section 4.3.3. We introduce two additional bits per entry of the ROB: (1) Bit 0 is 1 if the current entry is the first *rop* of a CI. (2) Bit 1 is 1 if the current entry is the last *rop* of a CI.

Table 4.2: CI statistics – average over largest CIs per loop

Benchmark	#Nodes	#Inputs	#Outputs	#Memory operations
bzip2	14.5	6.2	4.7	5.8
libquantum	11.4	5.1	4.4	1.9
hmmer	17.5	5.5	8.4	4.3
mcf	16.3	4.2	6.2	4
gobmk	47.5	5.6	9.6	7.6
h264	37.6	12.4	17.8	4.5
sjeng	74.9	10.6	31.2	9.6

Table 4.3: Simulation processor parameters

Parameter	Value(s)
Issue-width	2/4
ROB	128/256 entries
RS	128/256 entries
Register file	256 physical entries
LSQ	64 entries
Cache	64 KB L1 I/D cache, 4MB L2 cache

Table 4.4: Normalized performance (#cycles elapsed) with non-pipelined CFUs on FPGAs

	2-issue/128 entries		2-issue/256 entries		4-issue/128 entries		4-issue/256 entries	
	baseline	CFU	baseline	CFU	baseline	CFU	baseline	CFU
bzip2	1.000	0.960	0.999	0.940	0.958	0.947	0.956	0.945
libquantum	1.000	0.847	1.000	0.810	0.653	0.607	0.653	0.606
hmmer	1.000	0.850	1.000	0.842	0.775	0.715	0.775	0.704
mcf	1.000	0.980	1.000	0.978	0.973	0.963	0.973	0.962
gobmk	1.000	1.001	0.999	0.960	0.982	0.971	0.981	0.969
h264	1.000	0.880	0.998	0.870	0.765	0.701	0.763	0.688
sjeng	1.000	0.930	0.999	0.920	0.895	0.890	0.894	0.888
Average	1.000	0.921	0.999	0.903	0.857	0.828	0.857	0.823
Improvement(%)		7.886	-	9.664	-	3.443	-	3.896

Table 4.5: Normalized performance (#cycles elapsed) with pipelined CFUs on FPGAs

	2-issue/128 entries		2-issue/256 entries		4-issue/128 entries		4-issue/256 entries	
	baseline	CFU	baseline	CFU	baseline	CFU	baseline	CFU
bzip2	1.000	0.945	0.999	0.944	0.958	0.901	0.956	0.902
libquantum	1.000	0.564	1.000	0.536	0.653	0.213	0.653	0.193
hmmer	1.000	0.714	1.000	0.701	0.775	0.483	0.775	0.472
mcf	1.000	0.964	1.000	0.963	0.973	0.937	0.973	0.936
gobmk	1.000	0.977	0.999	0.976	0.982	0.960	0.981	0.958
h264	1.000	0.693	0.998	0.700	0.765	0.465	0.763	0.447
sjeng	1.000	0.863	0.999	0.866	0.895	0.757	0.894	0.756
Average	1.000	0.817	0.999	0.812	0.857	0.674	0.857	0.666
Improvement(%)		18.298	-	18.722	-	21.401	-	22.202

Table 4.6: Normalized total energy consumption with pipelined CFUs on FPGAs

	2-issue/128 entries		2-issue/256 entries		4-issue/128 entries		4-issue/256 entries	
	baseline	CFU	baseline	CFU	baseline	CFU	baseline	CFU
bzip2	1.000	0.691	1.046	0.716	1.367	0.949	1.452	0.929
libquantum	1.000	0.620	1.058	0.735	1.044	0.726	1.141	0.700
hmmer	1.000	0.615	1.094	0.743	1.079	0.685	1.291	0.888
mcf	1.000	0.698	1.060	0.723	1.035	0.717	1.123	0.711
gobmk	1.000	0.694	1.011	0.663	1.391	0.895	1.412	0.881
h264	1.000	0.678	1.051	0.657	1.137	0.718	1.224	0.768
sjeng	1.000	0.700	1.029	0.719	1.290	0.818	1.343	0.867
Average	1.000	0.671	1.050	0.708	1.192	0.787	1.284	0.821
Improvement(%)		32.894	-	32.582	-	33.986	-	36.075

4.6 Compiler flow for creating CIs

We implement all our analysis and program transformations as part of the LLVM compiler infrastructure framework [10]. We use profiling to find the 10 most computationally intensive loops of an application and then use the pattern enumeration and selection approach described in [39] to determine the CIs – however, we are allowed to have memory access operations inside our CI. The pattern enumeration approach has to satisfy the constraints imposed in sections 4.4 and 4.5. There are additional constraints imposed to generate valid patterns for CIs. We enumerate these constraints below:

- *Convex data flow* [39]: Assume instruction X is included as part of a CI Y which depends on X is not part of the CI. Then, any instruction Z which is dependent on Y cannot be part of the CI because this would lead to a cyclic dependency between the CI and instruction Y.

- *No special instructions*: Instructions such as function calls, return, volatile memory operations, atomic instructions etc cannot be part of a CI as our synthesize logic is incapable of handling these instructions.

- *No loops within CIs*: We do not allow CIs to include the back-edge for a loop (a single CI by itself cannot execute multiple iterations). Allowing back edges would lead to the CFUs storing some context that is invisible to the processor making rollback impossible (since the CFUs do not possess any means to rollback).

- *Maximum number of memory operations per CI*: Each type of CI can launch only a predetermined maximum number of memory operations – this allows the dispatch logic to determine the number of entries to create in the ROB and RS. CIs spanning multiple basic blocks are handled as described in 4.3.5.

- *Bound on the number of memory operations*: The number of memory operations launched by a CI should be no more than the number of entries in the ROB/reservation station/LSQ.

In Table 4.2, we present some statistics for the largest CIs we found in our programs. We determined the largest CIs that our compiler pass could identify in the top 10 loops and took the average over these loops. The first column shows the number of nodes in each CI, the second shows the number of register inputs, the third shows the number of register outputs and the fourth shows the number of memory operations per CI. Allowing memory operations in the CI allows us to increase the size primarily because: (i) Fewer input/output registers. (ii) Not allowing memory operations increases the likelihood of non-convexity of a CI. Note that we cannot really use the largest CIs because of architectural constraints that limit the number of register inputs/outputs and memory operations per CI.

4.7 Results

4.7.1 Evaluation setup

We use the LLVM compiler framework to analyze and extract CIs from the SPEC integer 2006 benchmark suite (since FPGAs are not very area efficient at implementing floating point operations). Our compiler framework is limited to C programs (3 of the SPEC integer benchmarks are C++ programs) – *perlbench* and *gcc* would crash when compiled with LLVM. We use AutoESL HLS tool for synthesizing our CFUs and Xilinx XPower for energy numbers for the FPGA. We use Wisconsin's Gems simulation infrastructure [80] to model the performance of our system – the simulation parameters we use are listed in Table 4.3. We use McPAT [74] to estimate energy of the processor core – for compatibility reasons, we assume that the core is implemented in a 45 nm process (the same as Xilinx

Virtex-6). Our cores and the CFUs run at different frequencies –the core runs at 2 GHz while the frequency of the CFU is provided by Xilinx ISE. To keep the interface logic as simple as possible, the CFU is adjusted so that the CPU frequency is an integer multiple of the CFU frequency (see section 4.4). The reconfigurable fabric is reconfigured once per application – hence, the actual time per reconfiguration is not critical in our methodology. Based on our description in section 4.4, we can have most 32 CFUs at a time. For the CFUs that our compiler selected for our benchmarks, for 5 of the benchmarks the CFUs could operate at 200 MHz (1 FPGA cycle = 10 core cycles) while for the other 2 benchmarks, the frequency we had to use was 125 MHz (1 FPGA cycle = 16 core cycles).

We first examine the effect of allowing memory operations inside the CI. We find that the average number of nodes per CI is 3.6 when memory operations are not allowed inside the kernel and 12.3 when allowed. Additionally, the primary constraining factors that prevent the increase in CFU size in the first case are memory operations and limited number of register ports. For the case where memory operations are allowed, the primary constraining factor is the presence of aliased memory read operations (section 4.6).

4.7.2 Comparison with baseline

We compare the performance of our system with several core configurations in Tables 4.4 and 4.5 – All numbers are normalized with respect to the 2-issue baseline processor (by performance we mean the normalized number of cycles taken to execute the programs).

Performance: Table 4.4 shows the performance when the CFUs are not pipelined. As can be seen, the average performance improvement is small compared to software. This is primarily because of the massive frequency difference between the FPGA and CPU. Table 4.5 shows the performance when the CFUs are pipelined –

initiation interval of pipelining varies between 1 and 3 FPGA cycles (as determined by AutoPilot). With pipelining, we begin to see significant performance improvements – average of around 18% for 2-issue cores. The key point in our approach is to compare the performance of a 2-issue core augmented with CFUs (column 2) with a 4-issue core (column 5) – *our architecture can beat the performance of a 4-issue core using 2-issue core and CFUs*. For benchmarks with significant ILP (*libquantum, hmmer, sjeng, h264*), the speedup is reasonable. Benchmarks such as *mcf* which have a large working set see very little improvement mainly because they are memory bound and encounter a high fraction of cache misses.

Energy: Table 4.6 shows the energy consumption (normalized to the 2-issue core). Here, we see that having CFUs provides significant energy savings. On the average, we see a 32% energy reduction. Of the total energy savings, we observe that 41% of the total energy savings in our system comes from reduced number of accesses to the I-cache, instruction buffer and decode logic, 32% of the savings comes from reduced energy consumption of the ALUs (since many arithmetic operations are performed in the FPGA now) and register files and the remaining 27% of the energy savings are distributed among the reservation station, rename logic and ROB.

MIPS/J: To sum it up, we plot the performance over energy average values for the different configurations that we studied in Figure 4.9. We compute the MIPS/J metric for each configuration and normalize them with respect to the MIPS/J metric for the 2-issue/128 entry window core. As we can see, using our CFUs significantly improves the MIPS/J metric, upto 1.8X times the baseline core. The 4-issue cores have relatively low values of MIPS/J metric because of the large energy consumption of the 4-issue base core. However, with CFUs their MIPS/J values also rises significantly thanks to the larger reduction in energy consumption relative to the 2-issue core. An additional conclusion that can be drawn is that the increase in window size from 128 to 256 entries provides no

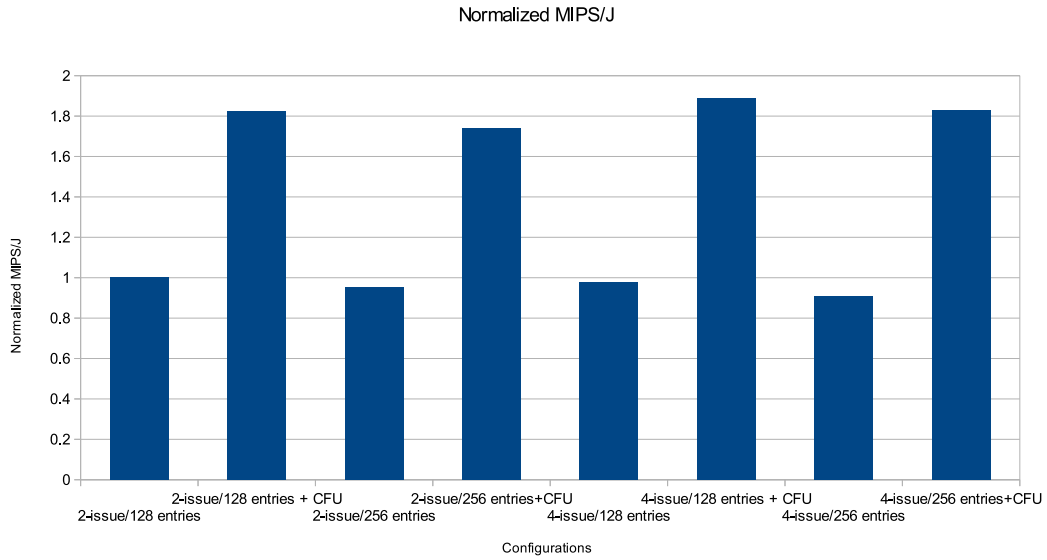

Figure 4.9: Graph showing average MIPS/J values normalized to the 2-issue/128 entry window core

significant benefits while causing an increase in energy consumption.

Resource usage: Table 4.7 shows some statistics and resource usage associated with our selected CFUs. Column 1 shows the number of different CFUs selected, column 2 shows average number of operands per CFU, column 3 shows average number of memory operations per CFU, columns 4 and 5 show the total slices and DSP blocks used respectively.

4.7.3 Comparison with restricted CIs

We examine the benefits of allowing memory operations inside a CFU. Table 4.9 shows the results of a system where the CFU is not allowed to launch memory operations compared to a 2-issue superscalar processor. As can be seen, the performance and energy improvements are very small mainly because the size of each individual CFU (and the custom datapath) is limited to a few operations and so the CFUs do not perform significantly better than software. *Not having memory*

Table 4.7: Resource usage for selected CFUs

	# CFUs	#Instructions/CI	#memops	Slices	DSP
bzip2	27	20	4	1958	114
libquantum	16	11	3	1961	112
hmmer	21	16	7	1886	139
mcf	25	18	3	1852	135
gobmk	23	11	4	1893	121
h264	22	13	7	1918	124
sjeng	16	13	4	1892	114

operations launched from within a CI limits the speedup and energy savings for superscalar processors.

We examine the benefits of allowing CIs to execute speculatively – namely the case where the memory read operation launched by a CI can proceed before preceding store instructions or CIs have completed. Table 4.10 shows the results of a system where the CFU is not allowed to execute speculatively – the CI has to wait for all preceding memory operations and CIs to complete before being launched. Again as can be seen, the performance improvement compared to a 2-issue superscalar processor is very small mainly because multiple CFUs within the same loop cannot execute speculatively – they wait for the preceding operations to complete. In a superscalar processor without CFUs, load-store instructions can execute speculatively in parallel – adding synchronization operations between software instructions and CIs significantly degrades performance (specially if the number of memory aliases is small). *If the CI needs to wait for preceding load-/store instructions to complete before execution, the improvement is significantly degraded.*

4.7.4 Sensitivity with respect to FPGA resource availability

We study the effect of increasing FPGA resources available to implement CFUs. Tables 4.11 and 4.12 show the normalized performance and energy consumption as we increase the slices available. The baseline is a 2-issue core and the 2K slices column is the same as shown in subsection 4.7.2. As can be seen the performance increase is not significant for most cases. For cases like mcf and gobmk, the primary reason is the high rate of cache misses. The observation that we made during our experiments is that pipelined CFUs implemented on FPGAs are most effective when they cover instructions which are part of hot innnermost loops. In such cases, the pipelined CFUs provide high throughput (relative to baseline). High resource availability leads to covering parts of the program which are not in the innermost loops (still in hot loops). For such CIs, the latency of execution is more important than the throughput since successive invocations of the CI are spread out over relatively large periods of time. Since the FPGA frequency is significantly lower than the core frequency, such CIs do not provide signficant performance improvements. The normalized energy consumption does decrease by 47% when 50K LUTs are available – the addition of extra CFUs reduces the instructions fetched/decoded/renamed which leads to smaller energy consumption.

4.7.5 Comparison under equal area constraint

To make a meaningful comparison under an almost equal area constraint, we compare the performance of a baseline system with two cores and a one core system with an FPGA tile for implementing the CFUs. In this way, we compare systems with equal silicon area. We run two copies of each application and examine the benefits of our approach. Table 4.8 shows the performance and energy reduction of our system. Our system does suffer from significant slowdown (40-52%) because our CFUs are not able to provide a large speedup. Also, the energy reduction

Table 4.8: Comparing the slowdown and energy reduction of 2 baseline cores with 1 core + FPGA CMP system

Benchmark	2-issue/128 entries		2-issue/256 entries		4-issue/128 entries		4-issue/256 entries	
	Slowdown (%)	Energy reduction (%)	Slowdown (%)	Energy reduction (%)	Slowdown (%)	Energy reduction (%)	Slowdown (%)	Energy reduction (%)
bzip2	76.729	23.974	76.614	24.921	75.928	25.467	76.419	30.949
libquantum	5.486	35.405	0.221	28.949	-38.949	31.466	-44.709	39.327
hmmer	33.623	33.945	31.151	28.592	16.605	33.531	14.007	29.142
mcf	80.289	23.142	80.214	24.974	80.260	23.877	80.011	29.923
gobmk	82.652	23.297	82.662	27.009	82.767	30.061	82.522	31.984
h264	29.592	28.412	31.216	33.410	13.712	34.189	9.652	34.941
sjeng	61.326	24.254	62.081	24.475	58.183	31.850	58.119	30.969
Average	52.814	27.490	52.023	27.476	41.215	30.063	39.432	32.462

(27-32%) is slightly lower than what we observe for the individual cores case (32-36%). This is because the larger execution time on our system leads to the shared L2 cache being active and dissipating leakage power for a longer time. This causes an increase in the energy consumed by our system relative to the baseline.

In the age of Dark Silicon, when energy consumed and not chip area is not the limiting constraint, it is useful to minimize the energy consumption of every application that is executed on the system. In scenarios where the system is lightly loaded with respect to utilization of all cores in the system (example: few single threaded applications are running), it may be beneficial to power down most of the cores and use CFUs implemented on the FPGA fabric to run the applications. In times of high load, the OS may decide to use all the cores in the system for obtaining the best throughput.

4.8 Conclusions

In this paper, we present an architecture by which CIs can launch memory operations and execute speculatively when integrated with a superscalar processor pipeline. Our approach does not need any synchronization or detailed memory analysis by the compiler. Our architecture uses the components of the superscalar pipeline to detect conflicts and mis-speculation at run-time and to rollback to a safe state.Our experiments show that even for pointer-heavy benchmarks our approach can average of 24% energy savings over software only implementations and significant performance improvements (average of 14%).

Table 4.9: Comparison of CFUs with 2-issue superscalar processor with no memory operations

Benchmark	Performance improvement (%)	Energy reduction(%)
bzip2	0.9	0.76
mcf	0.2	1.2
gobmk	3.7	2.9
hmmer	5.3	8.4
sjeng	6.2	10.6
libquantum	-2.1	1.4
h264ref	-4.6	6.6
Average	1.45	5.18

Table 4.10: Comparison of CFUs with 2-issue superscalar processor with synchronization

Benchmark	Performance improvement(%)	Energy reduction(%)
bzip2	-2.7	4.7
mcf	-4.1	2.2
gobmk	-9.5	4.5
hmmer	-3.5	6.2
sjeng	-2.3	0.2
libquantum	-6.1	1.1
h264ref	-7.6	2.3
Average	-5.51	2.75

Table 4.11: Sensitivity with respect to FPGA resources – normalized performance

Benchmark	baseline	2K slices	10K slices	50K slices
bzip2	1.000	0.945	0.941	0.935
libquantum	1.000	0.564	0.529	0.484
hmmer	1.000	0.714	0.692	0.663
mcf	1.000	0.964	0.961	0.958
gobmk	1.000	0.977	0.975	0.972
h264	1.000	0.693	0.669	0.639
sjeng	1.000	0.863	0.852	0.838
Average	1.000	0.817	0.803	0.784
Improvement(%)		18.298	19.725	21.580

Table 4.12: Sensitivity with respect to FPGA resources – normalized energy

Benchmark	baseline	2K slices	10K slices	50K slices
bzip2	1.000	0.691	0.611	0.541
libquantum	1.000	0.620	0.540	0.470
hmmer	1.000	0.615	0.535	0.465
mcf	1.000	0.698	0.618	0.548
gobmk	1.000	0.694	0.614	0.544
h264	1.000	0.678	0.598	0.528
sjeng	1.000	0.700	0.620	0.550
Average	1.000	0.671	0.591	0.521
Improvement(%)	-	32.894	40.894	47.894

CHAPTER 5

Architecture support for dynamic instruction set customization

5.1 Introduction

As described in the previous chapter, instruction set customization provides significant opportunities for obtaining improvements in both performance and energy consumption. Compared to coarse grained, fixed function accelerators, using custom functional units provides significantly greater flexibility. However, techniques that customize the instruction sets of processors offline are typically faced with the following challenges:

1. *Storage of configuration bits for a reconfigurable functional unit*: As the number of custom instructions (CIs) increases, the size of memory needed to store the configuration information increases.

2. *Reconfiguration overhead*: Reconfiguring a customizable functional unit may require several cycles as well as reconfiguration energy. For example, for FPGA-like fabric, the reconfiguration time could be in the order of hundreds of micro-seconds. This overhead has implications on the frequency of reconfiguration that can be performed for a given system.

3. *Binary compatibility*: Customized executables/library modules may not be able to work on systems where the customized functional units are not available. This challenge was addressed by introducing branch and link instruc-

tions in [34] with a software fall back mode. Pre-compiled libraries in which programs may spend significant amount of time provide additional challenges.

4. *Legacy binaries*: Finally, for those portions of a program for which the source code is not available (pre-compiled libraries, for example), offline instruction set customization may not be possible.

Dynamic instruction-set customization techniques could overcome some of the challenges described above. Issue (1) could be handled by generating configuration bits on the fly while (3) and (4) would be handled implicitly as no changes are made to the binary. However, dynamic techniques still face issue (2) and must also take into account the overhead (time and energy) of generating configurations for the reconfigurable functional units at runtime. The overhead depends both on the reconfigurable fabric used to implement the custom functional units as well as the heuristics used to extract and map program segments onto the custom functional units. For example, FPGAs provide an extremely fine-grained reconfigurable fabric which implies that significant effort is needed to map (place and route) data flow computations onto the fabric. Alternatives such as CGRAs (coarse-grained reconfigurable arrays) are less flexible in terms of configurability but the effort to map computation onto a CGRA may be lower.

Highlights of our contribution

1. We present a microarchitecture which can generate custom instructions on the fly for frequently executed code blocks during the application execution. Along with the custom instruction creation, the microarchitecture can generate the configurations for the custom functional units on a reconfigurable datapath.

2. Our microarchitecture generates the custom instructions and the modifies the instruction stream (with custom instructions) dynamically while ensur-

ing correct program operation, guaranteeing correct program state to deal with context switches and interrupts.

The above contributions are enabled by:

1. A hardware module to generate legal custom instructions and configurations from the instruction stream.

2. A L0 instruction cache to store the modified instruction stream (with custom instructions) along with hardware to determine the entry and exit points.

3. An architecture by which the custom instruction generation hardware can be shared among multiple cores in a chip multiprocessor (CMP) system.

We note that previous work on custom instructions has focused mainly on offline custom instruction selection/generation [21][1][13][99][36][15][119], sometimes with dynamically configurable functional units [34][57]. While other work has focused on dynamic instruction stream optimization [104][18][97][42], to the best of our knowledge, this is the first work to dynamically generate fine grained custom instructions and optimize the instruction stream in hardware. For a more detailed review of previous work, we refer the author to section 5.2.

5.2 Related work

5.2.1 Static instruction set customization

A vast body of previous work exists for customizing instruction sets for optimizing performance or energy efficiency [21][1][13][99][36][15][38][119][92][48][68][64][23][107]. However, all these approaches use a static (or offline) method to determine the operations to group into custom instructions and also statically determine the configuration of the custom functional units (CFUs) to be implemented (mostly on FPGAs). The typical flow is to identify 'hot' program segments and map the

operations in these segments to CFUs implemented on a reconfigurable fabric such as FPGAs. These approaches suffer from the limitations described in the introduction section. While FPGAs are reconfigurable, the overhead of reconfiguration is in the order of 100s of micro-seconds which prevents fine-grained reconfiguration.

5.2.2 Dynamically configurable functional units

All approaches described in this subsection determine the mapping of computations to the configurable functional units offline, i.e., either by the compiler or a trace optimizer beforehand. This implies that legacy binaries or dynamic libraries loaded at runtime cannot use the configurable functional units.

The authors in [34][35][84][51] propose a customizable datapath with multiple levels of ALUs. The datapath can be configured to connect ALUs between successive levels. Configuration is done by using a hardware block that can generate the configuration bits based on the operations selected and scheduled on the datapath. A similar datapath called versatile processing unit (VPU) with configurable interconnect between the ALUs is proposed in [28]. In the Dyser work [57], the configurable functional unit is a two dimensional array of functional units (ALUs) with configurable switches to determine the connections. A credit-based scheme is used to ensure that data transfer is scheduled correctly. The configuration of the array is selected offline, however, reconfiguration is done through a series of configuration instructions which load the configuration bits into the array. Such a functional unit resembles a CGRA system used in [84]. In QsCores [108], each core has access to a small set of custom functional units (selected offline) and the compiler partitions computations in an application among these units (where possible). In [37], the authors compose large accelerators at runtime using a set of simpler units called accelerator building blocks.

5.2.3 Dynamic instruction stream customization

While JITs (just-in-time compilers) and code instrumentation techniques can be used to generate CFUs [79], our focus is more on fine grained instruction stream customization for efficient utilization of hardware. In DISE [42], the authors identify instruction patterns of interest offline and replace them with a more optimized sequence. The patterns are stored in a pattern table and instructions executed at runtime are compared by the hardware with the patterns in the table to check whether a match occurs. If a match is detected, then a replacement sequence of instructions is inserted into the instruction stream of the processor. A similar approach was proposed in [27][26] for achieving program monitoring. In Mini-Graph [19], frequently executed dataflow patterns are identified and stored by the compiler in a hardware table before execution. At runtime, the dataflow patterns in the table are consulted to reserve functional units and bypass paths of the processor for scheduling the dataflow pattern statically. The drawback of such approaches is that the number of distinct patterns that can be stored/matched is relatively small and cannot be updated frequently.

In [121], the authors propose an approach to use the rePLay mechanism [85] to modify the instruction stream to optimize instruction segments on the fly. However, the architecture of the target configurable functional unit and the latency to map to such a functional unit is unclear.

5.2.4 Trace cache based methods

While the methods described in this subsection did not directly customize the instruction set, the goal was to optimize instruction traces at runtime for energy efficient execution. Several previous papers propose the use of an out of order (OoO) front-end to generate an instruction trace at runtime which is then fed into a simpler in-order backend [104][18] using a trace cache [97]. Repeated execution

of an optimized trace that is constructed once by the OoO engine by a simple back-end could lead to energy efficiency. Trace cache based techniques are orthogonal to our approach of customizing instruction traces as our technique can be modified to use the trace cache. The rePLay framework [85] is a popular system to generate long optimized instruction segments which commit or abort atomically. However, the use of such a framework depends on the baseline processor's microarchitecture to maintain speculative state and commit when no hazards have occurred.

5.3 System overview

Figure 5.1 shows the high level view of our processor microarchitecture with the new components shaded. In our system, an application begins execution on a processor core. Instructions are fetched and executed from the core's L1 cache. The core's pipeline is augmented with a ***Configurable Functional Unit (CFU)*** (and control logic). Our goal is to identify frequently executed parts of the program at runtime from the program binary (performed by the block indicated as ***Re-use Detection Hardware***) and schedule and map operations onto the CFU to provide speedups for frequently executed blocks of instructions. This is performed by the ***Custom Instruction Generation Hardware***. Instruction blocks are optimized by grouping instructions into custom instructions (CIs) which run on the CFU. This optimized instruction stream with CIs is written to a small ***L0 I-cache*** from which it is fetched and executed. Anytime the program's control leaves the optimized stream, the core begins fetching instructions from the L1 cache. Additional control logic is provided to detect entry and exit points in the L0 cache. Section 5.8 provides the implementation details for each module in the microarchitecture.

We enumerate the basic steps involved in achieving the described goal:

- Identifying frequently executed parts of the application to exploit CFU.

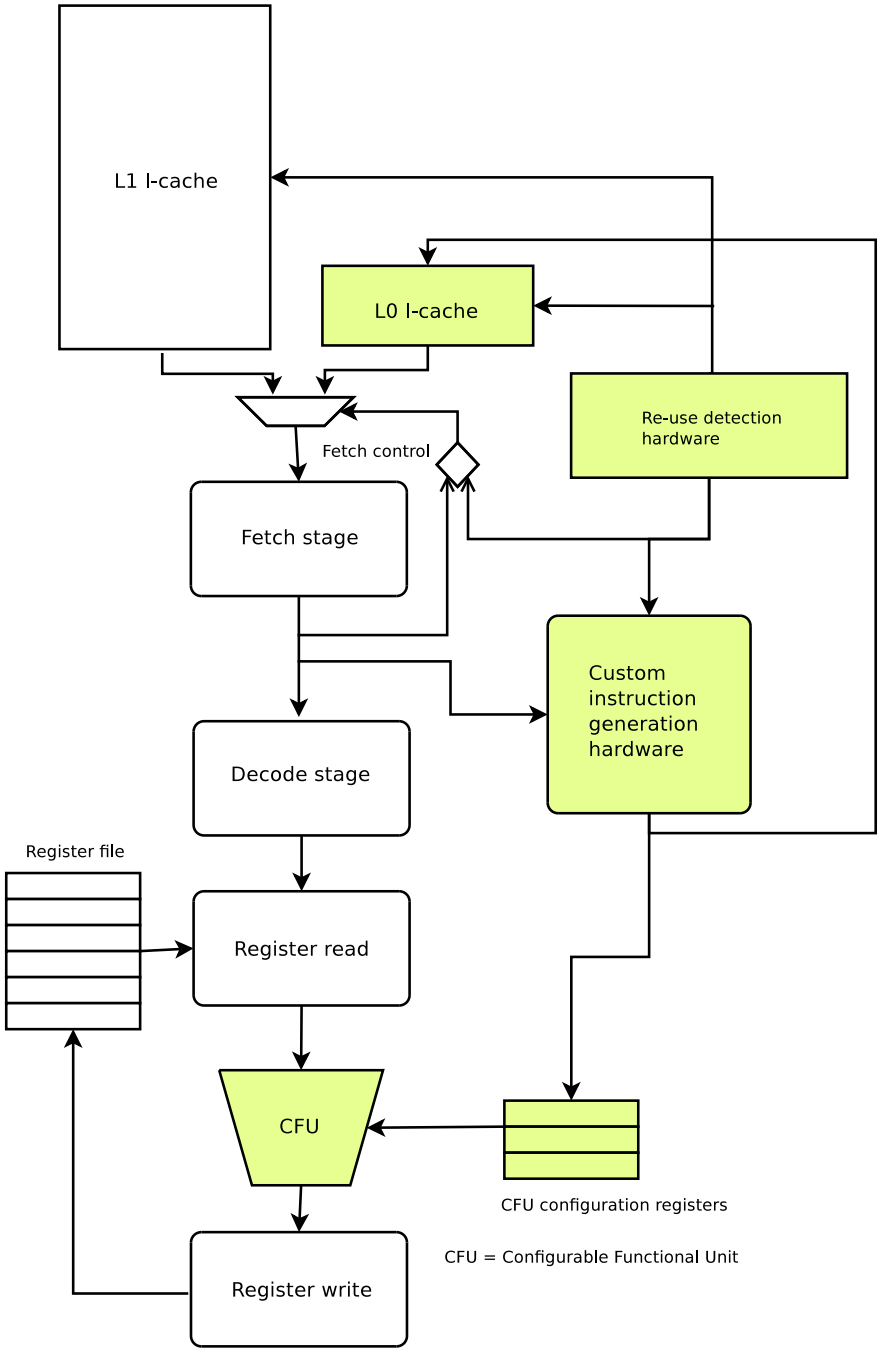

Figure 5.1: DISC microarchitecture

- Generating control information for scheduling and mapping instructions onto the CFU.

- Generating optimized and legal instruction blocks with CIs that run computations on the CFU. The system must ensure that control flow within the program is functionally correct even when CIs have been inserted into the instruction stream.

- Running optimized instruction stream on the core and CFU. The system must ensure correct execution when the instruction stream branches out of the optimized portion of the stream. Also, program state must be correctly maintained to handle context switches and interrupts. This is because after a context switch the L0 cache is invalidate and the optimized instruction stream will not be available when the application is re-scheduled.

- As a final step, we investigate the possibility of sharing the new components we add across multiple processor cores in a CMP (chip multi-processor) system. The idea is to amortize the area overhead of the components of the system. We present more details in section 5.8.3.

5.4 CFU architecture

We use the CFU design proposed in [33][34][35] for our CFU. The authors in [34] selected the design based on their analysis of applications from the SPECint and MediaBench suites. The CFU design is shown in Figure 5.2 which they call Configurable Compute Accelerator (CCA). The shaded functional units can perform logical operations, bitwise operations, sign extraction and moves on integer operands while the other functional units can perform 32-bit addition/subtraction and logical operations. The CCA has two outputs which are ultimately written to the register file and four inputs which are read from the register file. The inter-

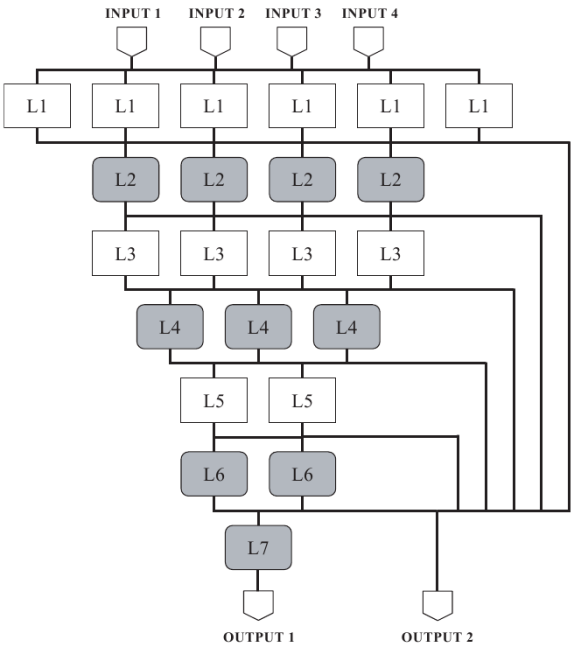

Figure 5.2: CCA from [33]

connect consists of a configurable crossbar between every two successive levels of functional units. Configuration bits are provided to configure the interconnect of the CCA. The authors compute that for the given CCA, a total of 245 configuration bits would be needed. Unlike FPGAs, where one configuration bit would be needed for every connection, in the CCA, a single bit would control the connection of a 32-bit data source with its destination port.

We introduce one modification to the above CCA – in the previous work, the authors assume that the CCA is not pipelined and end up with a delay of 5.62 ns at the 90 nm technology node. In our setup, the CCA is pipelined and hence is able to run at a higher frequency – 2 GHz at the 45 nm technology node.

```
1  L1:   add R1, R2, R3
2  L2:   xor R3, R4, R5
3        load [M1], R1
4        sub R5, R6, R7
5        and R7, R1, R2
6        store R2, [M2]
7        cmp R2, 0
8        bnz L1
9        .
10       .
11       branch L2
```

Figure 5.3: Assembly code for our running example

5.5 Identifying frequently executed parts of an application

We use a simple hardware mechanism to identify frequently executed parts of an application. We maintain a small cache of previously executed branch instructions called the *Branch History Cache (BHC)* (similar to the ones in [55][97]). If any branch is seen in the history, then very likely it is part of a loop and assume that this branch is the beginning of a loop. Even if this assumption is incorrect, it does not affect the correctness of the program. Apart from the virtual address of the branches, this cache also stores the number of instructions executed. If the number of instructions executed between two successive invocations of a branch instruction is too large, then very likely this branch is not part of an inner loop and any CIs generated for the instructions between two successive invocations of the branch would have a lesser chance of being re-used. The *BHC* uses the FIFO replacement policy.

5.6 Challenges in generating CIs and optimized instruction stream

Once a frequently executed segment of the program has been identified, the next step is to identify subgraphs of computation that can be mapped and scheduled onto the CCA. In the following subsections, we describe the challenges in the construction of legal CIs and optimized instruction streams.

For the purpose of explanation, we use the running example shown in Figure 5.3. The assumption is that the code from the first instruction to the *bnz* instruction is part of a loop while the unconditional branch to $L2$ at the end is outside the loop. Our loop detection logic described in section 5.5 would store the branch *bnz* in line 8 in the *BHC* and detect a loop when it is executed again.

5.6.1 Generating correct CIs

For a CI to be legal, it must satisfy the following constraints:

- *Number of inputs/outputs* should be no more than the number of inputs and outputs provided by the CCA (four and two respectively).
- *Restricted operations*: Only the integer operations that are supported by the CCA (see section 5.4 can be included in a subgraph. This implies that memory, branch, control transfer, or atomic operations cannot be part of an enumerated subgraph. For our example, in Figure 5.3, only the arithmetic, bitwise and comparison instructions can be executed on the CCA.
- *Localized to a single basic block*: Since the CCA cannot deal with control flow and the baseline core may not posses any logic to rollback and recover from branch mis-predictions, subgraphs that span multiple basic blocks are deemed illegal.

- *Convexity*: Only convex subgraphs [39] can be executed on the CCA. Assume instruction Y depends on instruction X. Instruction X is selected to be part of a subgraph while instruction Y is not. Then, any instruction Z which is dependent on Y cannot be part of the subgraph because this would lead to a cyclic dependency between the custom instruction representing the subgraph and instruction Y.

5.6.2 Generating correct optimized instruction stream

In our work, after constructing the CIs, we generate a distinct optimized instruction stream with the CIs that is stored in a separate L0 instruction cache.

- *Maintaining virtual addresses of instructions in the program*: Since many branch instructions in the executable specify their destination as an offset with respect to the program counter, the system should ensure that the virtual addresses of branch targets do not change. This is achieved by inserting the appropriate number of nops in the instruction stream when multiple instructions have been collapsed into one CI and inserting a branch to ensure that nops are not fetched and decoded by the processor pipeline.

  For our running example in Figure 5.3, consider a possible custom instruction created by grouping all the arithmetic, logical and comparison instructions into one subgraph (see Figure 5.4. Additional nops and a branch instruction need to be added to make sure the virtual addresses of the instructions are maintained (shaded instructions in Figure 5.4).

- *Maintaining correct program state in case of interrupts/traps/context switches*: Consider the optimized instruction stream in Figure 5.4; the custom instruction *CI* in line 1 would update the register state for all arithmetic and logical operations in the program segment. However, it is possible that the *load* instruction in line 3 causes a TLB miss and possibly invokes the operating

```
1  L1:   CI In(R1, R2, R4, R6) Out(R5, R2)
2  L2:   nop
3        load [M1], R1
4        branch L3
5        nop
6  L3:   store R2, [M2]
7        nop
8        bnz L1
9        .
10       .
11       branch L2
```

Figure 5.4: Modified assembly code for our running example with CIs

system to perform a page walk. This would invalidate all entries in the L0 cache. The program would resume from the *load* instruction, however, instructions would be fetched from normal instruction stream from the L1 cache. This implies that the *sub, and* and *cmp* instructions in lines 4, 5 and 7 respectively would be re-executed. This could potentially lead to incorrect computation.

For our example in Figure 5.4, the subgraph has instructions which are non-consecutive. More specifically, there are three sets of instructions in the subgraph which are non-consecutive – set $S_1 = \{add, xor\}$, $S_2 = \{sub, and\}$ and $S_3 = \{cmp\}$. The custom instruction executes instructions from all three sets and updates the register file. However, we observe that even though the register file is updated, re-executing instructions from S_2 and S_3 do not change the values of the registers. Such sets of instructions are called idempotent sets in [46]. Hence, even if the *load* instruction in line 3 causes a TLB fault, re-executing these instructions would not affect program

correctness.

- *Identifying correct entry/exit points into/from the optimized stream*: Consider the example in Figure 5.4 – the branch labeled *bnz* in line 8 is the back edge of a loop. The optimized stream would be stored in the L0 cache of the core and would continue execution. However, after exiting the loop the second branch from line 11 could redirect control flow in the middle of the optimized stream where the original instructions have been replaced with nops.

  The problem that occurred in the above example is that the subgraph enumeration logic considers only a limited part of the program. In the above example, the branch in line 11 was outside the loop of interest and hence, the subgraph enumeration logic could not determine all the basic block boundaries in the program. As a result the custom instruction CI spanned across multiple blocks.

  This problem is solved by our system by restricting the portions in the program from where control can enter an optimized stream. In the above example, our loop-detector from section 5.5 would mark branch *bnz* in line 8 as the branch which started the frequently executed portion and $L1$ as the start of the block which was optimized. Thus, when control flows from *bnz* to $L1$, instructions are fetched from the L0 cache. After the loop exits, subsequent instructions would be fetched from the L1 cache. The branch in line 11 is not explicitly marked as a legal entry point into the optimized stream and instructions would continue to be fetched from the L1 I-cache.

5.7 Hardware support for CI construction and optimized instruction stream generation

5.7.1 Determining instruction dependences

After the repeat execution of a loop branch is detected, the process of optimizing the instruction stream for the next iteration (second iteration of the loop) begins. All the intermediate data for processing an instruction stream is stored in a table called the *Optimized Instruction Stream Table (OIST).*The basic step in the optimizing step is to remove WAR and WAW dependences from the instruction stream so that only true dependences are maintained. The hardware for the optimizing step is similar to the rename logic in superscalar processors, except that we do not have any physical registers or free list (since we are not using the renamed instructions for actual execution). Renamed instructions are stored in the *OIST*. Note that the renaming is only used for generating the optimized instruction stream – the baseline core is still executing the instructions with possible false dependences.

If the *OIST* is full before the loop branch is detected again, then the process of storing the optimized stream is aborted. If the loop branch is detected before the *OIST* is full, then the system proceeds to the next step of subgraph enumeration.

The *OIST* contains one bit each to mark the boundaries of basic blocks. Branch instructions which can continue in the optimized stream are marked legal. For conditional branches within the optimized stream, as long as the direction of the branch is the same as that during the optimized stream construction, the processor can keep fetching instructions from the optimized stream.

5.7.2 Subgraph enumeration

The subgraph enumeration algorithm uses the optimized stream in the *OIST* to produce subgraphs and performs three passes to enumerate the subgraphs.

Matching operands: The first pass maps operands to the instruction ids in the *OIST* table. If any source instruction is in a different block or there exists a special instruction (function call, atomic) between the source and the target instruction, then the source cannot be clustered with the target and the input can only be passed through the register file. A bit is set for this source operand for such a target instruction to imply that the source instruction cannot be clustered.

In the first pass, an additional computation is performed where each instruction is assigned a completion time based on ASAP scheduling meaning that the completion time of an operand is one over the maximum completion time of its predecessors.

Maintaining data flow convexity: Data convexity is maintained by maintaining a set of successors for each instruction in the OIST and comparing it with the inputs of the current CI being produced. If a successor is an input to the CI, then the instruction cannot be clustered into the CI.

Subgraph enumeration and selection: A simple list scheduling heuristic to identify feasible subgraphs. Subgraph growth begins at an instruction (root) and moves up its predecessor instructions in a breadth-first approach. The root instruction's output is considered to be the primary output of the CCA and each predecessor instruction is scheduled at one level higher than the current instruction. Subgraphs grow by including the next predecessor instruction into the subgraph as long as the following conditions are satisfied:

- The operation can be scheduled on the higher CCA level.

- The number of inputs does not increase beyond four.
- Data flow convexity is maintained.

Our next step is to determine which instructions can be safely replaced with nops while maintaining the correctness of the program. An instruction is said to be ***internal*** to a given subgraph if it satisfies one of the following conditions

- The output register of the instruction is also an output of the subgraph OR
- The output register of the instruction is consumed only by other instructions that are *internal* to the subgraph.

Instructions that are *internal* a subgraph can be replaced with nops in the instruction stream.

Subgraph selection happens in parallel with enumeration. The latency of a subgraph is obtained by adding the maximum completion time of the predecessors (see subsection 5.7.2) with the worst case path length. If this completion time is at least half the latency that would be taken by executing all instructions within the subgraph in-order, then the subgraph is selected.

We synthesized all the hardware for subgraph enumeration and selection using the Synopsys Design Compiler with TSMC 45nm technology library. For the table structures, we use McPAT [74] to estimate the area and energy.

5.7.2.1 Storing the optimized instruction stream

The optimized instruction stream with its CIs and nops is stored in a small L0 instruction cache – in our experiments this L0 cache is a M KB 2-way set associative cache with LRU replacement. Like the L1 cache, the L0 cache is indexed and tagged using the virtual and physical addresses of the instruction blocks respectively. The L0 cache is invalidated whenever there is a context switch.

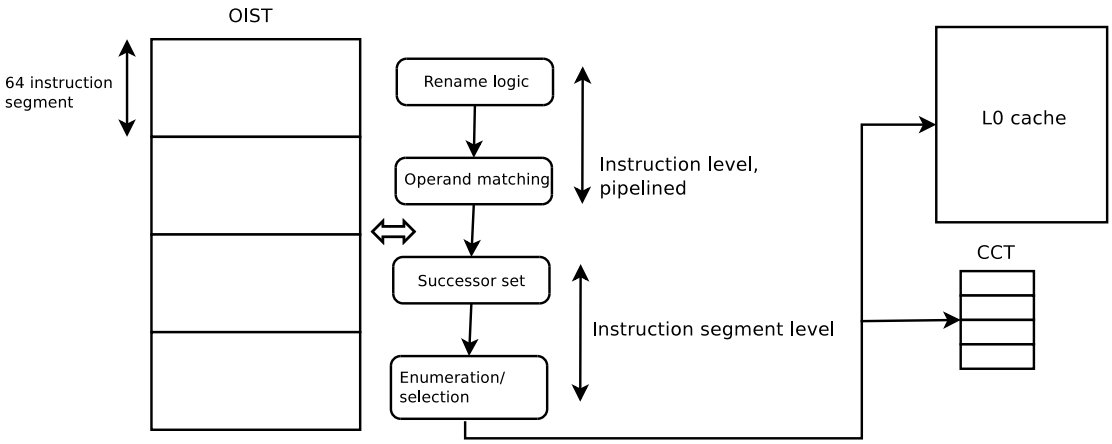

Figure 5.5: Block diagram for producing optimized instruction stream

Additionally, the configuration bits associated with the CIs are stored in a *CCA Configuration Table (CCT)*. The *CCT* is a simple SRAM table indexed using the opcode of the CI.

Figure 5.5 shows a block diagram for the whole subgraph enumeration and selection logic. Note that this logic is power gated with sleep transistors and does not dissipate power when not being used.

5.8 Putting it all together

Figure 5.6 shows the complete microarchitecture of the core's pipeline. Apart from the additional CCA unit at the execute stage, all our changes are restricted to the fetch and decode stages.

5.8.1 Fetch stage

Initially, instruction fetch starts from the L1 cache normally. PC values for branch instructions are stored in the *BHC* as well as compared with previous entries. When a loop branch is detected, the subgraph enumeration hardware is activated as execution continues in the core. As mentioned before, for our running example

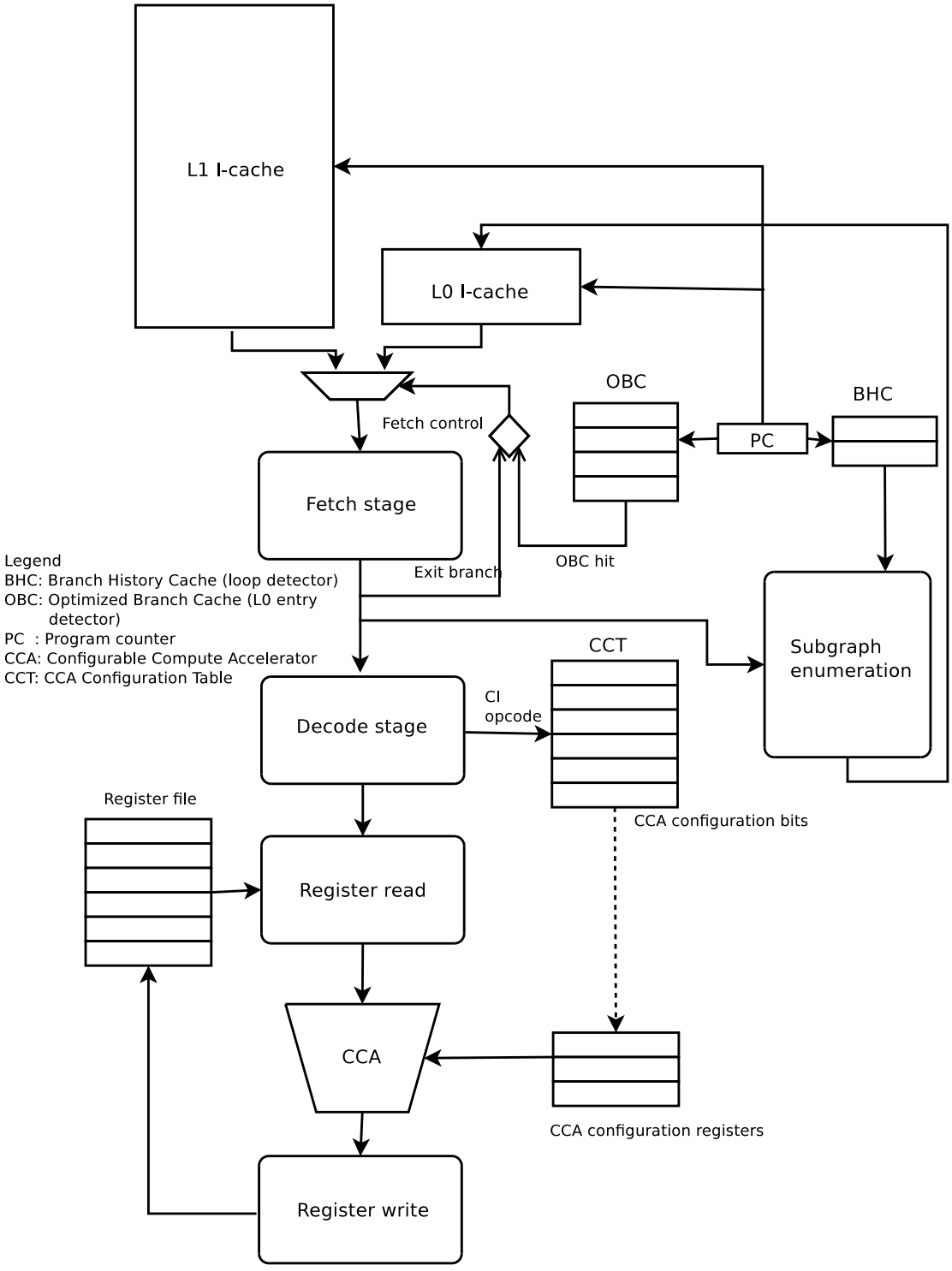

Figure 5.6: Microarchitecture of the processor pipeline

in Figure 5.3, the branch *bnz* in line 8 would be stored in the *BHC* and the loop would be detected.

When the optimized instruction stream produced and stored in the L0 cache, the loop branch is added to the *Optimized Branch Cache (OBC)* to mark a legal entry point into the optimized instruction stream. The *OBC* is a N-entry fully associative content addressable memory which stores the legal entry points into the optimized instruction stream. It is tagged using the virtual addresses (PC) of the branch instructions. The fetch stage checks the *OBC* whenever an instruction is fetched. If the *OBC* provides a hit, subsequent instructions are fetched from the L0 cache. For our running example, the branch *bnz* in line 8 would be stored in the *OBC* and would allow fetch to proceed from the L0 cache. However, the branch in line 11 is not inserted into the *OBC*.

Exit from an optimized instruction stream in the L0 cache can happen in two ways. First, a branch instruction fetched from the L0 cache can direct control flow to an address which misses in the L0 cache. Second, a branch could exit from the optimized instruction stream because of the conditions described in section 5.6.2.In both scenarios, subsequent fetches are directed towards the L1 cache.

5.8.2 Decode stage

When the decode stage encounters a CI, it uses the CI opcode to index into the *CCT* and reads the configuration bits. The configuration bits are loaded into the CCA pipeline. Register reads happen in normal fashion; however, since a CI can have upto four inputs, it may take two cycles for the read to complete due to the register port limitation.

If the baseline core is an out-of-order (OoO) core, the configuration bits are read and loaded into the CCA when the CI is scheduled for execution and not in the decode stage.

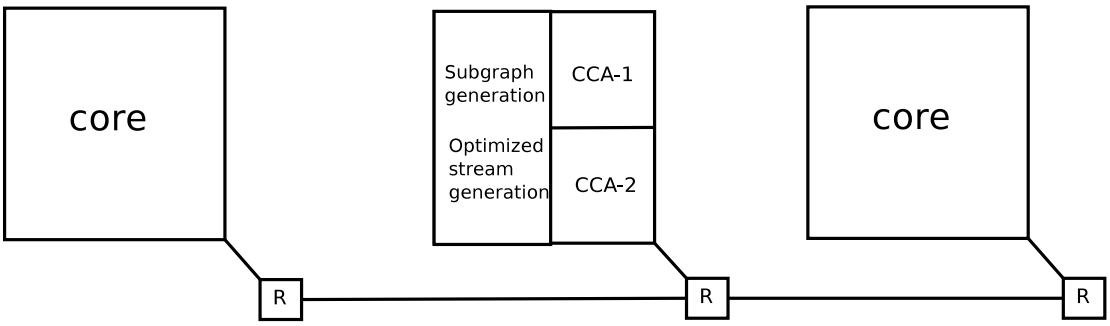

Figure 5.7: Shared CCA and subgraph enumeration hardware in a CMP system

Table 5.1: Area numbers for the different components of the dynamic enumeration logic

Component	Area (in mm^2)	Relative Area (%)
Baseline core	15.924	–
Integer CCA	0.029	0.182
FP CCA	0.052	0.327
BHC (32 entry)	0.069	0.433
OBC (16 entry)	0.062	0.389
CCT (64 entry)	0.235	1.476
L0 (8 KB)	0.707	4.440
Subgraph enumeration hardware	1.915	12.026
Total overhead	3.069	19.273

5.8.3 Extension to CMP systems

The area overhead of adding all the extra hardware described in the previous sections amounts to around 19% of the core area (see Table 5.1). Column one in table 5.1 shows the absolute area of each component (in mm^2). Column two shows the area relative to a baseline single issue core. Out of these the L0 cache and the subgraph enumeration and optimized stream generation logic take most of the area. We consider the possibility of sharing the CCA, subgraph enumeration and optimized stream generation logic among multiple cores in the system (see Figure 5.7). The effect of sharing these hardware structures among multiple cores is explained in the results section (section 5.9). Sharing reduces the area overhead for dynamically customizing instructions. The L0 cache, the *OBC* and *BHC* cannot be shared because they are accessed frequently during the fetch stage and any increase in latency of access would cause significant performance degradation.

Table 5.2: Relative energy of components normalized to baseline core energy

Component	Relative energy (%)
Integer CCA	11.53
FP CCA	9.75
BHC (32 entry)	0.023
OBC (16 entry)	1.557
CCT (64 entry)	0.478
L0 (8 KB)	21.45
Subgraph enumeration hardware	3.589
Total overhead	48.377

Table 5.2 shows the energy overhead of each component relative to the baseline single issue core. Again, the L0 cache dominates in terms of total energy consumed. We would like to clarify that the overhead mentioned in table 5.2 is not the total energy overhead with respect to the full system. For example,

Table 5.3: Simulation processor parameters

Parameter	Value(s)
Core type	Pipelined, in-order
Issue-width	1/2/4
Cache	64 KB L1 I/D cache, 4-way set associative, 4MB L2 cache

Table 5.4: Sizes of *BHC, OBC* and L0 cache

Parameter	Value(s)
BHC size	16/32/64 entries
OBC size	8/16/32 entries
CCT size	16/32/64 entries
L0 Cache	4/8/16 KB, 4-way set associative

although the L0 cache consumes significant energy, the L1 cache will see a corresponding decrease in dynamic energy because instruction fetches are directed to the L0 cache. Similarly, the ALUs in the baseline core pipeline will see an energy reduction because several computations are now executed on the CCA. We report the actual energy overhead of our approach in section 5.9.

5.9 Results

5.9.1 Evaluation framework

We use Wisconsin's Gems simulation infrastructure [80] to model the performance of our system – the simulation parameters we use are listed in Table 5.3. We use McPAT [74] to estimate energy of the processor core. Since our CCA is synthesized using TSMC 45 nm library, for compatibility reasons, we assume that the core is implemented in a 45 nm process as well. Table 5.4 lists the sizes of the *BHC*,

OBC and the L0 cache that we use in our experiments. We use the SPEC 2006 integer benchmarks for assessing the impact of the sizes of the different structures on the performance of the system and both integer and floating point benchmarks for subsequent sections.

5.9.2 Evaluating the impact of sizes of the proposed hardware structures

Table 5.5: Performance improvement (as %) with *BHC* size (# entries) relative to 16-entry *BHC* and *OBC* size with respect to 8 entry *OBC*

Benchmark	BHC sensitivity		OBC sensitivity	
	32	64	16	32
bzip2	5.781	6.263	9.132	10.343
hmmer	7.181	9.436	7.789	8.688
gobmk	7.471	8.494	4.938	6.410
sjeng	6.026	8.996	10.392	11.415
libquantum	4.006	4.313	5.345	7.323
h264	7.523	10.940	10.627	12.477
xalanc	5.808	6.804	11.707	12.998
mcf	5.208	6.725	7.191	8.880
gobmk	1.297	1.986	10.713	13.181
astar	0.896	2.426	8.352	8.876
Average	5.120	6.638	8.619	10.059

Table 5.5 shows the impact of the sizing of the *BHC* and *OBC* respectively. Increasing the size of the *BHC* increases the likelihood of detecting loops which have several basic blocks inside them. From Table 5.5 it can be seen that increasing the size of the *BHC* from 32 to 64 has minimal impact on the performance. This is because large loops with more than 32 basic blocks (and hence more than 32

branches) are unlikely to be selected for optimization since the custom instructions for large loops are unlikely to be re-used from the *CCT*.

Increasing the size of the *OBC* increases the number of legal entry points into the optimized stream located in the L0 cache. From Table 5.5, using a small number of entries (8), does impact the performance significantly. However, increasing this number beyond 16 provides no significant benefits. The temporal and spatial locality of programs (with respect to the instruction stream) ensures that an optimized program segment with limited entry points is executed frequently.

Table 5.6: Performance improvement (as %) with *CCT* size (# entries) relative to 32-entry *CCT*

Benchmark	64	128
bzip2	15.322	17.878
hmmer	18.758	24.817
gobmk	30.397	36.926
sjeng	15.765	22.149
libquantum	41.007	42.658
h264	16.978	21.799
xalanc	33.550	38.955
mcf	40.314	47.484
gobmk	18.373	24.390
astar	18.746	21.845
Average	24.921	29.890

Table 5.6 studies the impact of the size of *CCT* which determines the number of valid custom instruction (CI) configurations that are stored simultaneously. Small table sizes affect performance significantly because older CI configurations need to be invalidated frequently as and when new CIs are created. The performance improvement starts to taper off as the size is increased because the configuration

for all the CIs which are active simultaneously can fit inside the *CCT*. Since the *CCT* is an SRAM table and not a CAM (content addressable memory), we use 64 entries for all our subsequent experiments.

Table 5.7: Performance improvement (as %) with L0 size (KB) relative to 4 KB L0 cache

Benchmark	8 KB	16 KB
bzip2	21.027	26.330
hmmer	11.056	13.054
gobmk	22.606	25.086
sjeng	10.261	11.927
libquantum	21.163	24.239
h264	23.102	25.916
xalanc	32.050	35.845
mcf	13.836	16.038
gobmk	20.173	21.612
astar	11.708	13.401
Average	18.698	21.345

Finally, we study the impact of the L0 cache in table 5.7. Beyond 8 KB, we observe minimal performance improvements. 8 KB of a cache can store up to 2048 32-bit instructions which is adequate in most cases to hold inner loops of the program. For our subsequent experiments, we assume that our L0 cache is sized at 8KB.

5.9.3 Comparison with baseline and previous work

Table 5.8 and 5.9 show the performance improvement provided by dynamic instruction customization over the baseline software implementation for integer and floating point operations respectively. For completeness sake, we compare the per-

Table 5.8: Speedup with respect to baseline software implementation – specint

Benchmark	inorder	2-issue	4-issue
bzip2	1.865	1.532	1.12
hmmer	2.012	1.678	1.08
gobmk	1.775	1.498	1.102
sjeng	1.69	1.27	1.05
libquantum	2.192	1.603	1.09
h264	1.972	1.802	1.18
xalanc	1.52	1.378	1.139
mcf	1.821	1.087	1.02
omnetpp	1.796	1.796	1.07
astar	2.463	1.873	1.153
Average	1.910	1.551	1.100

formance with respect to 2-issue and 4-issue cores also. As the baseline processor complexity increases, the performance improvement drops. For example, for the 4-issue core, the performance improvement is minor. One of the reasons is that the issue width of the core is the same as the number of inputs for the CCA; hence, the customized instructions are unlikely to provide any speedup.

Table 5.10 shows the speedup with respect to the previous work which determines the CIs at compile time [34]. Our method is able to match the performance of the previous approach for all benchmarks and significantly outperform in the cases of *gobmk, omnetpp, astar* and *xalanc*. For *gobmk*, the lack of specific "hot spots" in the program ensures that the compilation pass is unable to select the best set of CIs. For the other three benchmarks, additional performance gains are obtained because these applications spend a significant amount of time in library code (stl library) and accelerating library code dynamically improves performance.

Table 5.11 and 5.12 show the impact of adding our hardware on energy con-

Table 5.9: Speedup with respect to baseline software implementation – specfp

Benchmark	inorder	2-issue	4-issue
410.bwaves	2.37	1.3	1.13
416.gamess	2.16	1.14	1.12
433.milc	1.94	1.31	1.07
434.zeusmp	2.6	1.24	1.13
435.gromacs	2.53	1.34	1.08
436.cactus	2.71	1.32	1.11
437.leslie3d	1.96	1.35	1.09
444.namd	1.93	1.18	1.12
447.dealII	2.41	1.17	1.1
450.soplex	2.49	1.14	1.06
453.povray	1.8	1.29	1.12
454.Calculix	1.8	1.13	1.13
459.Gems	2.77	1.18	1.11
465.tonto	1.89	1.13	1.07
470.lbm	1.83	1.31	1.05
481.wrf	2.29	1.28	1.12
482.sphinx3	2.01	1.33	1.11
Average	2.205	1.243	1.101

Table 5.10: Speedup (as X) over previous work [34]

Benchmark	inorder	2-issue
bzip2	1.23	1.129
hmmer	1.1	1.087
gobmk	1.34	1.23
sjeng	1.29	1.21
libquantum	1.08	1.02
h264	1.07	1.06
xalanc	1.37	1.32
mcf	1.05	1.021
omnetpp	1.67	1.52
astar	1.72	1.59
Average	1.292	1.2187

Table 5.11: Energy overhead with respect to baseline (as %) – specint

Benchmark	inorder	2-issue	4-issue
bzip2	13.87	2.3	0.72
hmmer	7.896	1.9	1.43
gobmk	18.91	2.6	1.07
sjeng	14.29	2.08	0.92
libquantum	9.05	0.18	-0.09
h264	14.59	2.36	2.36
xalanc	19.765	3.12	0.903
mcf	15.275	1.78	1.67
gobmk	21.635	1.62	2.57
astar	16.923	4.2	0.78
Average	15.2204	2.214	1.2333

Table 5.12: Energy overhead with respect to baseline (as %) – specfp

Benchmark	inorder	2-issue	4-issue
410.bwaves	14.16	1	0.09
416.gamess	13.53	1.79	0.51
433.milc	10.54	1.38	0.7
434.zeusmp	12.09	1.45	2.04
435.gromacs	11.07	1.53	0.15
436.cactusADM	10.68	1.45	0.62
437.leslie3d	8.23	1.62	0.67
444.namd	9.98	1.73	0.16
447.dealII	10.53	1.08	1.41
450.soplex	9.14	1.4	1.08
453.povray	7.05	1.14	1.08
454.Calculix	7.2	2.1	0.3
459.GemsFDTD	9.19	1.32	0.88
465.tonto	12.03	2.06	1.58
470.lbm	11.93	1.35	0.4
481.wrf	12.32	1.89	1.09
482.sphinx3	14.83	2.07	2.05
Average	10.852	1.550	0.871

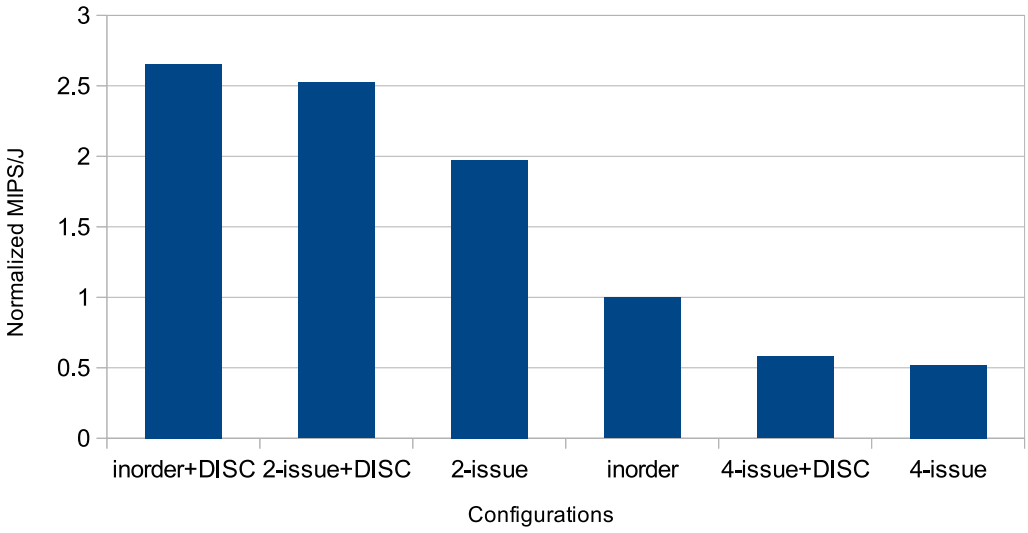

Figure 5.8: Graph showing average MIPS/J values normalized to the in-order core

sumption of the system for integer and floating point benchmarks respectively. For the simple core, the energy overhead of our system is around 15%. For more complex cores, the energy overhead is small primarily because of the higher baseline energy of the complex cores.

To sum it up, we plot the performance over energy average values for the different configurations that we studied in Figure 5.8. We compute the MIPS/J metric for each configuration and normalize them with respect to the MIPS/J metric for the in-order core. As we can see, using DISC with the in-order and 2-issue cores significantly improves the MIPS/J metric. The 4-issue cores have relatively low values of MIPS/J metric because of the large energy consumption of the 4-issue base core which is many cases provides relatively small performance improvement over the 2-issue core. The 2-issue core wins over the in-order core mainly because the core can be switched off earlier (as it completes execution earlier).

Table 5.13: Performance degradation (as %) when placing a CCA 1 or 2 hops away in the NoC – specint

Benchmark	Inorder		2-issue	
	1-hop	2-hops	1-hop	2-hops
bzip2	-19.86	-30.89	-29.73	-55.18
hmmer	-11.28	-27.46	-29.77	-59.75
gobmk	-18.02	-28.4	-29.61	-60.25
sjeng	-11.15	-23.35	-40.04	-50.87
libquantum	-14.08	-28.42	-29.49	-46.71
h264	-12.2	-28.78	-39.71	-49.63
xalanc	-12.37	-33.49	-36.31	-47.45
mcf	-12.89	-24.54	-41.6	-48.19
omnetpp	-17.07	-23.71	-37.98	-58.04
astar	-14.33	-33.98	-34.98	-47.35
Average	-14.325	-28.302	-34.922	-52.342

Table 5.14: Performance degradation (as %) when placing a CCA 1 or 2 hops away in the NoC – specfp

Benchmark	Inorder		2-issue	
	1-hop	2-hops	1-hop	2-hops
410.bwaves	-11.2	-24.54	-38.41	-49.44
416.gamess	-15.65	-30.53	-30.69	-52.92
433.milc	-12.76	-35.2	-33.17	-45.81
434.zeusmp	-17.73	-34.42	-30.64	-48.1
435.gromacs	-19.21	-24.83	-33.97	-62.35
436.cactusADM	-16.05	-29	-41.5	-62.09
437.leslie3d	-13.49	-36.38	-33	-48.59
444.namd	-11.13	-25.76	-38.44	-45.68
447.dealII	-13.77	-29.93	-32.44	-60.25
450.soplex	-11.1	-28.78	-31.87	-48.83
453.povray	-13.37	-30.92	-32.12	-63.69
454.Calculix	-14.24	-28.36	-30.94	-49.64
459.GemsFDTD	-15.63	-30.78	-31.07	-45.98
465.tonto	-19.87	-25.11	-41.14	-49.42
470.lbm	-11.76	-26.87	-34.39	-46.84
481.wrf	-15.27	-25.4	-30.69	-59.46
482.sphinx3	-17	-28.78	-36.26	-57.54
Average	-14.660	-29.152	-34.161	-52.742

5.9.4 Studying the impact of sharing hardware structures

We study the impact of sharing the *CCA* and the subgraph generation logic on performance. Before actually sharing the hardware structures, we estimate the

impact on performance of moving the structures one or two hops away on the network on chip (NoC). Tables 5.13 and 5.14 show the impact on performance if the CCA is located one and two hops away for integer and floating point benchmarks respectively. The performance degradation is very large. The reason for this is that CIs executing on the CCAs are typically small in terms of the number of operations executed. Adding a latency of four to ten cycles (which is approximately the latency to cross one and two hops in the NoC) impacts the performance greatly.

We also observe the impact of sharing the subgraph enumeration logic. This turns out to have minimal impact on performance. On the average, we observe less than 0.1% degradation when the subgraph enumeration hardware is placed two hops away. We do not enumerate the tables for these experiments as the results are practically identical to that shown while comparing against the baseline. The reason for this being that the subgraph enumeration is off the critical path of computation in a program. Our results are consistent with those observed for frame based optimizations in [85].

As noted before, the L0 cache, the *OBC* and *BHC* cannot be shared because they are accessed frequently during the fetch stage and any increase in latency of access would cause significant performance degradation.

To make a meaningful comparison under an almost equal area constraint, we compare the performance of a baseline system with three simple cores and a two core system with shared subgraph enumeration hardware and private CCAs. The two core system with DISC consumes slightly less area (see table 5.1). We run three copies of each application and examine the benefits of our approach. Tables 5.15 and 5.16 show the performance and energy reduction of our system for integer and floating-point benchmarks respectively. The performance of our two core system is comparable to the three core baseline, however, our system benefits from a 30% reduction in energy.

Table 5.15: Comparing the slowdown and energy reduction of 3 baseline cores with 2 core DISC (specint)

Benchmark	Inorder		2-issue	
	Slowdown (%)	Energy reduction (%)	Slowdown (%)	Energy reduction (%)
bzip2	-0.29	24.05	6.13	38.31
hmmer	1.19	32.54	5.91	31.47
gobmk	0.19	25.7	9.28	39.83
sjeng	2.07	37.02	8.67	39.3
libquantum	2.82	30.36	5.49	34.77
h264	1.63	35.5	-1.25	43.69
xalanc	-3.12	37.42	6.67	34.1
mcf	1.54	28.32	7.92	29.76
omnetpp	-0.11	31.8	4.25	45.73
astar	-1.44	26.91	-0.77	41.05
Average	0.448	30.962	5.23	37.801

Table 5.16: Comparing the slowdown and energy reduction of 3 baseline cores with 2 core DISC (specfp)

Benchmark	Inorder		2-issue	
	Slowdown (%)	Energy reduction (%)	Slowdown (%)	Energy reduction (%)
410.bwaves	3.06	34.61	7.77	30.57
416.gamess	-0.64	24.8	-1.48	40.98
433.milc	0.45	26.17	8.73	45.63
434.zeusmp	2.6	24.82	-1.63	41.16
435.gromacs	-1.38	37.46	8.72	35.09
436.cactu	-1.32	28.54	2.92	41.43
437.leslie3d	0.9	31.4	3.57	32.66
444.namd	4.44	24.19	7.67	42.32
447.dealII	3.54	33.54	7.62	33.06
450.soplex	3.28	36.17	8.42	34.74
453.povray	-2.57	35.68	1.45	43.62
454.Calculix	4.51	28.24	4.53	33.52
459.Gems	-0.66	30.79	4.62	39.56
465.tonto	3.85	23.63	1.89	36.77
470.lbm	1.4	35.31	5.19	38.99
481.wrf	3.94	30.31	2.1	42.25
482.sphinx3	-1.71	26.36	-2.09	30.4
Average	1.393	30.118	4.117	37.808

Table 5.17: Comparing the slowdown and energy reduction of 3 baseline cores with 2 core DISC (PARSEC)

Benchmark	Inorder		2-issue	
	Slowdown (%)	Energy reduction (%)	Slowdown (%)	Energy reduction (%)
blackscholes	1.91	37.30	8.52	41.63
bodytrack	0.52	23.60	2.79	37.72
facesim	1.44	29.46	4.13	35.17
ferret	21.45	7.56	27.94	5.34
freqmine	0.59	31.84	2.87	29.63
swaptions	11.34	16.43	15.34	19.61
fluidanimate	4.56	28.45	8.91	45.65
x264	42.30	2.10	51.45	5.98
canneal	29.33	-1.23	32.45	0.56
streamcluster	2.05	35.86	6.70	34.26
Average	11.55	21.14	16.11	25.56

Table 5.17 shows the results for the multi-threaded benchmarks from the PARSEC suite [16]. For most of the benchmarks, the behavior is similar to the multi-workload scenario. The reason for this is that most of the PARSEC benchmarks are data parallel in nature and hence, once the independent threads are launched, the behavior (in terms of computation) of individual threads is not very different from that of the SPEC benchmarks. There are three applications which suffer significant slowdown – *canneal, x264* and *ferret*. *x264* and *ferret* exploit pipelined parallelism and hence, having an extra core helps improve the throughput of the pipeline by balancing the resources available for the three threads. *Canneal* is a memory intensive benchmark and hence, our CCA does not provide adequate benefits over the software implementation. In the age of Dark Silicon, we may have DISC along with all three processor cores; however, it may be decided (by the OS scheduler, for example) to power down one of the cores when DISC can provide a good performance boost or power down DISC hardware when no performance boost is obtained.

5.10 Conclusions

In this chapter, we provided a detailed description of an architecture with the capability to customize instructions on the fly and execute them on a pre-designed CCA. Results show that the high frequency of the CCA combined with quick reconfiguration time and high re-use of optimized instruction streams lead to significant performance improvements with respect to the baseline. The fact that instruction customization happens dynamically allows us to obtain speedups for unmodified, pre-compiled binaries. Overall, we see around 1.9X speedup over the baseline when each core has a private copy of the hardware structures proposed and around 31% energy reduction when certain hardware structures used for customizing the instruction set are shared.

CHAPTER 6

Conclusions and Future Directions

Reducing energy consumption is a crucial problem for many computing systems. A wide variety of tools and techniques are needed to tackle the challenges posed by this problem. Designers must consider the best optimization strategies and decide whether to implement them offline or online. In this thesis, I studied three main aspects of energy efficient computing, namely application-level variability, application-level reliability and flexibility and explored the design and trade-off associated with offline and online optimizations.

- *Variation aware DVFS scheduling*: While DVFS based scheduling is an old problem, I explored scheduling optimizations for applications in the presence of input dependent variation. A rigorous convex optimization based schedule was generated offline and a simple lookup table based approach was used at runtime.

- *Identifying critical instructions*: Soft errors and single-event upsets are estimated to be increasingly an issue as device sizes decrease [4]. A compiler analysis technique along with a runtime monitoring technique is presented to reduce the number of instructions that need to be replicated and verified.

- *Improved ASIPs*: The use of customized computing hardware has been very effective at improving energy efficiency. Techniques and architectures to extend the impact of ASIPs to SPEC benchmark like workloads were presented. The techniques involved both compiler analysis as well as runtime

scheduling and hazard detection to obtain best results.

There are still lots of avenues for further improvement in each of the three areas. While we explored the concept of critical instructions in error resilient applications, further research is needed to develop architectures which can effectively exploit knowledge of such critical instruction segments. Additional analysis techniques which can determine the criticality of an instruction/instruction segment at runtime would provide further benefits.

For ASIPs, the interesting direction is to study the nature of the reconfigurable fabric used to implement the custom functional units – should it be FPGA or CGRA based? Additionally, future ASIPs could incorporate mechanisms from out of order execution such as speculative execution to obtain higher efficiency. Studying the tradeoff between the overhead of such mechanisms and the added benefits they provide in an ASIP environment could lead to interesting architectures.

To conclude, my thesis studies three areas of energy efficient computing and provides solutions which involve both offline and online optimizations. We believe that these studies form a useful stepping stone for future refinements and exploration.

References

[1] Altera NIOS-II processor. http://www.altera.com/devices/processor/nios2/ni2-index.html.

[2] Arm mpcore. http://www.arm.com/products/processors/classic/arm11/arm11-mpcore.php.

[3] Intel 64 and IA-32 architectures optimization reference manual. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.

[4] ITRS report, 2011. http://www.itrs.net/Links/2011ITRS/Home2011.htm.

[5] LDPC benchmark. http://www.cs.utoronto.ca/~radford/ldpc.software.html.

[6] MediaBench2 benchmark suite. http://euler.slu.edu/~fritts/mediabench/.

[7] MiBench benchmark suite. http://www.eecs.umich.edu/mibench/.

[8] Mips32 1074k. http://www.mips.com/products/cores/32-64-bit-cores/mips32-1074k/.

[9] Sesc simulator. http://iacoma.cs.uiuc.edu/~paulsack/sescdoc/.

[10] The LLVM compilation infrastructure. http://llvm.org.

[11] The Omega project. http://www.cs.umd.edu/projects/omega/.

[12] The SPARC architecture manual, version 9. http://developers.sun.com/solaris/articles/sparcv9.pdf.

[13] Xtensa customizable processor. http://www.tensilica.com/products/xtensa-customizable.

[14] Alexandru Andrei, Marcus T. Schmitz, Petru Eles, Zebo Peng, and Bashir M. Al Hashimi. Quasi-static voltage scaling for energy minimization with time constraints. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '05, pages 514–519, Washington, DC, USA, 2005. IEEE Computer Society.

[15] K. Atasu, C. Ozturan, G. Dundar, O. Mencer, and W. Luk. Chips: Custom hardware instruction processor synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(3):528 –541, march 2008.

[16] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[17] Partha Biswas, Nikil D. Dutt, Laura Pozzi, and Paolo Ienne. Introduction of architecturally visible storage in instruction set extensions. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(3):435 –446, march 2007.

[18] B. Black, B. Rychlik, and J.P. Shen. The block-based trace cache. In *Computer Architecture, 1999. Proceedings of the 26th International Symposium on*, pages 196–207, 1999.

[19] A. Bracy, P. Prahlad, and A. Roth. Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 18–29, 2004.

[20] M.A. Breuer. Multi-media applications and imprecise computation. In *Digital System Design, 2005. Proceedings. 8th Euromicro Conference on*, pages 2 – 7, aug.-3 sept. 2005.

[21] P. Brisk, A. Kaplan, and M. Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In *Design Automation Conference, 2004. Proceedings. 41st*, pages 395 –400, july 2004.

[22] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 83 –94, june 2000.

[23] Jorge E. Carrillo and Paul Chow. The effect of reconfigurable units in superscalar processors. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, FPGA '01, pages 141–150, New York, NY, USA, 2001. ACM.

[24] Jian-Jia Chen, Heng-Ruey Hsu, Kai-Hsiang Chuang, Chia-Lin Yang, Ai-Chun Pang, and Tei-Wei Kuo. Multiprocessor energy-efficient scheduling with task migration considerations. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 101 – 108, june-2 july 2004.

[25] Jian-Jia Chen and Chin-Fu Kuo. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (dvs) platforms. In *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pages 28 –38, aug. 2007.

[26] S. Chen, M. Kozuch, P.B. Gibbons, M. Ryan, T. Strigkos, T.C. Mowry, O. Ruwase, E. Vlachos, B. Falsafi, and V. Ramachandran. Flexible hardware acceleration for instruction-grain lifeguards. *Micro, IEEE*, 29(1):62–72, 2009.

[27] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 377–388, Washington, DC, USA, 2008. IEEE Computer Society.

[28] A.C. Cheng. Amplifying embedded system efficiency via automatic instruction fusion on a post-manufacturing reconfigurable architecture platform. In *Quality Electronic Design, 2008. ISQED 2008. 9th International Symposium on*, pages 744–749, 2008.

[29] Kihwan Choi, R. Soma, and M. Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(1):18 – 28, jan. 2005.

[30] P. Chowdhury and C. Chakrabarti. Static task-scheduling algorithms for battery-powered dvs systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 13(2):226 –237, feb. 2005.

[31] N. Clark, A. Hormati, and S. Mahlke. Veal: Virtualized execution accelerator for loops. In *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, pages 389 –400, june 2008.

[32] N. Clark, M. Kudlur, Hyunchul Park, S. Mahlke, and K. Flautner. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 30 – 40, dec. 2004.

[33] Nathan Clark. *Customizing the Computation Capabilities of Microprocessors*. PhD thesis, University of Michigan, 2007.

[34] Nathan Clark, Jason Blome, Michael Chu, Scott Mahlke, Stuart Biles, and Krisztian Flautner. An architecture framework for transparent instruction set customization in embedded processors. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 272–283, Washington, DC, USA, 2005. IEEE Computer Society.

[35] Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztian Flautner. Application-specific processing on a general-purpose core via

transparent instruction set customization. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 30–40, Washington, DC, USA, 2004. IEEE Computer Society.

[36] Jason Cong, Yiping Fan, Guoling Han, and Zhiru Zhang. Application-specific instruction generation for configurable processor architectures. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, FPGA '04, pages 183–189, New York, NY, USA, 2004. ACM.

[37] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. Charm: a composable heterogeneous accelerator-rich microprocessor. In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, ISLPED '12, pages 379–384, New York, NY, USA, 2012. ACM.

[38] Jason Cong, Guoling Han, and Zhiru Zhang. Architecture and compiler optimizations for data bandwidth improvement in configurable processors. *IEEE Trans. Very Large Scale Integr. Syst.*, 14:986–997, September 2006.

[39] Jason Cong, Hui Huang, and Wei Jiang. A generalized control-flow-aware pattern recognition algorithm for behavioral synthesis. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 1255 –1260, march 2010.

[40] Jason Cong, Glenn Reinman, Alex Bui, and Vivek Sarkar. Customizable domain-specific computing. *Design Test of Computers, IEEE*, 28(2):6 –15, march-april 2011.

[41] George A. Constantinides, Peter Y. K. Cheung, and Wayne Luk. *Synthesis and optimization of DSP algorithms*. Kluwer Academic Publishers, 2004.

[42] Marc L. Corliss, E. Christopher Lewis, and Amir Roth. Dise: a programmable macro engine for customizing applications. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 362–373, New York, NY, USA, 2003. ACM.

[43] F. Dabiri, R. Jafari, A. Nahapetian, and M. Sarrafzadeh. A unified optimal voltage selection methodology for low-power systems. In *Quality Electronic Design, 2007. ISQED '07. 8th International Symposium on*, pages 210 –218, march 2007.

[44] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: path-sensitive program verification in polynomial time. *SIGPLAN Not.*, 37:57–68, May 2002.

[45] Abhijit Davare, Jike Chong, Qi Zhu, Douglas Densmore, and Alberto Sangiovanni-Vincentelli. Classification, customization, and characterization: Using milp for task allocation and scheduling. Technical Report UCB/EECS-2006-166, University of California, Berkeley, December 2006.

[46] M. A. de Kruijf et al. Static analysis and compiler design for idempotent processing. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 475–486, New York, NY, USA, 2012. ACM.

[47] R.P. Dick, D.L. Rhodes, and W. Wolf. Tgff: task graphs for free. In *Hardware/Software Codesign, 1998. (CODES/CASHE '98) Proceedings of the Sixth International Workshop on*, pages 97 –101, mar 1998.

[48] Quang Dinh, Deming Chen, and Martin D. F. Wong. Efficient asip design for configurable processors with fine-grained resource sharing. In *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, FPGA '08, pages 99–106, New York, NY, USA, 2008. ACM.

[49] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35:57–72, October 2001.

[50] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[51] S. Gupta et al. Bundled execution of recurring traces for energy-efficient general purpose processing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 12–23, New York, NY, USA, 2011. ACM.

[52] David Evans, John Guttag, James Horning, and Yang Meng Tan. Lclint: a tool for using specifications to check code. *SIGSOFT Softw. Eng. Notes*, 19:87–96, December 1994.

[53] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 385–396, New York, NY, USA, 2010. ACM.

[54] Krisztin Flautner and et al. Automatic performance setting for dynamic voltage scaling. In *IN MOBILE COMPUTING AND NETWORKING*, pages 260–271. ACM Press, 2001.

[55] Alejandro García, Oliverio J. Santana, Enrique Fernández, Pedro Medina, and Mateo Valero. Lpa: a first approach to the loop processor architecture. In *Proceedings of the 3rd international conference on High performance embedded architectures and compilers*, HiPEAC'08, pages 273–287, Berlin, Heidelberg, 2008. Springer-Verlag.

[56] Soheil Ghiasi, Elaheh Bozorgzadeh, Siddharth Choudhuri, and Majid Sarrafzadeh. A unified theory of timing budget management. In *In IEEE/ACM International Conference on Computer-Aided Design*, pages 653–659, 2004.

[57] V. Govindaraju, Chen-Han Ho, and K. Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 503–514, 2011.

[58] R. L. Graham and R. L. Grahamt. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.

[59] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G.Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. In E.L. Johnson P.L. Hammer and B.H. Korte, editors, *Discrete Optimization II Proceedings of the Advanced Research Institute on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symposium co-sponsored by IBM Canada and SIAM Banff, Aha. and Vancouver*, volume 5 of *Annals of Discrete Mathematics*, pages 287 – 326. Elsevier, 1979.

[60] F. Gruian and K. Kuchcinski. Lenes: task scheduling for low-energy systems using variable supply voltage processors. In *Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific*, pages 449 –455, 2001.

[61] F. Gruian and K. Kuchcinski. Uncertainty-based scheduling: energy-efficient ordering for tasks with variable execution time [processor scheduling]. In *Low Power Electronics and Design, 2003. ISLPED '03. Proceedings of the 2003 International Symposium on*, pages 465 – 468, aug. 2003.

[62] Flavius Gruian. Hard real-time scheduling for low-energy using stochastic data and dvs processors. In *Proceedings of the 2001 international symposium on Low power electronics and design*, ISLPED '01, pages 46–51, New York, NY, USA, 2001. ACM.

[63] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 291–301, New York, NY, USA, 2002. ACM.

[64] J.R. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, pages 12 –21, apr 1997.

[65] M. Hiller. Executable assertions for detecting data errors in embedded control systems. In *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, pages 24 –33, 2000.

[66] Tohru Ishihara and Hiroto Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the 1998 international symposium on Low power electronics and design*, ISLPED '98, pages 197–202, New York, NY, USA, 1998. ACM.

[67] Wei Jiang, Zhiru Zhang, M. Potkonjak, and J. Cong. Scheduling with integer time budgeting for low-power optimization. In *Design Automation Conference, 2008. ASPDAC 2008. Asia and South Pacific*, pages 22 –27, march 2008.

[68] Kingshuk Karuri, Anupam Chattopadhyay, Manuel Hohenauer, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Increasing data-bandwidth to instruction-set extensions through register clustering. In *Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, ICCAD '07, pages 166–171, Piscataway, NJ, USA, 2007. IEEE Press.

[69] R.E. Kessler. The Alpha 21264 microprocessor. *Micro, IEEE*, 19(2):24 –36, mar/apr 1999.

[70] Theo Kluter, Philip Brisk, Paolo Ienne, and Edoardo Charbon. Speculative DMA for architecturally visible storage in instruction set extensions. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, CODES+ISSS '08, pages 243–248, New York, NY, USA, 2008. ACM.

[71] Theo Kluter, Samuel Burri, Philip Brisk, Edoardo Charbon, and Paolo Ienne. Virtual ways: Efficient coherence for architecturally visible storage in automatic instruction set extensions. In *HiPEAC*, volume 5952 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 2010.

[72] Lap-Fai Leung, Chi-Ying Tsui, and Xiaobo Sharon Hu. Exploiting dynamic workload variation in low energy preemptive task scheduling. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '05, pages 634–639, Washington, DC, USA, 2005. IEEE Computer Society.

[73] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Understanding the propaga-

tion of hard errors to software and implications for resilient system design. *SIGOPS Oper. Syst. Rev.*, 42:265–276, March 2008.

[74] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 469–480, New York, NY, USA, 2009. ACM.

[75] Xiaodong Li, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. Scaling of architecture level soft error rate for superscalar processors. 2005.

[76] Xuanhua Li and Donald Yeung. Application-level correctness and its impact on fault tolerance. In *In Proceedings of the 13th International Symposium on High Performance Computer Architecture*, pages 181–192, 2007.

[77] Ming Liu, W. Kuehn, Zhonghai Lu, and A. Jantsch. Run-time partial reconfiguration speed investigation and architectural design space exploration. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 498 –502, 31 2009-sept. 2 2009.

[78] Michele Lombardi and Michela Milano. Stochastic allocation and scheduling for conditional task graphs in mpsocs. In Frdric Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006*, volume 4204 of *Lecture Notes in Computer Science*, pages 299–313. Springer Berlin / Heidelberg, 2006.

[79] Ya-shuai Lü, Li Shen, Zhi-ying Wang, and Nong Xiao. Dynamically utilizing computation accelerators for extensible processors in a software approach. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '09, pages 51–60, New York, NY, USA, 2009. ACM.

[80] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33:92–99, November 2005.

[81] T. Mudge. Power: a first-class architectural design constraint. *Computer*, 34(4):52 –58, apr 2001.

[82] Naveen Muralimanohar and Rajeev Balasubramonian. Cacti 6.0: A tool to model large caches. Technical report, HP Laboratories, 2009.

[83] N. Oh, P.P. Shirvani, and E.J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *Reliability, IEEE Transactions on*, 51(1):63 –75, mar 2002.

[84] Yongjun Park, Hyunchul Park, and Scott Mahlke. Cgra express: accelerating execution using dynamic operation fusion. In *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '09, pages 271–280, New York, NY, USA, 2009. ACM.

[85] Sanjay J. Patel and Steven S. Lumetta. replay: A hardware framework for dynamic optimization. *IEEE Trans. Comput.*, 50(6):590–608, June 2001.

[86] K. Pattabiraman, Z. Kalbarczyk, and R.K. Iyer. Automated derivation of application-aware error detectors using static analysis. In *On-Line Testing Symposium, 2007. IOLTS 07. 13th IEEE International*, pages 211 –216, july 2007.

[87] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer. Symplfied: Symbolic program-level fault injection and error detection framework. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 472 –481, june 2008.

[88] K. Pattabiraman, G. P. Saggese, D. Chen, Z. Kalbarczyk, and R. K. Iyer. Dynamic derivation of application-specific error detectors and their implementation in hardware. In *Proceedings of the Sixth European Dependable Computing Conference*, pages 97–108, Washington, DC, USA, 2006. IEEE Computer Society.

[89] P.G. Paulin and J.P. Knight. Force-directed scheduling for the behavioral synthesis of asics. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 8(6):661 –679, jun 1989.

[90] Peter K. Pearson. Fast hashing of variable-length text strings. *Commun. ACM*, 33:677–680, June 1990.

[91] J. Pouwelse, K. Langendoen, and H.J. Sips. Application-directed voltage scaling. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 11(5):812 –826, oct. 2003.

[92] Laura Pozzi and Paolo Ienne. Exploiting pipelining to relax register-file port constraints of instruction-set extensions. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '05, pages 2–10, New York, NY, USA, 2005. ACM.

[93] Meikang Qiu, Chun Xue, Zili Shao, and E.H.M. Sha. Energy minimization with soft real-time and dvs for uniprocessor and multiprocessor embedded systems. In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pages 1 –6, april 2007.

[94] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendu S. Mukherjee. Software-controlled fault tolerance. *ACM Trans. Archit. Code Optim.*, 2:366–396, December 2005.

[95] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[96] H.P. Rosinger. Connecting customized ip to the microblaze soft processor using the fast simplex link (fsl) channel. `www.xilinx.com/support/documentation/application_notes/xapp529.pdf`.

[97] E. Rotenberg, S. Bennett, and J.E. Smith. A trace cache microarchitecture and evaluation. *Computers, IEEE Transactions on*, 48(2):111–120, 1999.

[98] S.K. Sahoo, Man-Lap Li, P. Ramachandran, S.V. Adve, V.S. Adve, and Yuanyuan Zhou. Using likely program invariants to detect hardware errors. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 70 –79, june 2008.

[99] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 422–433, New York, NY, USA, 2003. ACM.

[100] P. Schumacher and W. Chung. Fpga-based mpeg-4 codec. In *DSP Magazine*, pages 8–9, 2005.

[101] Dongkun Shin and Jihong Kim. Power-aware scheduling of conditional task graphs in real-time multiprocessor systems. In *Low Power Electronics and Design, 2003. ISLPED '03. Proceedings of the 2003 International Symposium on*, pages 408 – 413, aug. 2003.

[102] David B. Shmoys and Éva Tardos. An approximation algorithm for the generalized assignment problem. *Math. Program.*, 62:461–474, December 1993.

[103] Ayswarya Sundaram, Ameen Aakel, Derek Lockhart, Darshan Thaker, and Diana Franklin. Efficient fault tolerance in multi-media applications through selective instruction replication. In *Proceedings of the 2008 workshop on Radiation effects and fault tolerance in nanometer technologies*, WREFT '08, pages 339–346, New York, NY, USA, 2008. ACM.

[104] E. Talpes and D. Marculescu. Execution cache-based microarchitecture for power-efficient superscalar processors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 13(1):14–26, 2005.

[105] Darshan D. Thaker, Diana Franklin, John Oliver, Susmit Biswas, Derek Lockhart, Tzvetan Metodi, and Frederic T. Chong. Characterization of errortolerant applications when protecting control data. In *In Proc. of the IEEE Intl Symp. on Workload Characterization*, 2006.

[106] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 49–59, Washington, DC, USA, 2007. IEEE Computer Society.

[107] F. Vahid, G. Stitt, and R. Lysecky. Warp processing: Dynamic translation of binaries to fpga circuits. *Computer*, 41(7):40–46, 2008.

[108] Ganesh Venkatesh, Jack Sampson, Nathan Goulding-Hotta, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. Qscores: trading dark silicon for scalable energy efficiency with quasi-specific cores. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 163–174, New York, NY, USA, 2011. ACM.

[109] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica*, 48:37–66, March 2007.

[110] S. Verma and A. S. Dabare. Understanding clock domain crossing issues. *EETimes EDA DesignLine*, December 2007.

[111] Hangsheng Wang, Li-Shiuan Peh, and S. Malik. A technology-aware and energy-oriented topology exploration for on-chip networks. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 1238 – 1243 Vol. 2, march 2005.

[112] Nicholas J. Wang, Justin Quek, Todd M. Rafacz, and Sanjay J. patel. Characterizing the effects of transient faults on a high-performance processor

pipeline. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN '04, pages 61–, Washington, DC, USA, 2004. IEEE Computer Society.

[113] N.J. Wang and S.J. Patel. Restore: Symptom-based soft error detection in microprocessors. *Dependable and Secure Computing, IEEE Transactions on*, 3(3):188 –201, july-sept. 2006.

[114] Tao Wang, Zhihong Yu, Yuan Liu, Dong Liu, and Joel Emer. An architecture & mechanism for supporting speculative execution of a context-full reconfigurable function unit. In *Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL 2010)*, 2010.

[115] Dong Wu, Bashir M. Al-Hashimi, and Petru Eles. Scheduling and mapping of conditional task graphs for the synthesis of low power embedded systems. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, pages 10090–, Washington, DC, USA, 2003. IEEE Computer Society.

[116] Changjiu Xian, Yung-Hsiang Lu, and Zhiyuan Li. Energy-aware scheduling for real-time multiprocessor systems with uncertain task execution time. In *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pages 664 –669, june 2007.

[117] Ruibin Xu, Rami Melhem, and Daniel Mossé. A unified practical approach to stochastic dvs scheduling. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 37–46, New York, NY, USA, 2007. ACM.

[118] Ruibin Xu, Daniel Mossé, and Rami Melhem. Minimizing expected energy in real-time embedded systems. In *Proceedings of the 5th ACM international conference on Embedded software*, EMSOFT '05, pages 251–254, New York, NY, USA, 2005. ACM.

[119] Z.A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 225 –235, june 2000.

[120] Thomas Y. Yeh, Glenn Reinman, Sanjay J. Patel, and Petros Faloutsos. Fool me twice: Exploring and exploiting error tolerance in physics-based animation. *ACM Trans. Graph.*, 29:5:1–5:11, December 2009.

[121] S. Yehia and O. Temam. From sequences of dependent instructions to functions: an approach for improving performance without ilp or speculation. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 238–249, 2004.

[122] Dakai Zhu, Rami Melhem, and Bruce Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. In *IEEE Trans. on Parallel and Distributed Systems*, pages 84–94, 2001.