SHARP:

Sustainable Hardware Acceleration for Rapidly-evolving Pre-existing systems

by

Julie Beeston B.Sc., University of Victoria, 1992 M.Sc., Carleton University, 1994

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

© Julie Beeston, 2012 University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

SHARP:

Sustainable Hardware Acceleration for Rapidly-evolving Pre-existing systems

by

Julie Beeston B.Sc., University of Victoria, 1992 M.Sc., Carleton University, 1994

Supervisory Committee

Dr. Micaela Serra, Co-Supervisor (Department of Computer Science)

Dr. Jon Muzio, Co-Supervisor (Department of Computer Science)

Dr. Sudhakar Ganti, Departmental Member (Department of Computer Science)

Dr. Kin Li, Outside Member (Department of Electrical and Computer Engineering)

ABSTRACT

- Dr. Micaela Serra, Co-Supervisor (Department of Computer Science)
- Dr. Jon Muzio, Co-Supervisor (Department of Computer Science)
- Dr. Sudhakar Ganti, Departmental Member (Department of Computer Science)
- Dr. Kin Li, Outside Member (Department of Electrical and Computer Engineering)

The goal of this research is to present a framework to accelerate the execution of software legacy systems without having to redesign them or limit future changes. The speedup is accomplished through hardware acceleration, based on a semi-automatic infrastructure which supports design decisions and simulate their impact.

Many programs are available for translating code written in C into VHDL (Very High Speed Integrated Circuit Hardware Description Language). What is missing is simpler and more direct strategies to incorporate encapsulatable portions of the code, translate them to VHDL and to allow the VHDL code and the C code to communicate through a flexible interface.

SHARP is a streamlined, easily understood infrastructure which facilitates this process in two phases. In the first part, the SHARP GUI (An interactive Graphical User Interface) is used to load a program written in a high level general purpose programming language, to scan the code for SHARP POINTs (Portions Only Including Non-interscoping Types) based on user defined constraints, and then automatically translate such POINTs to a HDL. Finally the infrastructure needed to co-execute the updated program is generated. SHARP POINTs have a clearly defined interface and can be used by the SHARP scheduler.

In the second part, the SHARP scheduler allows the SHARP POINTs to run on the chosen reconfigurable hardware, here an FPGA (Field Programmable Gate Array) and to communicate cleanly with the original processor (for the software).

The resulting system will be a good (though not necessarily optimal) acceleration of the original software application, that is easily maintained as the code continues to develop and evolve.

Table of Contents

Su	Supervisory Committee ii			
Al	Abstract iii			
Ta	ble o	of Contents	iv	
\mathbf{Li}	List of Tables viii			
Li	st of	Figures	ix	
A	cknov	wledgements	xi	
De	edica	tion	xii	
1	Intr 1.1	oduction Thesis Roadmap	1 2	
2	Bac	kground and Rationale	4	
	2.1	Motivation	5	
	2.2	Research Questions	7	
	2.3	Definition of Terms and Concepts	9	
		2.3.1 Reconfigurable Hardware	12	
		2.3.2 HDL Versus C	13	
		2.3.3 SHARP in the context of codesign	17	
	2.4	How the SHARP process addresses the research questions	18	
3	The	SHARP process	21	
	3.1	What does SHARP Do?	21	
	3.2	How is SHARP Used?	24	
		3.2.1 File \blacktriangleright Open	26	

		3.2.2 Preferences \blacktriangleright Constraints		
		3.2.3 Preferences \blacktriangleright Board Characteristics		
		3.2.4 Recalculate \blacktriangleright SPs This File (or Directory/Directory Structure)		
		3.2.5 File \blacktriangleright Recalculate	1	
		3.2.6 File \blacktriangleright Save	1	
		3.2.7 SHARPdefines.c	1	
		3.2.8 SHARPUserControl.h	:	
	3.3	3 Technical Details of Determining POINTs		
		3.3.1 Identifying POINT statements	1	
		3.3.2 Grouping POINT statements 39	ł	
		3.3.3 Results	ł	
		3.3.4 Limitations of This Release 40	ł	
	3.4 Compiling		ł	
	3.5	3.5 SHARP at Run Time		
	3.6	Calculating the value of a POINT	ı	
		3.6.1 Static Constraints		
		3.6.2 Dynamic Constraints		
		3.6.3 Runtime Constraints	l	
	3.7	POINTs in a Hardware Description Language(HDL)		
	3.8	Board and Implementation Decisions		
	3.9	Discussion	1	
4	Res	ılts 55	I	
	4.1	Basic SHARPening	1	
	4.2	SHARPening Results	,	
	4.3 Running SHARPened Code Results		ł	
	4.4	Best POINTs are Kept		
	4.5	SHARPening Results	1	
	4.6	Vision Statement for SHARP 64	2	
5	Rela	ted Research and SHARP 68		
	5.1	Literary Survey of codesign	ł	
		5.1.1 Architecture design constraints and issues	i	
		5.1.2 Architecture design strategies $\ldots \ldots $ 71		
		5.1.3 Ross Video Case Study of generic architectures	:	

 5.2.1 Partitioning	77 79 a POINT 80 82 82
 5.2.2 Shared Memory for Communication	79 a POINT 80 82 82
 5.2.3 Simulation Based on User Defined Metrics to Determine the Benefit of a 5.2.4 Scheduling POINTs loading to hardware	a POINT 80 82 82
 5.2.4 Scheduling POINTs loading to hardware	82 82
5.2.5 Deadlock and Livelock Prevention	82
5.2.6ScalabilityScalabilityScalability5.2.7Future Changes to CodeScalability	
5.2.7 Future Changes to Code	84
	85
5.2.8 Expandability to New Hardware	85
5.3 Summary of this Chapter	86
6 Proofs	88
6.1 Proof of Deadlock Prevention	88
6.1.1 Deadlock Prevention	89
6.1.2 Deadlock Avoidance	90
6.1.3 Deadlock Detection and Recovery	90
6.1.4 Livelock	91
6.1.5 Conclusions \ldots	91
6.2 Notes on Starvation Prevention	91
6.3 Proof of Data Integrity	92
6.3.1 Shared Memory Structure	92
7 Evaluation	97
7.1 Future Directions	97
7.2 Notes to Future Developers	98
7.2.1 Determining POINTs	99
7.3 What this research accomplishes and does not accomplish	100
7.4 Research Contributions of SHARP	100
7.5 The Timeliness of SHARP	102
References	105
A Monte Carlo Algorithm for Radiotherapy Simulation	113
A.1 What is radiotherapy?	113
- *	114
A.2 Publicly available radiotherapy simulation code	

FPGA		
A.7	Calculating the Radiation of a Beam	119
A.6	Depositing radiation on the particle's path	119
A.5	Interactions by type	117
A.4	Possible Monte Carlo Interactions	116

B FPGA

List of Tables

Table 2.1	The SHARP process compared to existing processes	20
Table 5.1	Comparing and contrasting SHARP to related research	87
Table 7.1	Evaluation of what this research does and does not do	101

List of Figures

Figure 2.1 Sample code in C	
Figure 2.2 Sample code in Verilog	
Figure 3.1 Traditional codesign vs. SHARP.	
Figure 3.2 Flow Chart of the SHARP Proceed	5s
Figure 3.3 SHARP Preferences GUI \ldots	
Figure 3.4 An open file in the SHARP GUI $$	
Figure 3.5 The SHARP Constraints GUI	
Figure 3.6 The SHARP Board Characteristic	cs GUI
Figure 3.7 Timing diagram of the SHARP set \ensuremath{SHARP} set	cheduler at run time $\ldots \ldots \ldots \ldots 42$
Figure 3.8 Flow of control around a POINT $$	at run time $\ldots \ldots \ldots \ldots \ldots \ldots 43$
Figure 3.9 Timing data for a valuable POIN	$T \ldots \ldots \ldots \ldots \ldots 49$
Figure 3.10Timing data for a less valuable P	OINT
Figure 4.1 Results of SHARPening files	
Figure 4.2 Throughput increase results from	SHARP
Figure 4.3 Runtime statistics from SHARP	
Figure 4.4 Results of loading files in SHARF	9
Figure 4.5 The Vision For SHARP	
Figure 5.1 Traditional Partitioning	
Figure 5.2 Takeuchi's Algorithm Partitioning	g loop $\ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots 73$
Figure 5.3 Jaggies	
Figure 5.4 Codesign	
Figure 6.1 Block states in shared memory	
Figure 6.2 Block states in shared memory	
Figure A.1 Code layout	
Figure A.2 Flow chart of particle ionizing cal	culations $\ldots \ldots 120$

Figure A.3 The path of a high energy photon	121
Figure B.1 A conceptual field programmable gate array (FPGA).	123

ACKNOWLEDGEMENTS

First and foremost, I want to thank my supervisor Micaela Serra who was the first person to take the time to understand my vision and encourage me to write it as a PhD thesis, and my co-supervisor Jon Muzio for his patience and understanding in the final stages of this thesis. Your combined support, encouragement and occasional niggling throughout this process have stretched both me and my ideas far beyond what I could have accomplished on my own, yet have allowed me to keep my original vision. You encouraged me when I succeeded, consoled me through temporary setbacks and kept me focused through the long years of bringing this project from an idea to a finished product.

I would also like to thank Ken Kent for the many hours he spent on Skype answering my questions, Li Yu for letting me use his thesis as a case study for my own and Dr. Sudhakar Ganti for helping me get set up in the lab. Each new advancement in technology is built on the advancements that came before it. Your work has made my work possible.

I would also like to thank my co-workers in the private sector jobs I have had over the years. I would like to thank my co-workers at Ross Video for introducing me to the benefits of hardware acceleration. I would like to thank the people at MDS Nordion for showing me how to use my talents to make a real difference in the lives of others. I would like to thank Ambrose University College for teaching me how to express complex ideas to people outside of my field of study. Each of these jobs and the other jobs I have had over the years has given me the building blocks I have needed in this journey.

Finally I want to thank my family. I want to thank my husband David for his unwavering support and willingness to sacrifice to make my dreams come true. I want to thank my mom Cecile Mathews for instilling in me the confidence to succeed and the desire to make her proud of me. Finally, I want to thank my son Sterling for his willingness to discuss formal flaws in logic with me long after most other people would have abandoned the conversation. He has wisdom and insights that go well beyond his years and I look forward to seeing the incredible impact his life will have on the next generation.

DEDICATION

Sterling, this one is for you. I love you more than mashed potatoes.

1 Introduction

The current Central Processing Unit (CPU) cycle is to fetch instructions, decode them, execute them and store the result. Because of this well known cycle the most common methods of executing code faster are based on making the CPU faster, using multiple CPUs, reducing the number of instructions or using parallel algorithms. Hardware acceleration is distinct from the other strategies because it allows true parallel processing on a single processor, not just the illusion of parallel processing one can get on a single CPU.

This research encompasses a semiautomated system that gives almost all the benefits of hardware and its acceleration for a small amount of the cost. The resulting infrastruture requires very little training to use, fits in with standard testing procedures

Notable Quote:

One day when my son was three years old he sat at the dinner table with his curly blond hair and beautiful blue eyes and said in his sweet, young voice, "Mommy, your meatloaf tastes perniciously insipid. Can I have Mac-and-Cheese instead?" As his mother I did not know if I should be impressed at his vocabulary, insulted by his description of my food, or concerned that he needed to spend more time with children his own age. One thing I did take from that was to spice up my meatloaf ... and my writing. Therefore, at

the beginning of each chapter you will find a "Notable Quote". They are related to the chapter but in no way required reading.

and does not interfere with future development. It is not always the best answer to every speed issue, but it takes very little effort to decide if it is the right answer for a particular piece of software.

This infrastructure is called SHARP, which stands for "Sustainable Hardware Acceleration for Rapidly-evolving Pre-existing systems". SHARP is able, with user guidance, to select encapsulatable portions of the code, translate them to a HDL and to allow the HDL code and the original code to communicate through a flexible interface.

SHARP is a streamlined, easily understood infrastructure which facilitates this process in two phases. In the phase 1, the SHARP GUI (An interactive Graphical User Interface) loads a program written in a high level language, scans the code for candidate SHARP POINTs (Portions Only Including Non-interscoping Types) and then automatically translates the most promising such POINTs to a HDL. The infrastructure needed to co-execute the updated program is also generated, with clearly defined interfaces.

In the phase 2, the SHARP scheduler allows the SHARP POINTs to run on the chosen reconfigurable hardware, here an FPGA (Field Programmable Gate Array) and to communicate cleanly with the original processor (for the software). Profiling and evaluation complete the process.

1.1 Thesis Roadmap

This first chapter defines the organization for the rest of the document, including this roadmap, together with a brief introduction to the importance of reconfigurable computing in the context of hardware acceleration. Both chapter two and chapter six discuss the related research in this field. Chapter two explores the related ideas and concepts that are fundamental to understanding the new research of SHARP and are therefore placed before the discussion of SHARP itself. Chapter two also discusses the motivation for hardware acceleration, defines the research questions needed to be answered and how SHARP is uniquely designed to answer those questions. Chapter five compares and contrasts the new research of SHARP directly with other such research and has been placed after the discussion of Sharp itself. It contains both a literary survey of codesign in general and a literary review, comparing and contrasting SHARP to the research that most closely parallels it. Chapters three and four are the heart of this research. Chapter three describes the SHARP process in detail and chapter four presents the results of running SHARP on sample code.

Chapter six presents the proofs for deadlock prevention and data integrity.

Chapter seven is an evaluation of the research itself. It revisits the research questions presented in chapter two and discusses how well SHARP addressed these questions. It concludes with a discussion about why SHARP is well timed to be developed and released in the context of recent technological developments.

2 Background and Rationale

Notable Quote:

In 1935 the Austrian physicist Erwin Schrodinger devised a thought experiment to explain the paradox of the Copenhagen interpretation of quantum mechanics applied to everyday objects.

In the thought experiment a cat is placed in a sealed box with a flask of poison that will break at a random point in time. So long as the box is closed, the cat can be considered both alive and dead at the same time. It is only when the box is opened that the cat is truly one or the other.

In the popular television situation comedy "The Big Bang Theory" a main character, Sheldon, used this analogy to respond to Penny's question of whether or not she should go out with Leonard. Until she tried it (opened the box) she would never know if the cat was alive or dead. This is true for any real research. Until one opens the box and explores the research area one does not know what one will find.

The amusing but confusing (without this context) quote came earlier when Leonard was complaining about Penny rejecting him and the following conversation ensued between Leonard and Sheldon:

Sheldon: Okay, look, I think you have as much of a chance of having a sexual relationship with Penny as the Hubble telescope does of discovering that at the center of every black hole is a little man with a flashlight searching for a circuit breaker. Nevertheless, I do feel obligated to point out to you that she did not reject you. You did not ask her out.

Leonard: You're right. I didn't ask her out; I should ask her out.

Sheldon: No, no, now, that was not my point. My point was, don't buy a cat.

This chapter explores the background necessary to fully understand what this new research (SHARP) offers, including explaining the motivation for this new research and stating the main research questions.

The motivation for focusing resources and energy on SHARP is the first topic discussed in this chapter, followed by an articulation of the interesting research questions to lay out what this new work is designed to achieve. The initial necessary definitions, terms and concepts needed to understand this new solution are presented. Finally a discussion of the context of codesign explores how SHARP can answer the research questions.

2.1 Motivation

Faster! The consumer's demand for faster computing speed has driven the principle of Moore's Law. ¹ In his paper[Quinn 2004], Quinn explains why it is often impractical to simply wait for faster CPUs:

"Of course you could simply wait for CPUs to get faster. In about five years single CPUs will be 10 times faster than they are today (a consequence of Moore's Law). On the other hand if you can afford to wait five years, you must not be in that much of a hurry!"

The author was talking about using parallel computing, as in using a multi-processor computer system supporting parallel programming. Another way to make a program faster is to use advanced programming techniques to make critical functions process at their maximum speed. The new research in this thesis focuses on yet another way to stay ahead of the curve: acceleration using reconfigurable hardware.

Why is there so much effort towards making programs faster? The answer is that it is not *just* making the programs run faster; it is allowing programs to achieve more than they could otherwise. A good example is in the live video industry (e.g. live television). To do special effects, such as making the transition between camera shots look like a page of a book turning, each frame must be calculated and then displayed. Since television displays about

¹The complexity for minimum component costs has increased at a rate of roughly a factor of two per year ... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer. [Moore 1965] However consumer demand continues to stay ahead of this curve.

30 frames per second, calculations that cannot be completed in 1/30 second are useless. Faster computing increases the range of possible effects.

Clearly the benefits of faster computing time in live video are also beneficial to a large number of real time systems, but there are also great benefits to non-real time systems as well. A good example is the Monte Carlo Algorithm for Radiotherapy Simulation, which was the initial inspiration for this research. This algorithm is used in making the treatment of radiotherapy safer and more effective and it is explained in more detail in Appendix A. For the purpose of this discussion the two most important aspects of the Monte Carlo Algorithm are that it is crucial, life-saving application and that it is computationally expensive.

The Monte Carlo Algorithm is so important that at least three other methods of speeding up its processing time are already being actively exploited. It is run on the fastest CPUs available and the source code for the algorithm has been made available to the public so that the world's top scientists can make improvements to its code's efficiency. The BC Cancer Clinic at the Royal Jubilee Hospital exploits the benefits of parallel programming for this algorithm by using a single main computer to coordinate the calculations, and 12 sub-computers processing a batch of calculations at a time. Even with all of this, one complete calculation still takes over 12 hours. Twelve hours of time using hospital equipment is expensive, thus often faster but less precise algorithms are used instead. A faster Monte Carlo Algorithm would mean more treatments being given the best possible chance of success.

This new research fits nicely into the examples given above. It gives the developers one more tool to keep ahead of the curve without interfering with the processes already in place or adding excessively cumbersome new procedures.

2.2 Research Questions

[Bishop 2003] laid a solid foundation for this thesis by demonstrating that the use of a coupled configurable computer can be seamless to the end user and goes on to predict when it is advantageous to use such a configuration. The benefits of a coupled configurable computer [Bishop 2003] clearly laid out leads to a pre-research question in this new research: whyis hardware acceleration, possibly using reconfigurable computing, not being fully exploited *now?* The answer is that parts of the process require specialized, sophisticated training and thus the real need is for a simpler process to expand the possible applications for this technique. The avenues being explored in contemporary research regard new languages such as Impulse- C^2 and System C^3 vet no project has achieved full automation, but each has achieved significant levels of success. The differences between the new solution presented here and other projects are discussed in more depth in Chapter 6. At this point it is important to know that the primary differences between this new research and the other projects are the maintainability, flexibility and expandibility of the SHARP process as a whole. The target for this new research is code that will continue to develop after it has been accelerated through the use of reconfigurable hardware.

The primary research question is:

Can the process of hardware-accelerating an existing piece of software be simplified to the point that it approaches automation without limiting future development of the software?

The importance of defining a process to the point where it can be fully understood and replicated is crucial to science. In computer science, the importance of proper design practices, including documentation, is thoroughly explained from the beginning. In my career I had the opportunity to work with a manager who had no formal training but became the

²http://www.impulseaccelerated.com/

³http://en.wikipedia.org/wiki/SystemC

senior/only software developer in a small company. Having no formal training, he did not understand the implications of a statement he once made: "There is no point documenting code. I can read the code as easily as I can read the documentation." Anyone with formal training would have known how much that statement was restricting his company's growth. Without documentation no one could ever code-inspect his programs, collaborate with him on projects or interact with his interface without fully understanding his implementation. In short, he was limiting the company's product to the amount of code that could be written and maintained by a single person.

Formal design practices have had a profound impact on the field of computer science by allowing larger, more complex programs to be developed. The benefits are widely recognized even if the practices have no noticeable impact on the speed or efficiency of any particular program. In the same way one can assert that the introduction here of an automatable process of hardware-accelerating existing code is valuable even if it does not speed up a particular program. Hardware acceleration that must be done manually is limited because it is a time consuming, specialized task. Automating the process makes it scalable to larger projects and within the reach of smaller companies who do not have the resources to maintain specialized support staff dedicated to hardware acceleration.

As mentioned before, this is not the first attempt at automating the process of hardware acceleration[Wolf 2003]. Many advancements have been made in automating certain parts of the process. Therefore answering the primary research question requires looking at a number of smaller questions first. These questions arise from studying the current processes of hardware acceleration.

The two most important sub-questions that arise are:

- what *truly usable* tools already exist,
- and then how can the missing pieces be best implemented.

These questions can be decomposed further by comparing existing tools, but the most im-

portant first step is to define a framework in which these sub-questions can be answered independently, otherwise the possibilities and permutations would be unmanageable. One final basic question is how to measure the success of this, or any, new research. Complete automation in its purest form requires no human intervention whatsoever. It is actually quite rare in any field. This leads to another question: How automated does the SHARP process have to be in order to be considered a success?

It is important to emphasize one aspect: although the purpose of hardware acceleration is, generally, to make programs run faster, it is not the primary purpose of this new research. This new research focuses on the ease with which an arbitrary existing software system can be manipulated so that portions of it can be run on hardware and other portions run in software, with all the portions communicating with each other effectively. In other words, this new research may or may not improve performance directly. Instead it provides a framework and tools that allow the ability to improve performance. The performance of programs using this project will continue to increase as faster hardware becomes available and new third party tools are developed. The future success of this project is not the external tool and hardware themselves, but in how easily and quickly those new tools and hardware can be integrated into this framework making them exploitable by the end user. In this regard, the main goals of this thesis are to evaluate how easily the performance enhancement is achieved and how easily that performance impact can be measured and replicated.

2.3 Definition of Terms and Concepts

Two prevalent questions in Computer Science are how to make programs easy to develop ⁴ and how to make programs run faster[Quinn 2004]. Often these goals are in harmony. Pro-

⁴In the 1990's Nortel devoted a large amount of resources to simply restructuring its code into layers. This restructuring was not done to add any features, make the code run faster nor make the code size smaller. It was done primarily to make the code easier to upgrade and maintain in the future [Heldman 1992], [Freeman 1996].

grams can be made to run faster with faster hardware or smarter compilers without impairing future development of the program. Sometimes these are conflicting goals when the enhancements added to make a program run faster require a sophisticated level of programming skill that make programs harder to develop and maintain [Moser 2008].

Hardware acceleration is the use of specialized hardware to speed up the processing of procedures so they run faster than they do if they are run on general purpose hardware [Wolf 2003]. It can be argued that the amount of skill required to write programs on both specialized hardware and general purpose hardware plus the coordination of the interactions between the two platforms makes hardware acceleration fall into the category of *difficult* enhancements. That is, although they speed up execution, they also make development and maintainance much harder. A large amount of research has gone in to moving hardware acceleration more towards the first category: items to speed up programs without making them harder to develop and maintain [Wolf 2003], [Gerstlauer 1970], [Dong-hyun 2009] etc.

General purpose hardware is generally programmed in a high-level programming language (such as C/C++, C#, Java, etc), while specialized hardware is programmed in a hardware description language(HDL, such as VHDL or Verilog). These two programming language groups differ by more than just syntax and semantics. There is a fundamental paradigm shift between them and this is discussed in more details later [Bishop 2003]. There is already a great deal of research into making the process of shifting between the two language groups easier by allowing users to write HDL code in a C-like language [Black 2010], [Gerstlauer 1970], [Kamat 2009].

This is where the field of *Codesign* enters the horizon. Codesign refers to the synergistic system design process for the design, development and integration of complex applications (often an entire embedded system), where part of the solution is geared to specialized hardware, while other parts are programmed in a high level software language for general purpose hardware [Jerraya 2005]. Codesign is discussed in greater detail in Section 5.1.

In general three different strategies for partitioning in codesign. [Kent 2009]

- 1. Start with a software description, or implementation, of the system and selectively migrating components of the system to hardware until the desired constraints are met.
- 2. Start with a hardware description, or implementation, of the system and shift functionality of the system to software until a suitable solution is attained.
- 3. Start with a generic description that is neither hardware nor software based, but rather a specification of the system's behavior. From this, use heuristics to divide the system between the two partitions.

Since SHARP starts with legacy code written in software it clearly falls into the first of these three partitioning strategies.

There is an accepted convention of referring to the portions of the code running on specialized hardware as "running on the hardware", while the portions running on the general purpose hardware are "running in software". This is a misleading distinction since the code running on general purpose hardware can also be said to be "running on hardware" and it relies on the user to understand the importance of the definite article "the". In general it is better to avoid such precarious distinctions, but since this nomenclature is so widely used this document has adopted it as well.

This new research work cannot really be categorized as purely in the field of codesign as it does not offer a platform to design and implement a new artifact from beginning to end. Instead it uses codesign principles to transform an initial working software system into a codesigned one with the final purpose of accelerating its performance. *The focus remains on the process of transformation from pure software to codesign system.*

SHARP is the new acronym for Sustainable Hardware Acceleration for Rapidly-evolving Preexisting Systems. SHARP falls into the category of codesign because it is a set of tools combined with a process that simplifies the development of the hardware accelerated equivalent of an existing program that was developed for general purpose hardware. SHARP builds on the existing developments in codesign and fills in some of the missing pieces to make the whole process work better.

In this new research, the term *SHARP* is used to refer to the process and tools as a whole. A *SHARPenable* program is a program written in a high level programming language designed for general purpose hardware which can benefit from the enhancements SHARP provides. *POINT* is an acronym for Portions Only Including Non-Interscoping Types. POINTs (often referred to as SHARP POINTs) are segments of code that have been partitioned from a larger SHARPenable program and are capable of being translated to an HDL. During the SHARPening process, POINTs are identified, translated to a HDL and provided with the infrastructure to determine at run time if they should be run on the specialized hardware or the general purpose hardware. The term SHARPening also includes the process of determining which POINTs are most valuable to run on the specialized hardware.

The primary tool in the SHARPening process is the SHARP GUI (Graphical User Interface). The SHARP GUI is an interactive tool that loads a program written in a high level general purpose programming language, identifies the POINTs in that program based on user defined constraints, and automatically generates much of the infrastructure needed to evaluate the value of POINTs and for the SHARPened program to run.

2.3.1 Reconfigurable Hardware

A configurable computer is a computing device that provides hardware that can be modified at runtime to efficiently compute a set of tasks. [Bishop 2003]

A modern configurable computer is generally built from a PLD (Programmable Logic Device) which provides the ability to modify both the control logic and datapaths of a portion of a computer in real-time. A PLD is an integrated circuit that implements a digital circuit designed and programmed by a user. Programmable logic devices include the FPGA (Field

Programmable Gate Array) discussed in Appendix B and CPLDs (Complex Programmable Logic Devices).

2.3.2 HDL Versus C

To understand the difference between an HDL and C code (as an example of a high level software language), it is useful to look at what is happening on the underlying hardware.

The CPU follows the familiar machine cycle:

- 1. Fetch an instruction.
- 2. Decode the instruction
- 3. Execute the instruction
- 4. Store the result.

The CPU does this one instruction at a time. Even in a multi-tasking system with pipelining, there is only the illusion that several processes are running on the CPU at the same time. In reality, the system is simply giving a few cycles to each process before moving on to the next process. The only way to get two instructions to execute simultaneously is to add a second CPU, or with a super scalar CPU which is able to execute, for example, fixed and floating point instructions symultaneously.

In contrast, on any hardware there is true parallel processing given by the simple physics and layout of circuits. If a switch flips to close a circuit between a power source and a light bulb, the light will turn on as quickly as the electrical current can travel from the power source to the bulb. If there were several lights attached to the same switch, the electricity would not go to each light individually and sequentially. Instead, the electricity flows from the switch to each light bulb at the same time.

When programming in an HDL one thinks of *processes* running concurrently, assuming an architecture mimicking the digital layout. Consider the code from Figure 2.1 written in C. The [codeX] sections have been omitted for simplicity.

Figure 2.1: Sample code in C

If this code is run on general purpose hardware it will go through the normal, fetch, decode, and execute cycle for each statement. That means that if select equals seven, it will take one clock cycle to determine that it is not zero, a second to determine that it is not two, a third to determine that it is not five and a fourth to determine that it is seven. That means there are four clock cycles before [code4] can be run.

The equivalent code in Verilog is in Figure 2.2.

```
module mux4( select, q );
                       //Select has the values 0-7
input[2:0] select;
                        // so it needs 3 bits.
output g;
                        //In this code the output
req
    q;
                          is a register
wire[2:0]
          select:
always @( select
                  or d)
begin
       select == 0)
   if(
          [code1];
        =
       select == 2)
           [code2];
       select == 5)
           [code3];
       select == 7)
           [code4];
end
endmodule
```

Figure 2.2: Sample code in Verilog

The code looks similar, but in Verilog, the code is not executed sequentially. In the first clock cycle all options for select are checked simultaneously so [code4] can start being executed in the second clock cycle.

This parallelism available on special purpose hardware is one of the primary things that HDL programmers try to exploit.

The intricacies of a hardware circuit are often cumbersome to model in a software language, and the pipelining and parallelism can be near impossible to model in a pure software language. There are two widely used HDLs, Verilog and VHDL. For the purposes of this paper they are interchangeable since their differences are handled by third party software. Their programming structure represents a significant paradigm shift from software programming and is often confusing to developers who only have software development experience.

The true parallelism obvious in any digital circuits is a tremendous advantage. However

programming at this level is not at all productive and thus we need the translation from high level languages to executable code which runs on circuits designed with inherent parallelism. The bridge has been gapped somewhat by the introduction of FPGAs (Field Programmable Gate Arrays, described below) where the hardware is directly reprogrammable and reconfigurable at the hardware level (that is, it is redesigned into a new piece of hardware as necessary), yet the languages used for them still mainly remain in the area of HDL. New languages exist such as SystemC [Black 2010], SpecC[Gerstlauer 1970] and Handel-C[Kamat 2009] which are basically permutations of software language and provide a compiler-type translation to HDL and then implementation on an FPGA. However these languages are far from easy to use, the tools are difficult and a developer still needs to understand both the hardware and software paradigms.

The FPGA normally has a much slower clock speed than the general purpose hardware, but still can still make an application run faster because of its inherent parallelism, while general purpose hardware must complete each task before it can move on to the next task - or at best complete one task per processor in a multiprocessor platform. Unfortunately this benefit also introduces complexities that software designers rarely have to deal with. Furthermore this is difficult enough when designing and implementing a system aimed 100% towards reconfigurable hardware. If the system needs to be codesigned and partitioned between an FPGA and a regular CPU, the problem becomes immensely more complex. In fact, the most complex portion is coordinating the communication between hardware and software. This interface is also the least stable portion as for every iteration in a system design, when portions of the execution may move from software to hardware and viceversa, it must change.

2.3.3 SHARP in the context of codesign

The exact definition of codesign varies from source to source, but in broadest terms it refers to processes that simplify the overall development and integration of complex systems, where part of the system is developed for hardware (in an Application Specific Integrated Circuit (ASIC), Field Programmable Gate Array (FPGA), etc.) while other parts are developed in high level software programming languages.

Codesign is an interesting field with a lot of potential, but as [Noguera 2002] points out, most traditional codesign implementations are application specific. It has been used mainly for embedded systems, where the main implementation implies having closely connected software and hardware portions and a well-defined interface[Kent 2003]. This paper also notes that for the codesign processes that currently exist they are either so narrow that they only apply to a particular hardware or so broad that they give co-designer little direction to address each of the steps (such as partitioning) within the co-design processes.

SHARP falls under the general umbrella of codesign. SHARP bridges the gap by being both generic enough to encompass a large range of software applications and upgradable to new hardware configurations, while still giving clear instructions for each of the codesign steps. To do this, SHARP parallels the approach of being middleware[Santambrogio 2006], where the upper side of the middleware is completely hardware independent and the lower side only requires a few modifications between platforms. The upper side of this middleware in SHARP is the SHARP GUI (Graphical User Interface) that scans the software code for SHARP POINTs. The identification of SHARP POINTs is based entirely on the interface of data and is completely hardware independent.

Building off the Transaction Pair Model introduced in [Bishop 2003] the value of a SHARP POINT is based on constraints that are hardware specific, and are either static constraints or dynamic constraints. The dynamic constraints are determined directly by the SHARP system at run time so the user does not need any specific hardware knowledge to use them. The static constraints can be tweaked based on the specifications which can be obtained directly from the manufacturer's application notes and sample designs [Bindal 2005], or they can be left to the default values if the user has little hardware design experience. The weight that each of the constraints have in the final equation is completely under the user's control.

The bottom part of the middleware is the SHARP scheduler. It is by nature hardware specific. SHARP is flexible enough to encompass a large range of hardware, but the hardware must at least have the following capabilities:

- Accept and buffer input.
- Accept a signal that the input is ready.
- Direct input to the correct SHARP POINT in the hardware.
- Signal output is ready.
- Buffer output until signal is read.
- Indicate which input a specific output belongs to.
- Load a SHARP POINT dynamically at run time optional.

If the hardware had all those capabilities, then the SHARPening GUI can identify the portions of code that can be run on the hardware using the SHARP scheduler.

2.4 How the SHARP process addresses the research questions

In software development two flows are considered: the flow of control and the flow of data. As [Schaumont 2008] observed, control dependencies are artificial, but data dependencies are a genuine property of a design specification. Since SHARP focuses on data flow, this next section will take a closer look at the current research that more closely parallels SHARP.

The existing tools were a major driving force in determining the framework for SHARP. A large part of the process has already been automated, for example, the process of translating C like code to an HDL. Since this new research builds on these tools it also adapts much of the same development life cycle. Table 2.1 compares the SHARP process to the process currently endorsed by two commonly used products, Handel-C and Impulse C. The *italicized* sections in the table highlight the benefits of SHARP since they are the portions of the process that are automated. This new research defines the current processes as being partially automated. Step 3 is a complex and difficult step and it is a tremendous advantage to have the part of the process automated, but steps 1 and 4 require a large amount of specialized and advanced skill.

This SHARP process is much more automated than any previous process. Step 4 is the only truly manual part of the process and it only has to be done once per board (which really normally implies only once per design, no matter how many iterations and changes). SHARP automatically generates a few sample files for some common boards used in the development of SHARP, plus a few files with stubs of interfaces to help the user connect to a completely new board. This process is much more automated and requires a lot lower skill level. As discussed later in Chapter 5, if step 4 needs to be done (for a new board), it can be done by someone with software development experience at the college level. If step 4 does not need to be done (e.g. the board is already defined by SHARP), the process can be done by someone with merely a high school education. This process is discussed in greater detail in Chapter 3.

Step	Handel-C and Impulse C	SHARP
1	Manually determine which parts of the program are best run in hardware.	Automatically determine which parts of the code can be run in hardware based on the definability of the interface, allowing user defined restrictions for the C to HDL translator.
2	Manually determine the interface to the parts of the code to be run in hardware.	Automatically generate code to allow the communication between the hardware and the software.
3	Automatically generate the HDL code from the C-like code.	Automatically generate HDL code.
4	Manually coordinate the interaction of the hardware components and software components. (Note that there are some advancements in this area such as having a common interface for the communications that make it less manual.)	Manually create board specific functions to load and unload HDL code dynamically at run time.
5		Based on user defined metrics and simulation automatically determine which of the portions of code defined in step 1 are best to run on the hardware.

Note: The italics text indicate processes that are done automatically, while the normal text processes are done manually

Table 2.1: The SHARP process compared to existing processes

3 The SHARP process

SHARP is a framework (including the tools necessary to support that framework) designed to answer the research questions presented in Chapter 2 by demonstrating an almost automatic process for hardware accelerating legacy code. The SHARP *process* is the new solution to hardware acceleration offered by this project. This chapter gives a practical view of *what* SHARP does and *how* it does it. The following chapters analyze how well SHARP achieves its goals.

Notable Quote:

Be kinder than necessary because everyone you meet is fighting some kind of battle.

- T.H. Thompson and John Watson

As iron sharpens iron, So a man sharpens the countenance of his friend.

- Proverbs 27:17

The first section of this chapter gives a high level view of what SHARP is. The second section shows how the SHARP tools are used from the user's perspective. To allow this section to focus on the interface, the technical implementation details of the GUI are detailed in sections 3 through 6. Section 7 describes the design decisions made in developing the initial release of this project. The final section discusses the challenges and triumphs in developing this project.

3.1 What does SHARP Do?

The SHARP process is a specific sub area of hardware/software codesign, where the starting point is a complete program already existing in software. Legacy systems are in some ways simpler to codesign and/or hardware accelerate than systems that are still being designed since the specifications are clearly defined by the existing software code. Yet they also present their own unique challenges which SHARP addresses.

Whether a system is still in the early stages of design or is a legacy system, a vital step in hardware/software codesign is partitioning the functionality, that is, determining which parts are better suited for hardware and which parts are better suited for software. The approaches that look at the partitioning problem at the system design stage have as a primary concern the exploitation of the benefits of the hardware (e.g. parallelism). Instead, for SHARP the maintainability and the ability to automate the analysis of the process take first priority, while the traditional concerns of codesign are exploited only on those sections of the code that can be separated as SHARP-POINTs. SHARP is designed to hardware accelerate legacy software systems without having to redesign them or limit future changes.

Figure 3.1 depicts a key aspect where SHARP deviates from the traditional codesign approach. In traditional codesign the hardware and software are developed independently and integrated at every step. In SHARP, the software version of the code already exists and is maintained. The SHARP-POINTS are automatically generated from the software, but the original software version of the code is maintained to allow future development of the code by developers. Since each SHARP-POINT represents a section of code (one or more statement each as discussed in section 3.3) that is fully defined in both software and hardware, it can be processed in either platform depending on user defined criteria.



Figure 3.1: Traditional codesign vs. SHARP.

This figure depicts an abstract, graphical view of traditional codesign versus SHARP. In traditional codesign the hardware and software are developed independently and integrated at every step. In SHARP, the software version of the code already exists and is maintained. The hardware sections (labeled HW in the diagram and referred to as SHARP-POINTS in this document) are automatically generated from the software and can be processed either

in hardware or in software depending on the hardware constraints of space.

The non-inter-scoping aspect of the SHARP-POINTs is the primary concern and is the first main criterion for the logic used by SHARP. The hardware accelerator may have limited memory space and may not have access to the overall data pool or other such resources. Therefore SHARP-POINTs must be those sections of code using only temporary local variables that do not exceed the scope of the SHARP-POINT. This excludes any section of code that includes global variables, function calls, exceptions etc.

As discussed below the cost of the interface for switching contexts from software to hardware can be expensive, not to mention the possibly high cost of designing such an interface and updating the code to use the interface. This is one of the biggest challenges in general codesign – if not indeed the most difficult, especially if it is to be done in a non ad-hoc fashion. This cost is a main focus of SHARP. Thus at the partitioning stage there is a user defined value to determine the smallest number of contiguous instructions that can be considered a SHARP-POINT. This is the second basic criterion in the SHARP logic.

From these criteria for SHARP-POINTs the corresponding hardware code is automatically generated. All of the traditional *t*ricks for codesign can potentially be employed in this automatic code generation. As a baseline, the heuristics and logic used by off-the-shelf C to HDL translators/compilers are incorporated in this first release, but in future releases of SHARP more clever tricks will continue to be added.

3.2 How is SHARP Used?

While it is extremely important to examine the design decisions for SHARP and its performance, the easiest way to explain what SHARP does is by starting from the user's perspective. In this section all the steps in the SHARPening process are explained.

The bulk of the SHARPening process is done in the SHARP GUI. The SHARP GUI is a powerful, new, *user-friendly* GUI that guides the user through the SHARP process. Typically the user will go through a subset of the following steps:

- File ►Open
- Preferences ►Constraints
- Preferences ► Board Characteristics
- Recalculate ►SPs This File (or Directory/Directory Structure)
- File ▶Recalculate
$\bullet~{\rm File} \blacktriangleright {\rm Save}$

The remainder of the process requires manipulating the following two files:

- SHARPdefines.c
- SHARPUserControl.h

These steps are depicted in Figure 3.2. After these steps are completed, the resulting code can be run as usual. The best SHARP POINTs discovered so far are automatically loaded into the hardware at run time. The code generally runs faster, but at least runs no slower. Each of these steps is discussed in its own subsection.



Figure 3.2: Flow Chart of the SHARP Process

3.2.1 File ►Open

Although from the user's perspective this is simply opening a file, the GUI is actually doing a lot of work in the background for this step. It does all of the following:

- Loads the file in memory.
- Parses, tokenizes and lexically analyses the file.
- Assigns a color code to each token type.

When the GUI has completed the work required to open a file, the file is displayed in the GUI with all the different types of tokens appropriately color coded by the lexical analysis. Figure 3.4 shows an example of an opened file. Note that the choice of colors is arbitrary. If the user wishes to change the color code it is possible from the Recalculate Preferences menu (see Figure 3.3) on page 27).



Figure 3.3: SHARP Preferences GUI

This GUI allows the user to change the color scheme of each of the element types in an opened document in SHARP. The color scheme of the GUI shows the currently selected colors for that item. For example, local variables are shown in black with a light green background. To change that scheme, press **Local Variable** in the back GUI and a color map shows up like the front GUI.



Figure 3.4: An open file in the SHARP GUI

3.2.2 Preferences ► Constraints

This is an optional step for advanced users. The constraints GUI (in figure 3.5) allows the user to define which items should be given the most consideration when determining the value of a POINT. This gives the advanced user the ability to tweak the system for the unique characteristics of that system, such as limits on I/O points, limited space on the board or a slow communications bus. Once these changes are made, they are stored in "Directory/SHARP/Constraints.txt" so the less advanced users do not need to update them further. For this release of SHARP there are basic constraints, but the section on future work also discusses ways to expand this GUI.



Figure 3.5: The SHARP Constraints GUI

3.2.3 Preferences ► Board Characteristics

This step is only done when a new board is added to the system. The GUI window, as shown in Figure 3.6, opens to allow the user to enter the specific characteristics of the new board. These characteristics are used as the parameters in the **File** \rightarrow **Recalculate** step to allow the C-to-Verilog translator to optimize the Verilog code for the specific board. The



changes are stored in "C:/SHARP/SHARPBoardCharacteristics.txt".

Figure 3.6: The SHARP Board Characteristics GUI

3.2.4 Recalculate ►SPs This File (or Directory/Directory Structure)

This step is certainly the *core* of the SHARP process. In this step the GUI parses the code, isolating POINTs and determines the input and output for each point. After this step, the POINTs are displayed in a new color (blue by default), but the code itself is only updated with four pre-compiler directives and an include statement as follows:

- SP_LOAD_X
- SP_START_X
- SP_END_X
- SP_RESYNC_X
- #include "SHARP/SHARPControl.h"

The strategic value of SHARP toward not interfering with future development of the code is that these are the *only* changes that are added to the original code. Since this step actually updates the code slightly, it can be undone by using the **'Clean'** menu option even after the changes are saved.

3.2.5 File \blacktriangleright Recalculate

This step sends each point to the third party C-to-Verilog translator available at "c-to-verilog.com". Since third party software is used, the system has to package the code so that it can be used as input. The GUI packages each POINT as a function with inputs and outputs, and creates an html file with the board characteristics as previously defined. The user then has to press a button to synthesize the Verilog code and save it in "C:/SHARP/SHARPDB". This further generates two files, namely "SP_X.bit" and "SP_X.v". The original C version of the POINT that was used to generate the Verilog is stored in "C:/SHARP/SP_X.c". Later the system can determine if a POINT has been updated and thus proceed to generate an updated .bit file.

3.2.6 File \blacktriangleright Save

The original file is updated with the addition of the pre-compiler directives discussed earlier. The system also automatically generates a number of helper files in "Current Directory/SHARP" as follows:

• SHARPControl

This *read-only* file schedules when SHARP POINTs are loaded to/from the hardware and handles all communication.

• SHARPdefines_NetFPGA_DL SHARPdefines_Spartin SHARPdefines_Stub SHARPdefines_Stub_DL Only one of these files is needed and the details are discussed in the next subsection.

• SHARPUserControl

This file allows the user to define if SHARP is collecting statistics on POINTs and if so which kind of statistics. This is done simply by un-commenting specific lines as the comments in the file instruct. This file is discussed in the last subsection.

3.2.7 SHARPdefines.c

In previous steps the system automatically generated a number of SHARPdefines files, including:

- SHARPdefines_NetFPGA_DL
- SHARPdefines_Spartin
- SHARPdefines_Stub
- SHARPdefines_Stub_DL

Only one of these files is needed. However since the result is board specific, several boards are given as well as a few stubs for new boards. The extension DL indicates that the board supports dynamic loading of POINTs at run time. The user copies the appropriate SHARPdefines file to "SHARPdefines.c" and updates it as needed. If this is a new board, the following functions also need to be updated:

Board Specific Connections

- initializeConnection
- disconnect

Loading .bit files to FPGA

- loadSharpPoint
- unLoadSharpPoint

Status Information on loaded POINTs

- $\bullet~{\rm spaceLeft}$
- loadedSharpPoints

Give input and get output for loaded POINTs

- runSharpPoint
- abortSharpPoint
- getSharpOutput

Since this is the most specialized and difficult step in the current process, possible ways to simplify it are discussed in the future work section.

3.2.8 SHARPUserControl.h

Following the instruction in SHARPControl the user can, and should, run regression tests in all three modes by simply un-commenting commands.

- In the first mode, each POINT is run in software to collect statistics on how many clock cycles this takes and how often the POINT is run.
- In the second mode, each POINT is run in hardware to collect statistics on how many clock cycles this takes and how much space is needed.
- In the third mode, POINTs are moved in and out of hardware to calculate the relative value of loading different POINTs when other POINTs are already loaded.

During testing, statistics on how each POINT performs is stored in

"C:/SHARP/SHARPDB_Data_X". The final value for the POINT is calculated based on the values given in *Constraints* and is stored in "C:/SHARP/SHARPDB_Status_X". After POINT values are calculated, the user puts SHARPControl back into non-statistics mode for efficiency.

The code is now ready to be released!

3.3 Technical Details of Determining POINTs

Isolating SHARP POINTs is an extremely important first step in the codesign of legacy software code, but it is conspicuously missing from many commercial codesign software packages. For example, in the *Impulse C Frequently Asked Questions (FAQ)* one finds:

Q: Does Impulse C allow me to compile my legacy C applications to hardware?

A: Impulse C is a set of library functions that support parallel programming using data streams, signals and shared memories. The CoDeveloper compiler tools are capable of accepting one or more C files containing such programs (multiple C subroutines connected via streams, signals and memories) and generating equivalent low-level hardware. As such, Impulse C and CoDeveloper are not specifically intended for taking large C applications that are written using traditional C programming techniques (function calls, etc.) and compiling these applications to equivalent hardware.

Impulse C is not designed to work with large scale C programs without manually analyzing and updating the code. SHARP, instead, is not only designed to work with large scale applications, it also is minimally intrusive to the original code, since it only inserts a few pre-compiler directives into the code.

¹http://www.directinsight.co.uk/products/impulsec/codeveloper-faq.html#3

Although the identification of POINTs within the code is done automatically from the user's perspective, the GUI performs an elegant algorithm for determining POINTs. SHARP POINTS are composed of one or more Statements. Statements are composed of one or more tokens. SHARP applies the same meaning to tokens as is used by most compilers. When compiling a high level language into binary, the high level code is tokenized and lexically analyzed. A token is the smallest unit that has any meaning, and lexing is the process of giving that token meaning.

Clearly the process of tokenizing and lexing is language specific. Since the initial release of SHARP is C based, SHARP tokenizes the way a C compiler would and then lexes the tokens into the relevant SHARP types. SHARP types in C include:

- comments,
- types (i.e. int),
- variables (safe and unsafe),
- operators (i.e. +, -),
- brackets,
- colons,
- numbers,
- C keywords,
- C flow control keywords (i.e. Return, GoTo and Break),
- sharp point directives,
- white space,
- SHARPenable functions (which include both functions and procedures) and

• the SHARP type simply referred to as 'other' since tokens in this set are not members of any other set.

The determination of safe and unsafe variables is also language specific. In general terms, a safe variable can only be accessed via one point of entry that SHARP can see the full access of. Unsafe variables can be updated from more than one location. In C unsafe variables include volatile variables, pointers, arrays (which can be thought of as pointers) and any safe variable that has been de-referenced using the '&' operator, effectively making it a pointer. Any variable that is defined outside of the group of files being SHARPened is assumed to be unsafe.

The process of SHARPening follows these steps:

Step 1. (optional) Obtain a list of SHARPenable functions that the C to HDL translator already understands. These may include math functions (i.e. sqrt) or other well known functions (i.e. sizeof()). Add to this list any SHARPenable functions that follow the rules outlined above. This step is optional since it will not inhibit SHARP's functionality; however, if it is omitted it will cause SHARP to not be able to find some possible POINTs. Step 2. Find each SHARP statement that meets the criteria given below and determine the inputs and outputs of each statement. For each statement, each variable in the statement is assumed to be both an input and an output initially. The rules for limiting a variable to being only an input or only an output for a statement are language specific. Note that if the rules for a new language are not determined, the impact of leaving all variables as both inputs and outputs is that the LOAD (explained in Section 3.5) may not be able to bubble up as high as it otherwise could, and the RESYN may not bubble down as far as it otherwise could. This will make the SHARP POINT less parallelizable, which may impact performance, but it will not change the function output.

Step 3. Group the statements into POINTs using the criteria given below. If the SHARP POINT contains a call to a SHARPenable function, the function gets unwound and

expanded directly into the HDL code, but the original C code is not changed. The inputs and output of a SHARP POINT is the union of inputs and outputs respectively of each of its statements.

Step 4. Determine the LOAD and RESYNC points for each SHARP POINT. The LOAD point must occur after any of the inputs for that SHARP POINT have be written to outside of the SHARP POINT. The RESYN point must occur before any of the outputs from the SHARP POINT are read outside of the POINT.

3.3.1 Identifying POINT statements

The smallest unit that can be considered a SHARP POINT is a statement. A statement is composed of tokens. In C a statement is generally terminated by a semicolon, although some statements are terminated by the end of a line (i.e #define or //comment), and others are terminated by a bracket }. Statements terminated by a bracket } are referred to as "SP_functionStart". There are 6 types of statements:

SP_no – The Statement contains at least one 'other' token, a bad variable or a flow control keyword.

SP_noOp – The statement can be part of a SHARP point but cannot stand alone as one. (e.g. a comment.)

SP_yes– A statement that is directly translatable to HDL with a clearly defined interface that can be completely encapsulated.

SP_scopeStart- An SP_yes terminated by a '{', but not an SP_functionStart.

SP_functionStart – A function declaration terminated by a '{' that does not contain any bad variables.

 $SP_scopeEnd - A \ bracket \ '\}$ '.

3.3.2 Grouping POINT statements

The rules for grouping contiguous statements into SHARP POINTs are as follows:

- A SHARP POINT must begin with a SP_yes or a SP_scopeStart.
- A SHARP POINT must end with an SP_yes or an SP_scopeEnd .
- A SHARP POINT must not contain a SP_no.
- A SHARP POINT must contain the same number of SP_scopeStarts as SP_scopeEnds.

The rules for a SHARPenable function are similar except for the following:

- A SHARP FUNCTION must begin with a SP_functionStart .
- A SHARP FUNCTION must end with an SP_scopeEnd .
- A SHARP FUNCTION must not contain a SP_no except a flow control return or break as the last statement.
- A SHARP FUNCTION must contain the same number of SP_scopeStarts as SP_scopeEnds plus an extra SP_scopeEnd for the SP_functionStart.
- A SHARP FUNCTION must not be recursive or contain a circular reference (i.e. function A calls function B, function B calls function C, ... function Z calls function A,).
- For SHARP FUNCTIONs the inputs are the parameters for the function; the outputs are the return value of the function and any of the parameters that have been passed by reference including any pointers.

3.3.3 Results

The end result of discovering a SHARP POINT implies finding a portion of code that can be totally encapsulated and can be run at any point between the LOAD point and the RESYNC point without affecting the rest of the code past the RESYNC point. In section 6.3 a proof is shown that the data integrity is maintained in this process.

3.3.4 Limitations of This Release

For the initial release "#define" statements are not unwound. This implies that all pre-compiler directives will be labeled as "SP_no" statements. This means that some statements that might possibly be sharpened may not be. This could also cause an issue if the define statement opens a scope that it does not later close (or closes one it did not previously open).

3.4 Compiling

The compile process is environment specific. The user needs to have a compiler for both the C code and the HDL code and may have to do some investigation to discover the best way to compile the points. POINTs are designed to work as standalone functions that can be compiled and loaded separately and are therefore compatible with many existing environments. For example if the user is using a NetFPGA board then the points can be loaded into the ISE Design Suite which generates a Makefile to compile the code.

3.5 SHARP at Run Time

When the code is running as software on the CPU and it arrives at a SHARP POINT, it determines the following:

- the inputs of the SHARP POINT,
- the outputs of the SHARP POINT and
- the process on the hardware.

It then allocates the necessary number of blocks in shared memory and loads this data. This process then goes to sleep and waits for a signal that the scheduler thread is ready for it to continue. The scheduler thread either accepts the request and initiates the POINT running in hardware, or rejects the request and stores the input. It then goes to sleep until either the output is ready from the hardware or the output is requested from the main thread. Figure 3.7 shows a timing diagram of the main thread and the SHARP scheduler. The code then follows one of the two possible execution paths, depending whether it is to be run in hardware or in software. Figure 3.8 shows a pictorial representation of this process.

In the original code the pre-compiler directives SP_Load, SP_Start, SP_End and SP_Resync have been added. This maintains the layout of the original code, but the code is not run in linear order (top to bottom). Instead it follows one of the two paths (left side or right side of Figure 3.7) depending on if the POINT is run in hardware or software. In both cases, the code runs as normal until SP_Load is reached, then the inputs of the POINT are sent to the SHARP scheduler in another thread. In the main thread both paths do the remaining non-POINT code until SP_Resync where they request the outputs from the scheduler. At this stage the two paths differ depending on the response. In the path on the left of Figure 3.7, the scheduler determined that the POINT should be run in hardware, loaded it if necessary and initiated it with the inputs. When the output was ready it was stored until the output request was made. The outputs are used to update the code and the program continues as normal. In the path on the left of Figure 3.7, the scheduler determined that it was better not to run the POINT in hardware so it stored the original inputs and marked the transaction as rejected. The original inputs are used as the inputs to the POINT which is then run in software.



Figure 3.7: Timing diagram of the SHARP scheduler at run time



Figure 3.8: Flow of control around a POINT at run time

In the original code (center) the pre-compiler directives SP_Load, SP_Start, SP_End and SP_Resync have been added. The code follows one of the two paths (left side or right side) depending on if the POINT is run in hardware or software. If the POINT is run in hardware it follows the procedure below.

Software

Hardware

- At the SP_Load directive, fork a new process to load and run the hardware version of the POINT.
- Continue to the SP_Start directive.
- Jump over to the SP_End directive and continue to the SP_Resync directive.
- Request the result from the hardware and re-integrate it in to the code.

- Load the POINT into hardware if there is room and it is not already loaded.
- Send the input to the POINT if it is loaded.
- Store the result.

If the POINT is run in software it follows the procedure below.

Software

Hardware

- At the SP_Load directive, fork a new process to load and run the hardware version of the POINT.
- Continue to the SP_Start directive.
- Jump over to the SP_End directive and continue to the SP_Resync directive.
- Request the result from the hardware but get inputs instead.
- Jump back to the SP_Start directive and continue to the SP_End directive.
- Jump past the SP_Resync directive.

- Determine it is better not to run the point in software.
- Store the inputs.
- Reject the request for outputs and return the inputs.

3.6 Calculating the value of a POINT

Determining which POINTs to load in hardware is one of the most crucial steps in the SHARP process. The steps outlined in Section 3.3 find a number of potential POINTs, but give no precise indication of the value of running each POINT in hardware. Some POINTs might have a major impact speeding up the code considerably; other POINTs might only speed the code up a little bit; still others might slow the code down due to the ovehead of the context switching.

The question is: how to determine programmatically which POINTs are the best to load onto the hardware?

The SHARP constraints GUI in Figure 3.5) displays how the user can control how the value of a point is calculated. Notice that the GUI is separated into three sections. The top section determines the length in software lines of code of each POINT. These constraints are used in the initial determination of the POINTs and are already satisfied before the value of each POINT is calculated. The other two sections, the *static constraints* and the *dynamic constraints* are discussed here. A third section, *runtime constraints* is conspicuously missing. It will be added in future releases.

Static constraints are easily calculatable from the static code, but they are not very accurate. The dynamic constraints are much more accurate, but they can only be calculated at runtime. The static value is calculated and can be used immediately, but it should be given very little weight relative to the dynamic constraints. Later when the dynamic value is calculated via the runtime testing discussed in section 3.2.8 it should be given more weight. In Figure 3.5) the static constraints are one tenth the weight of the Dynamic constraints Dynamic constraints are calculated during the runtime testing. The static value added to the dynamic value is the value stored for each point in the SHARP Database.

3.6.1 Static Constraints

To calculate the static constraints SHARP first scans the database for POINTs with the most inputs and outputs and stores the result in 'maxIO'. Similarly it stores in 'maxV' the POINTs with the most lines of code. The initial value of each point is then calculated with the following formula, where W1, W2 and W3 are the user-defined Static Constraints weights from the Constraints GUI. In Figure 3.5) each of these values are one.

Initial Value = W1(maxIO-inputs-outputs) + W2(lines of code in software) + W3(maxV-lines of code in .V)

These three constraints were chosen so the user could limit the number of I/O elements to the SHARP POINTs if the hardware board (e.g. the FPGA) had such limitations. In this way one can obtain the most lines of software code per line of hardware code.

3.6.2 Dynamic Constraints

The benefit of each SHARP-POINT is theoretically completely calculable by determining the difference between the number of clock cycles needed to process the SHARP-POINT in software and the number of clock cycles needed to process the SHARP-POINT in hardware multiplied by the number of times the SHARP-POINT is called in a normal execution. The number of cycles needed for the SHARP-POINT to be executed in hardware includes the processing time required plus the time required to switch the context from software to hardware and back again. This gives the formula:

$$B = N(S - (Ch+H+Cs))$$

where:

B = SHARP-POINT benefit.

S = Clock cycles for the software implementation.

H = Clock cycles for the hardware implementation.

N = The number of times the SHARP-POINT is executed in a normal run.

Ch = The cost of switching from software to hardware.

Cs = The cost of switching from hardware back to software.

It is tempting to tie into an existing profiler like 'gprof' on UNIX systems to determine an appropriate value for N, but SHARP is designed to work on a wide variety of platforms that have different profilers. Also, the values of Ch and Cs are determined by the physical characteristics of the hardware accelerator being used. Since SHARP is designed to work on a wide range of platforms, it implements its own profiler.

To profile code for SHARP, a copy of the code must be updated with the SHARP profiler and go through a few normal runs. On the first run, all the processing is done in software as the SHARP profiler keeps track of the number of times each SHARP POINT is run and the cumulative amount of time spend processing a SHARP POINT.

A naive first run at solving this could simply be calculating the difference in the clock from the time the SHARP-POINT starts and when it ends. In a single threaded environment this would give a reasonably good value, but in a multi-threaded, time-sharing environment the profiler has to take into account the possibility that the SHARP-POINT might begin to process and be interrupted by another thread.

In successive runs, each SHARP-POINT is moved to hardware and statistics are gathered on how long it takes the process to run in hardware. Since the hardware and the CPU are likely to have different clock speeds, all values are stored in CPU clock cycles. The following three values are stored:

- The average time to run in software (S);
- The average time to run in hardware (H); (Note: this includes the *Ch* and *Cs* values defined above)

• The number of time that POINT was run in normal testing.

The final value is then calculated from W1, W2 and W3 weights from the GUI. In Figure 3.5) each of these values are ten. The final formula is:

$$W3(N) * (W1(S-H) + W2(S/H))$$

For example the POINT data shown in Figure 3.9 shows a POINT that has been run ten times so N=10. The number in the right column of the figure shows the number of clock cycles it took to run this POINT. The S or H in the left column of the figure shows if the data is for a run in hardware or software. This POINT was run in hardware five times with an average time of 44 clock cycles so H=44. This POINT was run in software five times with an average time of 82 clock cycles so S=88. The dynamic value of this POINT is 10*10*(10*(82-44) + 10*(82/44)) = 39900. This is a relatively valuable POINT. The POINT data shown in Figure 3.10 shows a POINT that has been also been run ten times so N=10, but for this POINT H=44 and S=44. The dynamic value of this POINT is 10*10*(10*(44-44) + 10*(44/44)) = 100. This POINT is not as valuable. If the value of S/H is less than one, that means it takes longer to run the point in hardware than it does in software. That overrides all other values and gives the SHARP point a value of -1 and it is never considered for loading on the hardware. Otherwise the static value is added to the dyna<u>mic value for the final val</u>ue.

K:\>more	SP_1data.txt	
H 47		
S	78	
H 4	47	
H 4	46	
S	78	
H 4	47	
S	78	
S S	74	
H 4	17	
H :	31	

Figure 3.9: Timing data for a valuable POINT

K:\>more SP_2data.txt
H 31
S 47
S 46
H 47
S 47
H 47
S 47
S 31
H 47
H 47

Figure 3.10: Timing data for a less valuable POINT

3.6.3 Runtime Constraints

In this first release of SHARP, the most valuable POINTs are loaded into the hardware before execution starts and remain loaded for the entire run. In other words the first release of SHARP does not support dynamic loading.

In later releases individual POINTs will be loaded and unloaded on to the hardware as needed during run time. To do this, two new values will have to be added to the constraints GUI.

The first new value indicates how much better an unloaded POINT will have to be before it can bump a loaded POINT off the hardware. The overhead of switching POINTs on the hardware is considerable, so this value keeps POINTs from being switched too frequently. The second value is a depreciation value. The calculated value from the previous two subsections remains constant, and is given as the running value when the POINT is first loaded and each time the POINT is run, but the running value for the POINT needs to be depreciated each time some other POINT is run so that POINTs that have not been run for a while will get bumped by POINTs that are actively being used. This is useful if there are POINTs that are valuable during initialization but are not used much after initialization, or if the flow of control in the code naturally moves to different section of code as the code runs.

3.7 POINTs in a Hardware Description Language(HDL)

Ideally future releases of SHARP will be so user friendly that the user will not need to know how to program in any Hardware Description Language (HDL). The translation from C to Verilog is done by third party software and the SHARP GUI automatically generates the code to handle the communication between the HDL code and the C code. However, the translation from C to HDL is not a simple translation between languages. It is a fundamental paradigm shift as discussed in Section 2.3.2. It is necessary to have a basic understanding of how a SHARP POINT can be represented in a HDL to have a clear idea of the benefits and limitations.

There are two primary HDLs in common use today, Verilog and VHDL. The choice of which translator to use for SHARP was arbitrarily made based on the availability of a free C-To-Verilog translator.

3.8 Board and Implementation Decisions

One of the earliest research and development goals in this project was to be able to have some sample code actually running on real hardware and not simply on a simulator. Thus the first choice was for the board to be used, containing an FPGA programmable hardware. The selection fell on the *Xilinx Spartan-3E FPGA+ARM+USB-2*² development board. In order to connect it via Ethernet to a PC, the Eclipse³ IDE's WINSOCK⁴ DLL was used. Finally the translator from C to HDL was chosen to be the C-to-Verilog⁵. This configuration decision was driven initially by the decision to have the connection

²http://www.knjn.com/docs/KNJN FX2 FPGA boards.pdf

³http://download.eclipse.org/eclipse/downloads/

⁴http://www.snible.org/winsock/

⁵http://www.c-to-verilog.com/

between the host computer (PC) and the FPGA using the Ethernet as opposed to USB. The primary reasons were cost and simplicity. Ethernet is a more mature technology and as such the protocols are more firmly established and much of it exists as freeware.⁶ Using Ethernet, the drivers and protocols are already written and socket API is relatively simple. There have been some advance in producing the same protocols for USB such as the emerging USB 3.0 protocol⁷, but since they are still new, they remain relatively expensive, and, most of all, the API still needs to be written manually.

The primary disadvantage of using the Ethernet is that it consumes more resources to implement the TCP/IP stack than an experienced designer could customize for a specific application project on a specific hardware/software platform. However, in the custom case, a new API would have to be written for each Operating system (Linux, Windows, Mac, Solaris, etc.), whereas Ethernet is highly portable, and can be used to connect to any hardware with a network connection.

The choice of the hardware was driven by two criteria: cost and functionality. Although many advanced FPGA boards are available through the University of Victoria and CMC,⁸ using them would limit the test cases to programs that could be loaded and run in the lab. Not wanting to accept that limitation, I decided to give myself a modest budget to purchase hardware for testing and extrapolate from that testing the performance increase I would be able to achieve on the more advanced hardware. There are many FPGA manufacturers to choose from, including Actel⁹, Altera¹⁰, Aeroflex UTMC¹¹, Atmel¹², Lattice

 $^{^{6}}$ It was developed in the 1960-70s although the TCP specifications were initially defined in 1974 (see http://en.wikipedia.org/wiki/Internet_Protocol_Suite), whereas USB was only developed in 1996 (see http://en.wikipedia.org/wiki/Universal_Serial_Bus).

⁷http://www.usb.org/developers/docs/

⁸Canadian Microelectronics Corporation (see http://www.cmc.ca)

⁹http://www.actel.com/

¹⁰http://www.altera.com/

¹¹http://www.utmc.com/

¹²http://www.atmel.com/

Semiconductor¹³, NEC¹⁴, QuickLogic¹⁵, and Xilinx¹⁶¹⁷. The use of Ethernet limited the choice somewhat since through that port it needed to be possible to load and execute a SHARP POINT as well as to communicate data. The budget constraints limited the selection much further. The best performance for price found was the 'Xilinx Spartan-3E FPGA+ARM+USB-2' development board bought on eBay for \$USD299.95 + tax. The decision to use Eclipse, WINDSOCK and C to Verilog was primarily based on availability. Both are either free or come free with software already owned. Thus one can state that this configuration offers a good system at a reasonable cost. One of the most impressive features of SHARP is that even with this low budget environment it is still able to produce preliminary performance improvement. This happened even at the earliest stages, before the system was also ported to the NetFPGA boards available in the lab for the final tests recorded in Chapter 4. This demonstrates how SHARP has helped make hardware acceleration within the reach of the average software development team.

3.9 Discussion

The SHARP GUI itself took over a year to develop and the process as a whole took several years. In that time the underlying technology has changed dramatically. The board used to test the benefits of SHARP was not available when the bulk of the tools were being written. Overcoming challenges often inspires flexibility and resilience. Since SHARP had to be refined as it was developed it has become both portable and easily updatable. The technical complexity of designing the tools for SHARP was a large project. The issue of refining the SHARP project to incorporate new ideas as the project developed was an

¹³http://www.latticesemi.com/

¹⁴http://www.nec.com/

¹⁵http://www.quicklogic.com/

¹⁶http://www.xilinx.com/

¹⁷http://www.interfacebus.com/Programmable_Logic.html

even larger project.

It was a large project, but it was worth the effort. The benefits of codesign were laid out in Chapter 2, and SHARP adds to this valuable and exciting field. SHARP bridges the gap of partitioning the code which is explicitly stated as missing from in the Impulse C project. SHARP's expandability and portability make it an important project in this stage of the surrounding research and technology which justifies the large amount of resources that went in to developing it.

4 Results

This chapter records the results of applying the SHARP process to a variety of third party software and discusses how those results relate to the overall goals SHARP is designed to achieve. The first four sections are a series of test suites that quantify the benefits of SHARP, while the final section ties the results together by giving a vision statement of the ideal environment in which to implement the SHARP process. The test results sections of this chapter are organized into four sets of test suites to reflect the two very distinct aspects of the SHARPening process: (a) The ability of SHARP to automatically identify POINTs and (b) the effectiveness of these points once they have been identified and automatically set up for the user.

Notable Quote:

Before we allow ourselves to be consumed by our regrets, we should remember the mistakes we make in life are not so important as the lessons we draw from them.

> - Closing narration from the Last Supper episode of The Outer Limits

Here is a useful function I developed as an undergrad: void UniversalProblemSolver(Problem a){ if a ll();

else fails (askGrandpa());

- Julie Beeston

The first two suites of tests are designed to verify SHARP's ability to identify POINTs. The first suite is a sanity check comparing the automatically generated POINTs to manually determined POINTs. The second suite of tests confirms the breadth of the SHARP process by applying it to a large number of third party files.

}

The remaining suites of tests confirm the value of the SHARP process by running some of the SHARPened code and confirming that the POINTs identified by SHARP as valuable do indeed speed up the processing speed of the code.

The final suite of tests explores how well the SHARP process scales to larger projects. The last section in this chapter describes the vision for SHARP: the most ideal future for SHARP in a perfect world.

There were many other tests that were run on a smaller, slower Spartan FPGA board including the first test, running SHARPened Monte Carlo algorithm code. The results of those tests are not included in this chapter because they were focused on ensuring that the interfaces for SHARP were correct and that the process could be automated. These tests gave very modest performance increases, if any. The results shown in this chapter are from the more powerful NetFPGA board.

4.1 Basic SHARPening

The first suite is a sanity check comparing the automatically generated POINTs to manually determined POINTs.

Although SHARP is designed to help automate the process of hardware acceleration, the process of verifying the process itself involves some manual steps. This first sanity test is designed to ensure that the SHARP GUI is correctly identifying points. The procedure for this test case is as follows:

- 1. Manually determine the POINTs in a .c file.
- 2. Open the file in the SHARP GUI.
- 3. Automatically generate POINTs in the opened file.
- 4. Compare the manually determined POINTs to the automatically generated POINTs

The first run of this test was performed on sharpDemo.c (shown in Figure 3.4). This file was specifically designed to contain a single POINT. Once this test was complete, more

complex files were downloaded from web sites including a calculator from

http://mycsnippets.blogspot.com/2011/05/calculator-program-in-c-using-using.html and a sum of series calculator from http://sawaal.ibibo.com/computers-and-technology/how-usefunctions-c-factorial-fibonacci-number-entered-user-236513.html. The success of this suite of test cases laid the foundation for the following suites of test cases.

4.2 SHARPening Results

The second suite of tests confirms the breadth of the SHARP process by applying it to a large number of third party files.

The primary benefit of SHARP is the ease with which it can be used to isolate portions of code that can be run on the FPGA and automatically generates the supporting code needed. This made it possible to do a large amount of breadth testing for this process. In just a few hours, 80 files were downloaded from random sites such as http://www.paulgriffiths.net/program/c/ and SHARPened. The process for this stage of testing was as follows:

- 1. Download a piece of third party source code from a code depot.
- 2. Load the software into the SHARP GUI.
- 3. Use the GUI to identify POINTs and generate the HDL code and supporting C code.

The 80 files downloaded are in the following ranges of sizes:

- 63 files 0-99 lines each.
- 12 files 100-199 lines each.
- 3 files 200-399 lines each.
- 1 file 400-599 lines.

• 1 file over 600 lines.



Figure 4.1 summarizes some of the results.

Figure 4.1: Results of SHARPening files

In this graph the vertical axis indicates the number of POINTs found in a particular piece of third party software. The horizontal axis has the ranges for the number of lines in the original source code. For each range there are two statistics: The statistics represented by the grey box is the minimum number of POINTs found in the code in that size range. The black box represents the average number of POINTs found in the code in that size range. The number on top of each box is the height of that box. Note: The largest file tested was from the Monte Carlo code discussed in Appendix A.

In all there were 80 files downloaded ranging in size from 5 lines to 630 lines. POINTs were found in 90 percent of the files, and the eight files where no POINTs were found were all relatively small files of less than 100 lines each. Figure 4.1 shows how as the number of lines in the code increased, the average and the minimum number of POINTs found (and thus the number of possibilities for performance enhancement) tends to increase as well.

4.3 Running SHARPened Code Results

Quantifying the benefits of SHARPening was a manual process and therefore more labour intensive. When the SHARP process is fully implemented it keeps track of timing statistics to determine which POINTs are better to run on the FPGA when there are many possibilities to choose from. However, these statistics are limited to the scope of aiding the user, not the detailed statistics needed to argue the benefits of SHARP as productive new research.

As well, the free software used to automatically generate the HDL code did not produce code that was completely compatible with the NetFPGA board used to test on. As such the code had to be manually updated after it was automatically generated.

The process of testing in this section was:

- 1. Start with a piece of SHARPened code with a single POINT identified.
- 2. Manually update the HDL code.
- 3. Manually update the original code so that it ran 100 times (to get more accurate results).
- 4. Record the time running the code completely in software on a PC that contains an AMD Athlon X2 6000+ processor clocked at 3GHz with 2 Gigabytes of RAM, running CentOS on Linux kernel 2.6.18.
- 5. Record the time running the code completely in software on a PC that contains an AMD Athlon X2 6000+ processor clocked at 3GHz with 2 Gigabytes of RAM, running CentOS on Linux kernel 2.6.18.
- 6. Record the time running the code on the same PC as in the previous step, but with the PC connected via a PCI interface to the NetFPGA, clocked at 125 MHz, has 4 Gigabit Ethernet ports, 72 Megabytes SRAM and 64 Megabytes SDRAM.

7. Compare the resulting times from the previous two steps.

Since this process was more labour intensive, only two programs were tested in this way. The first program computed the square root of a number and the second computed exponents.

Before being SHARPENed, the squareRoot program took an average of 28.1 milliseconds to run. One millisecond was spent on the overhead of the program itself and 27.1 milliseconds were spent executing the POINT. After it was SHARPened, it took only an average 3.8 milliseconds to execute the POINT for a total running time of 4.8 milliseconds. This is a 585% throughput increase for the code as a whole.

The exponent program originally took an average of 14.3 milliseconds to run. One millisecond was spent on the overhead of the program itself and 13.3 milliseconds were spent executing the POINT. After it was SHARPened, it took only an average 6.2 milliseconds to execute the POINT for a total running time of 7.2 milliseconds. This POINT gave a more modest 199% throughput increase for the code as a whole. These results are shown in Figure 4.2.

	Square Root	Exponents
Processing Before Sharpening	27.1 ms	13.3 ms
Overhead Before Sharpening	1 ms	1 ms
Total Time Before Sharpening	28.1 ms	14.3 ms
Processing After Sharpening	3.8 ms	6.2 ms
Overhead After Sharpening	1 ms	1 ms
Total Time After Sharpening	4.8 ms	7.2 ms
Throughput Increase	585%	199%

Figure 4.2: Throughput increase results from SHARP
4.4 Best POINTs are Kept

SHARP finds all possible POINTs and uses direct testing to determine which POINTs are the most valuable and should therefore be loaded on to the hardware. This test suite was designed to confirm that the POINTs identified by SHARP as valuable were indeed the best ones.

The process for this stage of testing was as follows:

- 1. Merge the files from the previous test suite into a single process.
- 2. Use the results of the previous testing for SHARP to determine the value of each POINT.
- 3. Record the time required to run the code in its SHARPened state.
- 4. Manually corrupt the SHARP statistics so that the less valuable POINTs appear more valuable and get loaded and run.
- 5. Record the time required to run the code in this corrupt state. Confirm that this corruption slows down the processing speed.

Since this process was more labour intensive (which further shows the need for a structure like SHARP), only one program was tested in this way. The two files from the previous tests were linked into a single process and SHARP was updated to believe that only a single POINT could be loaded at a time.

SHARP correctly identified the squareRoot POINT as being more valuable and proceeded to load it.

When the program was allowed to without being SHARPened, it took 45.9 milliseconds to run, of which 28.2 milliseconds were spent on the square-root POINT while 14.7 milliseconds were spent on the exponent POINT. The remaining three milliseconds was spent on overhead. Once SHARPened, the running time for the program was 21.6 milliseconds. With the corrupted statistics in the SHARP database it took 37.3 milliseconds, thus confirming that SHARP indeed found the better of the two POINTs.

These results are shown in Figure 4.3.



Figure 4.3: Runtime statistics from SHARP

4.5 SHARPening Results

Since SHARP is designed to work on large software development projects, this suite of tests explores how well the SHARP process scales to large files.

To run this suite of tests, five files were chosen in three size ranges:

- Two Small files (less than 50 statements).
- Two Medium files (about 500 statements).
- One Large file (over 2,500 statements).

The largest file tested was 2558 statements. This is quite large for a C source file (about 120 KB). Although there are programs that are over a million lines, often these programs

are broken into several files, each of which would be SHARPened separately.

Using the test files several stages of the SHARP process were explored. Loading and parsing the file gave the most interesting results. The load and parse process is the first and most vital step in the SHARPening process. As the file is loaded, it is parsed in much the same way a compiler would parse the file forming a parse tree. Once the code is in a parse tree the process of determining POINTs is completely linear, so the process of recalculating POINTs took less than a millisecond for even the largest file. The process of determining the best POINTs is user defined. The more times the code is

run, the more accurate the relative value of POINTs, but even no test runs gives a reasonably good value.

Statements	Time to Load and Parse (msec)
34	203
37	280
508	5242
632	25272
2558	467845
1000	147141
10000	1878875
100000	19196218
1000000	192369642

Figure 4.4: Results of loading files in SHARP

The results of the load time for each of the files are shown in Figure 4.4. The results in the white boxes show the raw data. The longest time to load the largest file was about 7 minutes. The raw data was loaded into Excel, and the TREND function was used to extrapolate the values in the grey boxes. From this data Excel predicts that it would take about 5 hours to parse a file that is 100,000 statements long.

Since loading and parsing the file is the current bottle neck of the process and it is no more complex than compiling the file (which can be done much faster), this part of the process should be the first to be made more efficient, but even in this unoptimized state it is still usable on large files.

4.6 Vision Statement for SHARP

The previous sections in this chapter have demonstrated how each of the different components of SHARP work, but the primary benefit of SHARP is how well these components work *together*. The vision of this research is a SHARP standard resembling the one laid out here that will allow the industry to advance in ways that it could not otherwise.

The primary strength of the standard proposed in this new research is that it was developed by someone with industry experience. Its major weakness is the limitations inherit in a system developed by a single user.

Imagine for a moment the vision of a SHARP standard. Imagine the industry has adopted two, well defined industry standards. The first standard is between programming languages and Software Language-to-HDL translators on defining the interface of SHARP POINTs and making this translation available directly without requiring human interaction with a GUI. The second standard is the interface of the board specific file in SHARP.



With SHARP



Figure 4.5: The Vision For SHARP

Figure 4.5 shows the vision for SHARP. A user wanting to hardware accelerate their code would purchase a *SHARP compliant* hardware accelerator, a *SHARP compliant* compiler and a *SHARP compliant* HDL translator and they would work together. In the future the user could upgrade any of these components with a minimum amount of reintegration. This puts the requirement for expertises into the hands of the designers most skilled in that area. The end users are experts in their own software. That is where they would focus their attention. The people skilled in compilers, translators and accelerators would handle most of the rest of the process.

Programming language compilers have the advantage of being developed by teams of people who are intimately knowledgeable about a particular programming language. A compiler, almost by definition, needs to parse, tokenize and lex a programming language. It is not much more complicated from that point to define the interfaces of all the SHARP POINTs into the SHARP industry defined standard and automatically generating all the supporting SHARP infrastructure that could interact with an arbitrary Software-Language-to-HDL translator and board specific file following the same standard. Software-Language-to-HDL translators have the advantage of being developed by people with expertise in HDLs and can therefore exploit characteristics of a programming language in the context of the characteristics of the hardware board being used. In this area the advancements for SHARP are already actively being explored. In an ideal world this research would continue, adopting the SHARP standard as in interface. Hardware board manufactures have the advantage of knowing the interface requirements for their own particular board. In a perfect world, whenever a new board is developed, the board specific file would be developed and tested with it. The board could also include a recommended Software Language-to-HDL and recommended settings.

With each of these pieces fitting together, each area is free to develop independently and

while still benefiting the process as a whole.

5 Related Research and SHARP

The purpose of this chapter is to put SHARP into the context of the larger body of research which influenced its development. The chapter is divided into two main sections. The first one is a literary survey of codesign in general. The second section is a literary review,

comparing and contrasting SHARP to the research that most closely parallels

SHARP.

Since SHARP is a specific type of codesign, the literary survey in the first section of this chapter traces the history of codesign and explains why it has gained so Notable Quote:

There is a simple game where the numbers one through nine are written on a board and two players take turns claiming each number by placing a token on it. Once a number is claimed, the other player cannot use it. The object of the game is to be the first player to have exactly three numbers that add up to 15.

This game is isomorphic to the game tic-tac-toe. Although Player One has the numbers laid out in order:

$1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9$

Player Two has the numbers laid out in a magic square formation:

8	1	6
3	5	7
4	9	2

In a magic square every row, column and diagonal adds up to 15. A nice benefit of this magic square is that there is no other way to make 15 by adding three digits. Therefore, although Player One is playing a rather tricky game, Player Two is merely playing tic-tac-toe. Often the complexity of a problem is all in how you look at it.

much popularity. This section introduces the importance and difficulties involved in partitioning of code segments targeted for special purpose hardware by extracting case study examples from a real world codesign cycle at a company called Ross Video. The purpose of this section is to demonstrate that the issues SHARP is designed to address are both important and non-trivial.

The literary review in the second section focuses on research that more closely parallels SHARP. In this section many of SHARP's design decisions are compared and contrasted with other similar projects. In this section, specific papers that relate to issues that SHARP specifically addresses are compared side by side with the architecture of SHARP. The purpose of this section is to show how SHARP has been built starting from ideas in the existing practices and to highlight the areas where SHARP deviates from the existing practices into new research.

5.1 Literary Survey of codesign

In 1965, Gordon Moore exposed the woeful inadequacy of many design strategies since the complexity of hardware design was increasing exponentially. His prediction that the complexity of the hardware would double every 1.5 years is now quoted in almost every forum where people study computers as the famous Moore's Law.¹ [Bertels 2012] goes as far as to say that scaling of technology has reached its limits with respect to power, heat dissipation, and suggests that having multiple processors on the same chip is the way of the future.

In the 1990's, the introduction of Integrated Circuit (IC) technology focused the attention on hardware/software codesign. The codesign of complex digital systems has been successfully used since the 1990s and is widely accepted where cheap, reliable fast systems are needed [Mudry 2006]. The limitations of the CAD tools that were initially available and the promise that the need for better tools would only increase led to many conferences

¹The complexity for minimum component costs has increased at a rate of roughly a factor of two per year ... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer. [Moore 1965]

and papers to discuss the issue. "Increasingly the package, and associated discretes, contributed critically to the overall circuit performance, rather than just providing a connection function. These performance issues are critical today and are fast becoming more complex than current CAD tool trends will be able to support." [Franzon 1999] More recent articles such as [Qingyang 2008] note that "TED (Traditional Embedded design) can not satisfy modern requirements because it develops hardware and software independently." A call went out for a Unified Modeling Language. Vahid [Vahid, Frank 2002] and Wolf [Wolf, W. 2003] stepped in to try to fill the void. Calls also went out for a common Hardware/Software programming language, and SystemC and Handle-C were born.

5.1.1 Architecture design constraints and issues

There are three primary issues in codesign: abstracting, communication and partitioning. As with all design methodologies its primary goal is to abstract out the details so the big picture is not lost in the complexity. No matter how complex the system becomes, it must be understandable as a single entity.

The issue of communication is important not only between the different components of hardware, it is also important between the hardware and software developers. Many attempts have been tried to allow the hardware and software to be developed in a common programming language. Since C/C++ is well known, many of these attempts have focused on creating a C-like language that can be used to program the hardware. This has led to languages like SystemC, SpecC and Handel-C. Each of these languages has advantages together with its own draw backs, including lack of proper tool support, and proprietary interfaces, but these issues are assumed to dissolve as the languages gain in popularity and support.

The more fundamental issue not addressed by any of these tools is that when programming the hardware the programmer needs to be thinking of a finite state machine; this is a fundamental paradigm shift from the linear or object-oriented programming done by C/C++ programmers. Perhaps the ever increasing speed of the hardware coupled with clever compilers will minimize the impact of this issue, but it is clear that to get the maximum benefit of performance out of a hardware chip, a Hardware Description Language (HDL) will always be needed.

The final issue of the partitioning of the Hardware and Software design is the most interesting to talk about since it is the one area where everyone has an opinion. Clearly an automated process to split the hardware issues from the software issues would be ideal, but as we will discuss in the *example* section of this paper, the partitioning of hardware and software design is precarious even with expert human intervention. An expert system cannot be expected to make better decisions than the person who programmed it.

5.1.2 Architecture design strategies

Most of the available design strategies include come sort of partitioning of the hardware and the software, parallel development of both the hardware and software followed by the integration of the two at the end of the design cycle. (As depicted in Figure 5.1 on page 72) Two examples of this process are Thomas' and Takeuchi's partitioning algorithms.

Thomas' partitioning Algorithm 1993

Thomas' algorithm focuses on extracting performance critical portions of the code [Thomas 1993]. Once these portions have been identified they are partitioned to be developed in hardware while the remaining bulk of the program is developed in software. The lifecycle of code development follows these steps:

- Characterize the hardware and software performance,
- identify hardware/software partition,
- transform the functional description into such partitions and



Figure 5.1: Traditional Partitioning

• synthesize the resulting hardware and software.

Takeuchi's partitioning Algorithm 1994

While Thomas focused primarily on identifying performance critical portions of code, Takeuchi proposed a more iterative approach that also takes into account the overhead involved in context switching and hardware/software communication. The Takeuchi life cycle is described in the following five steps.

- 1. Algorithm Description. Using an object oriented programming language describe the target processing.
- 2. Module Analysis. Decide if WS (Work Station, i.e. software development) or FPGA is better for a particular module.
- 3. WS and FPGA Instruction Coding. The direct translation from source program to FPGA using VHDL like language may also be available.
- 4. Module Scheduling. This step involves scheduling the algorithms under the current resource allocation. In this scheduling, the FPGAs' program and the WS's

program can be processed concurrently, but we must take the communication time into consideration for the data which are transferred from CPU program to FPGA program.

5. Algorithm Partitioning. There may be the case that the module needs much more time for the communication between FPGA and WS even though FPGA logic can run faster than the processing in WS for this module. Therefore, the repeated scheduling with the algorithm partition is indispensable in this system. Figure 5.2 on page 73 shows the algorithm partitioning loop.



Figure 5.2: Takeuchi's Algorithm Partitioning loop

5.1.3 Ross Video Case Study of generic architectures

A revealing example of the difficulties of hardware/software partitioning is a case study of the development of the Ross Video Squeeze and Tease 3D. Ross Video is a relatively small, family owned business that produces television switchers. The problem itself appears relatively simple: create a video driver that can rotate and move an Over the Shoulder (OTS) box in 3D space.² To do this, the system requires a complex equation that states: *"For every pixel in the destination image, here is the location of that pixel in the source image".*

The initial partitioning of hardware and software was straight-forward. It was decided that the software should determine what the coefficients for the equation should be, and then the hardware would use that equation to calculate the position of each pixel. There were a number of issues that arose during development, but for this case study we consider three that are very revealing. The first was a hardware issue that was correctly handed to the hardware team and solved. The second issue looked suspiciously like a hardware problem but needed to be solved by the software team. The third issue would have been impossible for either team to solve without the input of the other.

Issue 1: Jaggies

Jaggies is the technical term used to describe a diagonal line that is supposed to look straight. Yet because the line is composed of pixels, it appears to go up in 'stairs'. See Figure 5.3 on page 75 for an illustration.

The first iteration of the Squeeze and Tease 3D had jaggies. The issue was correctly identified as a hardware problem and fixed by adding a post-processing filter to smooth out the lines.

Issue 2: Video flipped and off-center after hardware upgrade

When the hardware was upgraded to a faster FPGA, the video on the television screen was

²For a demo of the final product go to: http://www.rossvideo.com/synergy/synergy_md.html



Figure 5.3: Jaggies

flipped upside-down and was off center. The new hardware was clearly at fault since this problem occurred immediately after a hardware upgrade; this was identified as a hardware problem. Unfortunately the hardware team tried to track down the bug for over a month before abandoning all hope of ever resolving this issue. A company-wide search was initiated to find a solution. The solution was found in the software department. Since all the software does is indicate how to flip and move the video, the solution to this problem was simply to send the video flipped and off center to the hardware so that when the hardware flipped it, it looked correct.

Issue 3: New requirements for expanded range of motion

Part way through the development life cycle, the requirements for the range of motion for the OTS increased by several orders of magnitude. The new requirement required pushing the hardware past the originally determined limits and looked like it would require a 64-bit multiply for each pixel. A television switcher has a non-negotiable real time constraint. It must display approximately 30 frames every second to make the motion on the screen look realistic to the human eye. Therefore, all of the computation for a single frame must be done in less than 1/30th of a second. The best hardware available within the company's price range could only do a 32 bit multiply on each pixel within the allotted time while these new constraints would require 64-bit multiplies using the current equation structure. To solve this issue required both the hardware team and the software team working together. The hardware team undertook an in-depth analysis of what variables could be factored out of the existing equations, since a 64-bit multiply was not feasible, but a 64-bit add was easily possible. The second part of the solution was to give the software team these factored variables as 64-bit numbers and see if the movement could be simulated by simply manipulating these variables. This issue could not have been solved by either the hardware team or the software team alone. It was only solvable through the collaboration of both teams.

Case Study Conclusions

The Ross Video case study demonstrates why partitioning is a complex and important task. The three issues encountered in this case study highlight the need for a shift in the design paradigm proposed earlier in this thesis. Rather than split the hardware and software design completely, it is better to have interaction between the hardware and software developers throughout the design phase (as indicated by the dotted arrows in Figure 5.4 on page 77 which are missing from Figure 5.1 on page 72). This is the focus of many of the advancements in codesign research. It can be argued that partitioning is the most fundamental step in codesign. Bad decisions in partitioning can limit a codesign project's success. The choices about how to partition code must be made early in the design cycle, but often the best decisions of how to partition code are not apparent until later in the design cycle. Attempts to simplify - or even automate - the process of partitioning are compared with SHARP's partitioning process in the next section of this chapter.



Figure 5.4: Codesign

5.2 Literary Review: SHARP Compared to Related Research

This section looks at the different aspects of SHARP, comparing them to the latest research in a variety of different fields. Table 5.1 on page 87 gives a quick summary of this section. Each of the functions listed in the far left column is the title of a subsection. The remaining columns refer to other papers that either acknowledge the issues raised by that function or contain a solution that either parallels SHARP or contrasts with SHARP's solution.

5.2.1 Partitioning

Partitioning of code into hardware and software components is perhaps the most difficult and most important process in codesign [Sangeetha 2007, Geguang 2006, Lee 2005, Mudry 2006, Olson 2007, etc.]. [Schaumont 2008] observed that a fundamental idea in codesign is that any given system behaviour that is captured as a software program can also be expressed as a hardware architecture. However, the process of choosing which portions to translate to hardware is often precarious.

[Shannon 2004] notes that a major problem with partitioning is that designers have to make decisions about partitioning with incomplete or inaccurate information.

"A common problem with hardware/software codesign is that the quality of the design is dependent on the partitioner's allocation of resources. However the partitioner must make choices based on estimates and models" [Shannon 2004]

[La Rosa 2005] takes it a step further by saying that those who primarily deal with software development do not want to have to deal with partitioning or hardware concepts at all. They simply want their programs to run faster:

"Software-oriented users are accustomed to high levels of abstraction and can not perform detailed hardware design. They are typically mostly interested in accelerating their software in heavily computation kernels by mapping part of them to the reconfigurable hardware." [La Rosa 2005]

This is not simply laziness on the part of the programmer. The number of criteria for partitioning hardware and software has grown in complexity exponentially over the past few years [Jerraya 2005]. Although the average programmer could (with some amount of training) partition (based on interface) a relatively small program, the process is labour intensive and tedious. Such processes are prone to errors. SHARP automates this process so that it can be run on large programs.

SHARPs portioning process includes scanning the code to isolate parts of code that can be encapsulated, defining the interface required and then providing two implementations for each: a software implementation and a hardware implementation. In its most abstract paradigm, a SHARP POINT is an object. In Object Oriented Design, objects must have a clearly defined interface, while the implementation details can be determined arbitrarily so long as they produce the required output. To do this the implementation must be completely encapsulated. Once the implementation is defined for each POINT, the system can then focus on the flow of data between the POINTs and the main code. In software development there are two flows that are considered: the flow of control and the flow of data. As [Schaumont 2008] observed, control dependencies are artificial, but data dependencies are a genuine property of a design specification. [Geguang 2006] focuses on both control flow and data flow using the familiar rule that SHARP also incorporates, namely that data access by different threads must be done in a particular order if they both access the same data element and at least one of them is a write. [Olson 2007] partitions code based on points of non-determinism in the code. External input/output operations are assigned to hardware while internal (data dependent) operations are assigned to software. The benefit of the Bayian Belief Network (BBN) is that it is easy to understand its graphical nature. The graphical nature is also a drawback since diagrams become too confusing for larger programs and the author admits that it may not be intuitively usable by those not familiar with the BBN symbology. SHARP automatically determines which parts of the code can be run in hardware based on the definability of the interface, allowing user defined restrictions for the C to HDL translator.

5.2.2 Shared Memory for Communication

Communication between the general purpose hardware and the specialized hardware can be done in a number of different ways. Like many other projects, SHARP has a uniform communication protocol to allow it to be scalable.

[Mudry 2006] uses input/output registers for communication between the hardware and the software. All functional units are uniformly accessed, which simplifies instruction decoding

to the single instruction, 'move'. SHARP also uses a uniform communication method. The POINTs are flexible in their content, but the interface is fixed.

[Moss 2007] uses fixed Integrated Design (ID) attributes assigned to each user module so the communication never changes regardless of whether the module is run in hardware or software.

[Noguera 2002] uses a loosely coupled model where there is no shared memory. Instead a direct communication channel is established between both ends of the internal hardware components that need to communicate. Buffering is done via local memory in each processing element.

5.2.3 Simulation Based on User Defined Metrics to Determine the Benefit of a POINT

Once the code has been partitioned, so that portions that can be executed on hardware have been identified, many codesign algorithms (like SHARP) go through a process of determining which portions give the greatest speed benefit by being run in the hardware, while either producing the lowest overhead or by keeping all processors busy at all times. Simulation is one option for determining the benefits, but is a time consuming process that takes orders of magnitude more time than on-chip execution [Shannon 2004]. SHARP opts for a combination of static constraints and run-time profiling. Like SHARP, [La Rosa 2005] enables a software designer who is unaware of hardware design subtleties to assess quickly the costs and gains of executing a single instruction (or POINT) on the FPGA. Like SHARP, [Olson 2007] uses clearly defined metrics to determine the suitability of a particular partitioning, but it also relies heavily on simulation to verify the results. SHARP concentrates on metrics that can be determined statically from the code since that is cheaper, and allows metrics that can only be determined via simulation to be gathered during normal operations without interrupting the flow of the code. [Sangeetha 2007] weights its constraints for Functional Units (FUs) much like SHARP does, but it does not automatically identify what the FUs should be the way SHARP identifies POINTs

There already exist a number of code profilers such as SnoopP (a non-intrusive, real time profiling tool) [Shannon 2004], but SHARP does its own real time profiling to get SHARP specific metrics. Like SHARP, SnoopP takes into account board characteristics such as strict performance, area, and power constraints, and SnoopP like SHARP counts clock cycles rather than executed instructions since pipeline stalls are not accounted for otherwise. However, SnoopP is currently only available for software profiling. It cannot profile the hardware.

Often the number of inputs and outputs is strictly limited by the FPGA board. [La Rosa 2005] parallels SHARP's ability to limit the number of input and output variables. It optimizes memory data structures to minimize memory access for input and output of intermediate values, while SHARP allows the user to define the maximum number of inputs and outputs.

[Olson 2007] uses complexity, bandwidth and frequency, and also notes that hardware delay and power consumption are also relevant. [Mudry 2006] includes how many times a particular line of code is accessed, hardware size and execution time. [Sangeetha 2007] includes communication costs between hardware and software. Each of these statistics can be added to the calculation of the value of a SHARP POINT via the SHARP Constraints GUI.

Most scheduling/allocation algorithms are based on the number of states and the number of resources. [Sangeetha 2007] SHARP's scheduler is based on the value of a specific SHARP POINT compared to the value of the currently loaded SHARP POINTs. Keeping both the CPU and the FPGA busy at all times is the ideal state for a codesigned system. Clock cycles that are wasted while one waits for the other slows down the overall performance of the system. [Noguera 2002] uses number of Discrete Event (DE) classes, hardware execution time versus software execution time, memory requirements and number of Dynamically Reconfigurable Logic (DRL) cells used and reconfiguration time. SHARP also keeps track of hardware verses software execution, but does not necessarily limit it to the hardware being faster. SHARP takes into account the possibility that running a POINT that is slower in hardware can be beneficial if the software can continue to run in parallel.

5.2.4 Scheduling POINTs loading to hardware

The decision to load a specific SHARP POINT onto the hardware or execute a SHARP POINT with a given input is made by the SHARP Scheduler based on the value given to the SHARP POINT via the user defined constraints on the metrics. Like SHARP, [Noguera 2002] also defines a dynamic scheduler but it does not automatically identify blocks (i.e SHARP POINTs) that can be scheduled. It does have an algorithm for partitioning, but the partition value is not considered dynamically. Like [Noguera 2002] model, SHARP POINTs are non-preemptive in hardware. [Noguera 2002] notes the value of pre-fetching hardware components to minimize latency. SHARP also does a limited version of pre-fetching by loading SHARP POINTs at the earliest point in the code where all input is available, but the pre-fetching in SHARP is limited to the scope in which the SHARP POINT is defined. Later releases of SHARP could look at improving this.

5.2.5 Deadlock and Livelock Prevention

[Lee 2005] observes that deadlock is not a critical issue with today's embedded systems since they use very few processors and custom hardware resources. However as embedded systems grow in complexity and numbers of resources there will be a greater focus on deadlock issues. SHARP's primary concern with deadlock is ensuring that SHARP does not introduce deadlock into an existing system.

Deadlock can be dealt with via detection, prevention or avoidance [Lee 2005]. If deadlock detection is used, then the system itself is already equipped to recover if deadlock occurs. If deadlock avoidance is being used, then although the hardware accelerator is a new resource that could be considered in allocating resources, it has no impact on the flow of events. The hardware accelerator is not considered since if a SHARP POINT cannot be loaded at the time of execution then the software portion of the POINT is executed instead, so the original algorithm for deadlock avoidance will suffice.

If the system is using deadlock prevention then SHARP does not introduce deadlock since for deadlock to exist, 4 conditions must be present:

- No Preemption
- Hold and Wait
- Mutual Exclusion
- Circular wait

Deadlock prevention consists of ensuring that at least one of these conditions does not apply. [Garcia-Molina 2008] Since all SHARP POINT activities are handled linearly, circular wait cannot be added to the system. SHARP cannot introduce the possibility of deadlock to a system, but it can expose a pre-existing possibility of deadlock. For example if two processes (A and B) require the same resource and deadlock can occur if A requests the resource first, it could be that in the legacy code B always ran faster so B always requested the resource first. SHARP could expose this possibility of deadlock by speeding up process A. Conversely, however, it is just as likely that this theoretical deadlock could occur if process B requests the resource first. In this case SHARP could actually make deadlock less likely by speeding up process A. SHARP could also introduce the possibility of livelock. Livelock exists when a process is repeatedly denied a resource and may never be allowed the use of that resource [Lee 2005]. Clearly if SHARP speeds up high priority processes so they access resources more frequently this could cause low priority processes to be repeatedly bumped.

5.2.6 Scalability

The manual process of hardware accelerating code is labour intensive and therefore cannot scale well to larger problems. Several other researchers have looked at automating the process, but still their solutions are computationally expensive. For example, [Mudry 2006] only scales to programs that are several hundred lines.

[Geguang 2006] notes that the partitioning problem is NP-complete and therefore does not scale well to larger problems. To decrease the impact of this limitation of scalability, this paper (like [Noguera 2002]) uses heuristics to determine good partitioning when the programs become large and the search space becomes exponentially larger. [Geguang 2006] uses a heuristic algorithm called the Tabu algorithm. The paper does not explain how loops are handled and it also requires the hardware execution time of a process to be 80 to 100 times faster than the software execution time. This requires either super fast hardware, or a high degree of expertise on the part of the practitioner to find portions of software that are ideally suited to be run on the hardware. SHARP settles for a slightly less ideal solution for partitioning which allows the process to be automated.

SHARP avoids the NP-complete problem altogether since SHARP's partitioning process only looks for possible POINTs, not the best POINTs. The value of each POINT is determined later. This makes the computational complexity of SHARPs partitioning process linear. This does have the disadvantage that the best solution may not be found by SHARP, but it has the incredible advantage that SHARP is scalable to even very large projects.

5.2.7 Future Changes to Code

[Dong-hyun 2009] notes that few studies have examined the cost of codesign decisions after the development even though the market changes. The original combination remains static even if a new partitioning would introduce new benefits.

Part of what make SHARP so versatile is that the bulk of the work is done automatically and the manual portions are done in response to new hardware, not new code. Therefore the user does not need to do a lot of extra work if the underlying code changes.

[Noguera 2002] points out that most traditional codesign implementations are application specific. It has been used mainly for embedded systems, where the implementation implies having closely connected software and hardware portions and a well-defined interface [Kent 2003].

Automatic partitioning of the code part is what makes SHARP so versatile. [Qingyang 2008] also does semi-automatic partitioning but does not handle it well when the underlying software changes. SHARP is unique in how well it handles changes to the underlying code. SHARP not only automates the partitioning, it also makes unobtrusive changes to the original code that can be automatically removed.

5.2.8 Expandability to New Hardware

[Olson 2007] notes that a major limitation in much of the research geared towards partitioning for pre-existing hardware platforms is that they are so geared to a particular hardware configuration that they cannot be ported to a different hardware configuration. [Sangeetha 2007] also noted that major modifications are required to adapt to new technology with much of the existing codesign processes.

One solution to this dilemma is to encapsulate the portions of the process that deals with the software from the portions that deal with the hardware. Like SHARP, [Schaumont 2008] focuses on applications that have a flexible component written in software with a fixed part written in hardware. SHARP addresses this problem by making the partitioner independent of the hardware scheduler. Even if there is a dramatic shift in the hardware paradigm, the software partitioner could still be used with a new scheduler. Smaller shifts in the hardware characteristics, like latency, can be optimized for a particular hardware platform by customizing the constraints.

[Sangeetha 2007] notes a limitation that SHARP is also vulnerable to, namely "rounding". *"Finite word length effects"* can be different on different platforms if there are different word lengths or if addition and multiplication are done in a different order. This may be true even if a single piece of hardware is chosen, but it may be accentuated if several different types of hardware accelerators are utilized.

5.3 Summary of this Chapter

SHARPs design is strongly influenced by the results of other researchers who have sought to automate or partially automate the process of hardware accelerate new or existing software. This chapter demonstrates how SHARP has derived both its motivation and design by adding strategic elements to the existing research.

SHARP derives its motivation from the observations many other researchers in this field as well as real world case studies of how co-design is currently being done and how the current generation and hopefully next - of users would like it to evolve. From this SHARP has created a co-design framework.

SHARP uses its unique co-design framework to incorporate as much of the third party software as possible, but adds strategic portions to make the process more automated and more scaleable.

Function	Acknowledges the	Similar	Contrasting
	issue	Solutions	Solutions
Partitioning	[Shannon 2004] [La Rosa 2005] [Jerraya 2005] [Schaumont 2008] [Sangeetha 2007] [Geguang 2006] [Lee 2005] [Mudry 2006]	[Geguang 2006]	[Olson 2007]
Shared Memory for Communication		[Mudry 2006] [Moss 2007]	[Noguera 2002]
Simulation Based on User Defined Metrics to Determine Benefit of a POINT		[La Rosa 2005] [Olson 2007] [Shannon 2004] [Dong-hyun 2009]	[Dong-hyun 2009] [Olson 2007] [Sangeetha 2007] [Noguera 2002]
Scheduling POINTs loading to hardware		[Noguera 2002]	
Deadlock and Livelock Prevention	[Lee 2005] [Garcia-Molina 2008]		
Scalability		[Noguera 2002] [Geguang 2006]	[Mudry 2006] [Geguang 2006]
Future Changes to Code	[Noguera 2002] [Kent 2003]		[Qingyang 2008]
Expandability to New Hardware	[Olson 2007] [Sangeetha 2007]		[Schaumont 2008] [Sangeetha 2007]

Table 5.1: Comparing and contrasting SHARP to related research

6 Proofs

Many people are satisfied with simply knowing that that the system works as intended, but for those who want the whole magilla this chapter contains the proofs that have been referred to in the rest of this document such as deadlock prevention and data integrity.

Notable Quote: A proof is a proof. And when you have a good proof, it's because it's proven. - Jean Chretien

Many of the aspects of SHARP can be proven, but SHARP does make some initial assumptions about the original code and the programming language the original code is written in. The following assumptions are made:

- The legacy code does not suffer from deadlocks or starvation.
- The C-to-VHDL translator is works correctly. (Given specific input it will produce the same output in a finite amount of time in VHDL as it does in C).
- Setting a flag is an atomic operation in shared memory.

Assuming that the assumptions are all true, the following sections prove:

- SHARP does not introduce deadlock, but may introduce starvation.
- SHARP maintains data integrity as data is being passed from main memory through shared memory to the FPGA and back again.

6.1 Proof of Deadlock Prevention

Deadlock is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does. [Silberschatz 2009]¹ For example gridlock in a

¹The concepts in this subsection and the following subsection are based on chapter seven of this reference.

busy city. In computer systems, competing processes prevent each other form completing. There are three primary ways of dealing with deadlock: Prevention, Avoidance and Detection and Recovery. There is a fourth option of not dealing with it at all, but if the original code does not deal with deadlock then SHARP does not either. The first three subsections deals with each of these options. A fourth subsection discusses livelock.

6.1.1 Deadlock Prevention

There are 4 conditions that must be present in a system for deadlock to occur. Deadlock prevention is the process of making sure that at least one of these conditions cannot be present. The 4 conditions necessary for deadlock are:

- 1. Mutual exclusion
- 2. Hold and wait
- 3. No pre-emption
- 4. No circular wait

If the original code prevents deadlock, one of these conditions must, by definition not be present in the original code. There are two places where data enters shared memory, and two places where data exits shared memory. This could potentially (in a different set-up) cause mutual exclusion, hold and wait and no pre-emption to be added to the system, but since SHARP does not wait for the shared memory indefinitely, but instead processes each function in software as soon as the hardware resources are not available (or even just running slower than the software), if any one of those condition did not exist in the original code, SHARP will not introduce them.

The fourth condition, no circular wait comes from the definition of the POINT itself. Since POINTs have no *inter-scoping* elements, each POINT only uses local resources, so it cannot be involved in circular wait; for circular wait to exist in a SHARPened system it must also exist in the original code.

Therefore if there is no possibility of deadlock in the original system, SHARP will not introduce it.

6.1.2 Deadlock Avoidance

Deadlock avoidance can only be done when the system knows in advance how many resources each process will need. The system will not allow any process to start if it cannot allocate all the resources the process will need.

SHARP does introduce a new resource (the FPGA), that ideally should be considered in the deadlock avoidance allocations, but as mentioned in the previous subsection, since SHARP does not wait for this resource, even if it is overlooked it will not cause issues for the system.

6.1.3 Deadlock Detection and Recovery

Deadlock detection and recovery consists of allowing the system to enter deadlock, detecting it and using an algorithm to get out of it. Typically this algorithm will consist of rolling back some processes. If a POINT is running in hardware when the parent process is rolled back, this might make the output for that process not be needed and therefore never read.

An ideal way for SHARP to deal with this in future releases would be to roll back the POINTs in hardware as soon as the software process is rolled back, thus freeing FPGA to execute other POINTs. A less ideal solution would be to have SHARP at least do garbage collection on the un-needed results.

The current release of SHARP does neither of these, and leaves it up to the end user to resolve any issues if a different recovery strategy is employed.

6.1.4 Livelock

Livelock is similar to deadlock, but in livelock the state of the processes continually change relative to each other. For example if two people meet in a narrow hall way and they both move to the same side to allow the other to pass, and then (realizing the way is still blocked) they both move to the other side only to find the way is still blocked. Many systems this by having a random amount of a wait time before trying another option. This way the people in the narrow hall (or the competing processes) will move to the other side at different times and the first one to move will be able to make progress and get out of the other one's way.

If this is the algorithm that the original system uses then SHARP will work well. If there is another algorithm used, then it will be up to the end user to resolve any issues.

6.1.5 Conclusions

SHARP works best if the original code uses Prevention or Avoidance. If the original code does Detection and Recovery SHARP might need to be updated. If the original system did nothing to deal with deadlock, it is possible that it was lucky enough to never encounter deadlock and SHARP will expose the flaws by changing the timing, but there is no way for SHARP to avoid this.

6.2 Notes on Starvation Prevention

Starvation is an issue in multi-tasking systems where a process is perpetually denied resources and can therefore never finish its task. It is usually caused by a simplistic scheduling algorithm. For example suppose there are three processes, A, B and X; A and B have high priority, while X has a low priority. If the scheduling algorithm is to let highest priority waiting process have any available resource there could be a situation where A and B alternate getting the required resource and X never gets a chance to run.

If there was a system with a simplistic scheduling algorithm in which starvation was possible, SHARP could cause starvation by changing the timing since it will speed up some processes.

6.3 **Proof of Data Integrity**

Data can be corrupted when two separate processes access the same data element and at least one of them is a write. [Garcia-Molina 2008]

The software side ensures mutual exclusion via semaphores which lock each block before updating it. Semaphores are well established. They are used to ensure mutual exclusion between software processes. So only a single software process interacting with the hardware is all that needs to be considered.

6.3.1 Shared Memory Structure

The shared memory is partitioned into blocks. Each block is the maximum size needed to hold the largest SHARP POINT. The SHARP POINTs are different sizes because they require different amounts of space to store inputs and outputs.

Each block is in one of four possible ordered states: Unallocated, Input Loaded, Processing and Output Ready. (Output Ready loops back to Unallocated.) Each block can only move forward from one state to the next in the list. No block can move backwards to a previous state or move forward skipping a state. Mutual exclusion is ensured in each of these states using two flags, F1 and F2.

Figure 6.1 on page 95 graphically displays the states of blocks of memory in shared memory. In each state both the HDL code (H/W) and C code (S/W) read both flags, so that read is not included in the diagram. Dark arrows indicate transitions that occur by

changing flag 2 (F2). Shaded arrows indicate transitions that occur by changing flag 1 (F1). The state of a particular block is determined by its first two bits which are used as flags. The remaining space in the block contains the Process id, a Unique Identifier and the Inputs and the Outputs.

F1 indicates that there is input ready for the FPGA to process. F1 indicates that there is output ready for the C code to reintergrate.

When the SHARP scheduler receives a request for a POINT to be executed on the FPGA, the request is queued and then processed in the order it arrives. If the queue is full, the request is denied. (Figure 6.2 on page 96 depicts the SHARP Scheduler as a Finite State Machine (FSM).) For each request in the queue, the scheduler finds a block in state (0,0)and immediately moves it to state (0,1) to indicate that the block is allocated. It is imperative that this operation be atomic since in state (0,0) the block does not belong to any particular process. In all other states there can only be one thread accessing a particular block.

Only when the initialization of the block is complete (inputs are loaded and the unique id is assigned) is the state of the block moved to (1,1). At this time the inputs can be used by the FPGA. Even if two threads are accessing the same POINT there will not be a conflict since it is reading from and writing to different blocks in the shared memory and the order the inputs are received will dictate the order the outputs are generated. When the outputs are written to the shared memory block the state is updated to (1,0).

The C code may have already requested the outputs, but the request is denied until the block is in state (1,0). Once a request is made for the outputs in this stage, they can be read and the state is moved to (0,0).

If the C code sends a request to abandon the output form a particular run of a POINT in hardware while the block is in state (1,1), the block is set to state (1,0) and if the block is in the state when the out is finished, the block is immediately moved to state (0,0). Thus there are the following times for reads and writes:

- State (0,0): Only flag F2 can be written in an atomic operation.
- State (0,1): Only one thread in the software code can read from or write to this block.
- State (1,1): Only one thread in HDL code can read from or write to this block, except that one software code thread can update flag F2 in a single atomic operation.
- State (1,0): Only one thread in the software code can read from or write to this block, unless that particular thread has already relinquished its claim on the block, in which case only one thread in the HDL code can write to this block.

In all cases, there are no race conditions where two threads can be accessing the same data at the same time and one of the accesses be a write.



Figure 6.1: Block states in shared memory.



Figure 6.2: Block states in shared memory.
7 Evaluation

As a writer of a thesis this is the most satisfying chapter. Here one can look back on the new ideas and creation and attempt a declaration of 'good' or 'not good. The first section contains notes of how to expand SHARP. This is where the ideas that could not be implemented in the limited scope of this thesis are recorded. The second section expands this by giving notes for future developers about the algorithms and underlying philosophies of SHARP.

Notable Quote:

Today you are You, that is truer than true. There is no one alive who is Youer than You.

- Dr. Seuss. Happy Birthday to You!

Be who you were meant to be through the very act of creation, and let your actions flow out of who you are.

- Rev. Gord Patterson

The third section looks at this research in terms of what it proves and demonstrates about the SHARP process (both positive and negative) and what it has left as an exercise to the reader to conclude. The fourth section states explicitly the research contributions this project has made to the codesign community.

The final section evaluates the SHARP process in the context of the wider body of research relative to the fourth dimension, time. It argues that SHARP is not merely good project, but that it is in tune with the current trend of research and that its impact today is greater now than any other time in the history of codesign.

7.1 Future Directions

The development of SHARP has opened a number of questions and avenues for related research. In order to maintain the coherence of this project and to focus the scope of the research, these avenues have been excluded until this point of the thesis. They will now be discussed.

SHARP's encapsulates the sub processes involved in hardware accelerating existing software. It does not give a perfect solution for all of the sub processes, but it is a great launch point for future development. Any of the sub-processes could be updated and fit back in to the system as a whole. For example, future developers could update SHARP to work with more programming languages or on different hardware platforms without having to understand the entire system.

One thing to note is that dual core processing is special case of SHARP that is actually much simpler than the example given in this thesis. Simply make the target FPGA the second processor. Since both processors are running the same hardware, no C-to-HDL translator is required. SHARP will automatically identify the SHARP POINTs and co-ordinate the communication between the two processors. Voila, you now have parallel processing for programs that were not designed to take advantage of dual core processors!

7.2 Notes to Future Developers

The complete SHARP project could easily be the focus of several careers, but to narrow the focus so that it was achievable in a single thesis many simplifications were made that are intended to be expanded latter. The initial project is complete in that it takes legacy code as input and produces codesigned code as output using a process that is modular, but many of the modules in the SHARP process can be optimized by future developers who are experts in the field of study covered by a particular module.

This section contains notes to future developers about the requirements and the subtleties of this development. This sections uses terms that are not defined since it is assumed that the experts in each filed will already be aware of the common vocabulary of their field.

7.2.1 Determining POINTs

In determining POINTs there are 2 main issues that have been left to future developers.

- 1. Points that are too big.
- 2. Differentiation of input and outputs.
- 3. Overlapping Points.

The first issue is relatively simple and could be solved in a strait forward manner. The issues is that the user defines a constraint so that POINTs can only contain a maximum number (MAX_POINT) of statements, but there could be more than MAX_POINT contiguous statements that are all SHARP Statements. The current solution is to take the first MAX_POINT statements and ignore the rest. Clearly this could be improved upon by making more than one POINT. So long as the multiple POINTs do not overlap in any way (from SP_LOAD to SP_RESYNC) the solution is strait forward. If the POINTS

overlap, then the simplistic solution will not work!

To solve the issue of overlapping POINTs and differentiation of inputs and outputs, the initial release of SHARP has a simplistic solution: it does not allow POINTs to overlap and all parameters of POINTS are considered both inputs and outputs. Making pure outputs would not be difficult, but there is a subtly with pure inputs. Pure inputs can be written between the SP_END and the SP_REYNC. If the POINT is then run in software, the inputs of the POINT need have the initial values they had at SP_LOAD, but after the SP_RESYNC they need to have the updated values.

Ideally the whole process of finding POINTs would be analyzed by an expert in removing the *impurities* from *Imperative Functions* making them *Functional*. The portion of each POINT that runs on hardware is *referentially transparent*: The side effects to the output are not applied until SP_RESYNC. This opens many areas for new optimizations such as:

- Identifying more pure statements that are not currently identified.
- Applying new techniques to unobtrusively re-arrange statements at compile time so that impure statements are filtered out and larger POINTs can be built.
- Applying the latest technology and ideas of functional programming to SHARP where applicable.
- Proving that the enhancements do not affect data integrity.

With all of these enhancements, future developers should keep in mind two very important rules. The first is that **the referential transparency of POINTs must be maintained**. Any enhancements that compromise this will severely limit development. The second is that the **enhancements must have a minimal impact on the existing code**. SHARP is designed to not interfere with future development of code.

7.3 What this research accomplishes and does not accomplish

This section is a brief critique of the research itself. Table 7.1 on page 101 gives a quick look at the major points this thesis has achieved and for each point that it does achieve it gives a corresponding limiting factor defining the scope of the achievement in terms or what was not achieved.

7.4 Research Contributions of SHARP

The SHARP process is an elegant tool that brings hardware acceleration out of the realm of specialized developers into the realm of mainstream software developers. Moreover, SHARP's biggest benefit is in the new areas of research that it opens up.

This research does	This research does not do
This research proves that data integrity is maintained across the platforms	This research does not prove that rounding errors cannot be introduced by the new hardware and acknowledges that they may be introduced.
This research proves that SHARP cannot introduce the possibility of deadlock.	This research acknowledges that if the possibility of deadlock already exists in a system but has not been apparent to the user, SHARP may change the timing so that the deadlock becomes apparent.
This research demonstrates that the SHARP process works from end to end.	This research does not even attempt to demonstrate that SHARP is the best process.
This research demonstrates that there is improvement in performance in many of test cases.	This research does not claim that SHARP produces the best performance increase.
This research demonstrates that the SHARP process is easy enough to follow that it can be said to approach automation.	This research leaves as an exercise to the reader to conclude that the SHARP process is easy enough that it can be said to approach automation.

Table 7.1: Evaluation of what this research does and does not do

SHARP is a completely modularized, end-to-end system for hardware acceleration. Using SHARP as a baseline, experts in each of the step in hardware acceleration are free to improve their particular part of the process without needing to understand the process as a whole, so long as they adhere to the interfaces SHARP provides.

SHARP also expands the number of people who can do research on hardware acceleration, since users can quickly and easily see the results of tweaking the constraints to see the impact it has on a system. This could lead to techniques that have not yet even been thought of.

The most satisfying research contribution that SHARP makes is that it answers the research question proposed in this thesis with a resounding "Yes". The process of hardware accelerating an existing piece of software can be simplified to the point that it approaches automation without limiting future development of the software.

7.5 The Timeliness of SHARP

The SHARP process enables the benefits of codesign to impact legacy software in a profound new way. No longer will the different paradigms, clock speeds and communication protocols between software and hardware hinder the potential beneficial interactions between software and reconfigurable hardware for SHARPened programs. No longer will hardware acceleration of an existing piece of software be dismissed as too expensive for SHARPenable programs. No more will hardware accelerated programs be too cumbersome to update.

SHARP allows an existing software application to simply be merged with a custom hardware/software run time environment, tested, fine tuned, and have its run time performance significantly improved.

SHARP is targeted for legacy applications, with the goal being to provide a significant performance boost on specific legacy applications by allowing selected portions to be diverted to run in hardware and seamlessly interact with the main program running in its traditional software environment. However there is no reason it should be limited by this design goal. While hardware design is generally less familiar to many software developers, there is no reason that general design cannot occur in the traditional software environment, then at the pre deployment phase, SHARP can be engaged to provide the simple hardware/software split and runtime performance boost. As the SHARP interface is targeted towards software application knowledge, little knowledge or training on the hardware side is required.

Since SHARP is such a great idea, why hasn't it been around for the last 20 years? After all, FPGA chips have been around since 1960^{-1} and in general distribution and usage since

¹First proposed by Gerald Estrin 6 in 1960, the RC is a "fixed plus variable structure" computer that can be "temporarily distorted into a problem-oriented special-purpose computer." The RC languished in relative obscurity for more than 30 years. [Estrin 1963]

the early 1980s². Actually the ideas behind SHARP probably have been thought about before, but without the recent advancements in technology the ideas were too difficult to implement.

The reason is the same as why we are just now getting live video on cell phones while we have had video images for over 75 years - TV, and Cell phones for over 30 years. However, 30 years ago the bandwidth available to a cell phone was not capable of transmitting live video in real time. As well the technology of the day would have required the cell phone to weigh 25 pounds and would need a battery suitable for a Cadillac to operate. The development of the hardware technology as well as the advancement in wireless transmission and data compression now make this possible and profitable. In simple terms the foundation technology was not at a maturity level that would support cell phone video. In the same way, SHARP has benefited from the increased interest in hardware acceleration as well as the lower cost and higher performance of hardware accelerators such as FPGAs. In 1985 Xilinx's first 3 chip - the XC2064TM - had a speed of only 50 MHz 4 Embedded software designers needed to ensure that the maximum benefit of each piece of software run in the FPGA was chosen and designed for the maximum benefit. With each successive generation (the Virtex family in 1998, the Spartan family in 2003 and the latest next-generation families of products just released in 2009) the technology has gotten faster and the storage capacity has gotten larger. Also the earlier chips could not be reprogrammed at run time the way that today's chips are.

With today's FPGAs SHARP's idea of producing a "good though not necessarily optimal" hardware acceleration will still accelerate many systems, and the benefits of a completely optimal solution can be assessed and discarded in favor of the simplicity of SHARP.

²Zycad Corporation, the first company to develop simulation accelerator technology, was founded in 1981. Zycad offers design verification, rapid prototyping, and test analysis solutions to companies developing highperformance, electronic systems. [Zycad 1996]

³http://www.xilinx.com/company/history.htm

⁴http://www.amiga-stuff.com/hardware/xc2064.html

A number of hardware and software requirements are necessary, and until recently they either did not exist, or were not developed to the technological point that SHARP could make use of them. There have been enormous gains in the capacity and performance of the programmable logic chips and it continues to grow. Until this was accomplished, there was not enough space within the FPGA chips to make the exercise worthwhile. The overhead of the context shift (moving the code in and out of hardware) and the hardware/software interaction would use up far more resources than the program would recover by the exercise. The program would most likely have run much slower except in specific circumstances. Indeed, it was these specific circumstances that often promoted the further development of the chip technology.

So now that we have reached the state that the software programming languages/compilers/linkers can support the interlinear code (running in both hardware and software seamlessly) and the hardware chipset has reached a capacity/performance/interface level to allow the same, what remains is the tool to encapsulate portions of the software and determine what portions are appropriate for the hardware shift and what portions are more appropriate to remain in software, as well as the tools to make it work. Thus SHARP is born.

Every technology has it's time. SHARP's time is now.

References

Codesign References

- [Bass 2002] BASS, M.J. AND CHRISTENSEN, C.M. 2002. The future of the microprocessor business. Spectrum, IEEE 39, 34-39.
- [Bertels 2012] Koen Bertels. Programming FPGAs. Retrieved June 3, 2012 Online: http://research.microsoft.com/en-us/um/redmond/events/ss2011/slides/friday/ koenbertels.pdf
- [Bindal 2005] An undergraduate system-on-chip (SoC) course for computer engineering students. Bindal, A.; Mann, S.; Ahmed, B.N.; Raimundo, L.A.; Education, IEEE Transactions on. Volume 48, Issue 2, May 2005 Page(s):279 - 289. Digital Object Identifier 10.1109/TE.2004.842911
- [Bishop 2003] William D. Bishop (2003). Configurable Computing for Mainstream Software Applications. Ph.D. dissertation, University of Waterloo (Canada).
- [Black 2010] SystemC: From the Ground Up. Schaumont, David C. Black, Jack Donovan, Bill Bunton and Anna Keist, US, 2010. doi: 10.1007/978-0-387-69958-5_2 URLhttp://dx.doi.org/10.1007/978-0-387-69958-5_2
- [Brown 1992] Brown, Stephen D., Francis, Robert J., Rose, Jonathan, and Vranesic, Zvonko G. Field-Programmable Gate Arrays, Kluwer Academic Publishers, 1992.
- [Compton 2002] COMPTON, K. AND HAUCK, S. 2002. Reconfigurable computing: a survey of systems and software. 34, 171-210. http://doi.acm.org/10.1145/508352.508353.

- [Dong-hyun 2009] A Survival Kit: Adaptive Hardware/Software Codesign Life-Cycle Model. Dong-hyun Lee; Keun Lee; Sooyong Park; Hinchey, M.; Computer. Volume 42, Issue 2, Feb. 2009 Page(s):100 - 102. Digital Object Identifier 10.1109/MC.2009.34
- [Estrin 1963] G. Estrin, B. Bussell, R. Turn, and J. Bibb. Parallel Processing in a Restructurable Computer System. IEEE Transactions on Electronic Computers, 12:747-755, December 1963.
- [Freeman 1996] Roger L. Freeman. Telecommunication system engineering, New York : John Wiley, c1996.
- [Garcia-Molina 2008] Database Systems: The Complete Book. Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer D. Widom Prentice Hall, 2nd Edition.
- [Geguang 2006] Integrating timed automata into tabu algorithm for HW-SW partitioning. Geguang Pu; Zhang Chong; Zongyan Qiu; Jifeng He; Wang Yi; Engineering of Complex Computer Systems, 2006. ICECCS 2006. 11th IEEE International Conference on. 0-0 0 Page(s):8 pp.. Digital Object Identifier 10.1109/ICECCS.2006.1690362
- [Gerstlauer 1970] System design :a practical guide with SpecC. Andreas Gerstlauer et al. Kluwer Academic Publishers.
- [Grattan 2002] GRATTAN, B., STITT, G. AND VAHID, F. 2002. Codesign-extended applications. In CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign, Estes Park, Colorado, Anonymous ACM Press, New York, NY, USA, 1-6.
- [Heldman 1992] Robert K. Heldman. Global telecommunications : layered networks' layered services, New York : McGraw-Hill, c1992.

- [Hyunuk 2002] Efficient hardware controller synthesis for synchronous dataflow graph in system level design. Hyunuk Jung; Kangnyoung Lee; Soonhoi Ha. Page(s): 423- 428. Digital Object Identifier 10.1109/TVLSI.2002.807765
- [Jerraya 2005] Hardware/software interface codesign for embedded systems. Jerraya, A.A.; Wolf, W. Computer. Volume: 38 Issue: 2 Feb. 2005. Page(s): 63- 69. Digital Object Identifier 10.1109/MC.2005.61
- [Kamat 2009] Unleash the System On Chip using FPGAs and Handel C. Rajanish K. Kamat et al. Springer Netherlands.
- [Katz 1994] KATZ, R.H. 1994. Contemporary Logic Design. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California.
- [Kent 2003] Kent, Kenneth Blair (2003). The co-design of virtual machines using reconfigurable hardware. Ph.D. dissertation, University of Victoria (Canada), Canada. Retrieved February 11, 2011, from Dissertations & Theses @ University of Victoria.(Publication No. AAT NQ85195).
- [La Rosa 2005] Software development for high-performance, reconfigurable, embedded multimedia systems. La Rosa, A.; Lavagno, L.; Passerone, C.; Design & Test of Computers, IEEE. Volume 22, Issue 1, Jan 2005 Page(s):28 - 38. Digital Object Identifier 10.1109/MDT.2005.20
- [Lee 2005] Hardware/software partitioning of operating systems: focus on deadlock detection and avoidance. Lee, J.J.; Mooney, V.J., III; Computers and Digital Techniques, IEE Proceedings - Volume 152, Issue 2, Mar 2005 Page(s):167 - 182. Digital Object Identifier 10.1049/ip-cdt:20045078

- [Moore 1965] Moores law. (2010). In Encyclopdia Britannica. Retrieved August 13, 2010, from Encyclopdia Britannica Online: http://www.britannica.com/EBchecked/topic/705881/Moores-law
- [Moser 2008] Moser, Raimund; Abrahamsson, Pekka; Pedrycz, Witold; Sillitti, Alberto; Succi, Giancarlo. A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team. Balancing Agility and Formalism in Software Engineering, ISSN 0302-9743, 2008, Lecture Notes in Computer Science, ISBN 9783540852780, Volume 5082, pp. 252 - 266
- [Moss 2007] Seamless Hardware/Software Performance Co-Monitoring in a Codesign Simulation Environment with RTOS Support. Moss, L.; de Nanclas, M.; Filion, L.; Fontaine, S.; Bois, G.; Aboulhamid, M.; Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07. 16-20 April 2007 Page(s):1 - 6. Digital Object Identifier 10.1109/DATE.2007.364403
- [Mudry 2006] A hybrid genetic algorithm for constrained hardware-software partitioning. Mudry, P.-A.; Zufferey, G.; Tempesti, G.; Design and Diagnostics of Electronic Circuits and systems, 2006 IEEE. 0-0 0 Page(s):1 - 6. Digital Object Identifier 10.1109/DDECS.2006.1649561
- [Noguera 2002] HW/SW codesign techniques for dynamically reconfigurable architectures.
 Noguera, J.; Badia, R.M. Very Large Scale Integration (VLSI) Systems, IEEE
 Transactions on. Volume: 10 Issue: 4 Aug 2002. Page(s): 399- 415. Digital Object
 Identifier 10.1109/TVLSI.2002.801575
- [Olson 2007] Hardware/Software Partitioning Using Bayesian Belief Networks. Olson, J.T.; Rozenblit, J.W.; Talarico, C.; Jacak, W.; Systems, Man and Cybernetics, Part A, IEEE Transactions on. Volume 37, Issue 5, Sept. 2007 Page(s):655 - 668. Digital Object Identifier 10.1109/TSMCA.2007.902623

- [Platzner 2000] PLATZNER, M. 2000. Reconfigurable accelerators for combinatorial problems. Computer 33, 58-60.
- [Qingyang 2008] A practical HW/SW codesign method for system-level embedded system. Qingyang Meng; Jianzhong Qiao; Jun Liu; Benhai Zhou; Intelligent Control and Automation, 2008. WCICA 2008. 7th World Congress on. 25-27 June 2008 Page(s):7697 - 7701. Digital Object Identifier 10.1109/WCICA.2008.4594126
- [Quinn 2004] Parallel Programming in C with MPI and OpenMP First edition. Michael J. Quinn, McGraw-Hill 2004. Page(s):1-2.
- [Samek 02] Samek, Miro, Practical Statecharts in C/C++: Quantum Programming for Embedded Systems, CMP Books, 2002, ISBN: 1-57820-110-1.
- [Sangeetha 2007] Optimization of Behavioral Modeling for Codesign of Embedded System. Sangeetha, M.; RajaPaul Perinbam, J.; Signal Processing, Communications and Networking, 2007. ICSCN '07. International Conference on. 22-24 Feb. 2007 Page(s):414 - 419. Digital Object Identifier 10.1109/ICSCN.2007.350773
- [Santambrogio 2006] Partial Dynamic Reconfiguration: The Caronte Approach. A New Degree of Freedom in the HW/SW Codesign. Santambrogio, M.D.; Sciuto, D.; Field Programmable Logic and Applications, 2006. FPL '06. International Conference on. 28-30 Aug. 2006 Page(s):1 - 2. Digital Object Identifier 10.1109/FPL.2006.311355
- [Schaumont 2008] A Senior-Level Course in Hardware-Software Codesign. Schaumont, P.; Education, IEEE Transactions on. Volume 51, Issue 3, Aug. 2008 Page(s):306 - 311. Digital Object Identifier 10.1109/TE.2007.910434
- [Schaumont 2010] A Practical Introduction to Hardware/Software Codesign. Schaumont, P.; Springer US, 2010. doi: 10.1007/978-1-4419-6000-9_1 URL: http://dx.doi.org/10.1007/978-1-4419-6000-9_1

- [Shannon 2004] Shannon, L. and Chow, P. 2004. Using reconfigurability to achieve real-time profiling for hardware/software codesign. In Proceedings of the 2004 ACM/SIGDA 12th international Symposium on Field Programmable Gate Arrays (Monterey, California, USA, February 22 - 24, 2004). FPGA '04. ACM, New York, NY, 190-199. DOI= http://doi.acm.org/10.1145/968280.968308
- [Silberschatz 2009] Operating system concepts. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. Hoboken, NJ : J. Wiley and Sons, c2009.
- [Vahid 2002] VAHID, F. AND GIVARGIS, T.D. 2002. Embedded System Design: A Unified Hardware/Software Introduction. John Wiley and Sons ; ISBN: 0471386782. Copyright (c) 2002.
- [Vahid 2007] Vahid, F.; , "It's Time to Stop Calling Circuits "Hardware", Computer , vol.40, no.9, pp.106-108, Sept. 2007 doi: 10.1109/MC.2007.322 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4302628&isnumber=4302594
- [Wiangtong 2005] Hardware/software codesign: a systematic approach targeting data-intensive applications. Wiangtong, T.; Cheung, P.Y.K.; Luk, W. Signal Processing Magazine, IEEE. Volume: 22 Issue: 3 May 2005. Page(s): 14- 22
- [Wolf 2001] WOLF, W. 2001. Computers as Components: Principles of Embedded Computing System Design. Morgan Kaufman Publishers, 2001.
- [Wolf 2002] W. Wolf. What is embedded computing? IEEE Computer, 35(1):136-137, January 2002.
- [Wolf 2003] Wolf, W.; , "A decade of hardware/software codesign," Computer , vol.36, no.4, pp. 38- 43, April 2003 doi: 10.1109/MC.2003.1193227 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1193227&isnumber=26760

[Zycad 1996] Press Release. (1996). Retrieved June 3, 2012, from: http://www.ryt9.com/es/anpi/62

Monte Carlo References

[Evans 1995] EVANS, R.D. 1955. The atomic nucleus. McGraw-Hill, New York.

- [Kawrakow 2000] KAWRAKOW, I. 2000. Accurate condensed history Monte Carlo simulation of electron transport. I. EGSnrc, the new EGS4 version. Medical Physics 27, 485–498.
- [Kawrakow 2000 B] KAWRAKOW, I. AND ROGERS, D.W.O. 2000. The EGSnrc Code System: Monte Carlo simulation of electron and photon transport, Technical Report PIRS-701. National Research Council of Canada, Ottawa, Canada.
- [Rogers 2002] ROGERS, D.W.O. 2002. Monte Carlo Techniques in Radiotherapy. Medical Physics Special Issue 58#2, 63–72.
- [Sempau 2000] SEMPAU, J., WILDERMAN, S.J. AND BIELAJEW, A.F. 2000. DPM, a fast, accurate Monte Carlo code optimized for photon and electron radiotherapy treatment planning dose calculations. Phys. Med. Biol. 45, 2263–2291.

A Monte Carlo Algorithm for Radiotherapy Simulation

A.1 What is

radiotherapy?

Also known as radiation therapy, radiotherapy is a method of using ionizing radiation to treat cancer. This treatment method deliberately deposits ionizing radiation into the cells with which it comes in contact. This injures healthy cells as well as cancerous cells and might even cause healthy cells to become cancerous. Notable Quote: Growth for the sake of growth is the ideology of the cancer cell. - Edward Abbey Healing of soul and spirit comes not with the removal of PAIN but with the RESTORATION of PURPOSE!

- Rev. Dr. C. Rod Hembree

However, healthy cells are more likely to recover from radiation than cancerous cells. In radiotherapy, cancer cells are damaged to the point of being unable to reproduce, while healthy cells are usually only temporarily damaged. Even though the healthy cells do tend to recover, the radiation process has very unpleasant side effects and should therefore be minimized as much as possible. Dose simulators allow the doctor to set up a precise treatment that will maximize the amount of radiation delivered to the tumor, while minimizing the amount of dose delivered to the surrounding organs.

Radiotherapy has two general forms: internal or external. In internal radiotherapy the radioactive particles are injected directly into the target (brachytherapy), or ingested by the patient in a form that will collect in the target (such as ionized iodine which collects in

the thyroid). The case study in this proposal involves external radiotherapy. In external radiotherapy the dose is introduced using an external beam. In a good radiotherapy plan, there is a relatively high dose being delivered to the tumor while a relatively low dose is delivered to the surrounding organs at risk (OAR). Using an external beam requires a high level of precision. Before the doctor applies a treatment to a patient she must determine the ideal setup for the beams using a dose simulator. A dose simulator can be thought of as a virtual reality video game that has a 2 or 3 dimensional virtual copy of the patient (called the phantom) and the radiation beams. The organs that are blocking doctor's view of the tumor (for example the skin) can be turned off so the doctor can evaluate the dose cloud in the tumor and OARs. The dose cloud is a 3D representation of where the dose is calculated to be a determined value or threshold. Inside the cloud the dose is higher than the threshold. Outside the cloud, the dose is lower than the threshold. If the doctor determines that the proposed plan needs to be improved before it is implemented, he can adjust the beam settings and re-run the simulation.

A.2 Publicly available radiotherapy simulation code

Before installing the BEAM code, you need to have the EGSnrcMP code downloaded and installed from the following URL:

http://www.irs.inms.nrc.ca/EGSnrc/EGSnrc.html.

This creates 2 directories, HEN_HOUSE (containing the source code and libraries) and EGSnrc_mp (containing input and output files as the executables).

The BEAM code uses the libraries downloaded in the previous paragraph. It is available for download at the URL:

http://www.irs.inms.nrc.ca/BEAM/user_manuals/DISTRIBUTION.html.

This includes the BeamNRC code (which calculates the particle transport to the MLCs)

and DosXYZNRC which is the primary computing engine for the Monte Carlo dose calculations. Figure A.1 on page 115 gives a graphic representation of the different elements of the code.



Figure A.1: Code layout

The DosXYZNRC code had recently been eclipsed by the VMC++ code which does the same calculation in less than $1/5^{th}$ the time. VMC++ is not currently available for public download but a copy of the executable can be obtained by sending an e-mail describing your research interests to Iwan Kawrakow (iwan@irs.phy.nrc.ca).

A.3 How is the dose calculated?

Dose calculation is done by tracing the path of several million simulated radioactive particles as they travel through the virtual patient and calculating the dose that each particle deposits as it travels. The complete dose calculation includes two distinct parts: The Pencil Beam algorithm that predicts the new path of a particle as it crosses the barrier from one medium to another, and the Monte Carlo algorithm used to predict the path of radioactive particles as they travel within a medium.

The initial simulated particles are placed in the queue. As each particle is removed from

the queue a random value is chosen for the distance to the next interaction. If this distance crosses a medium barrier, then the particle is moved to the intersection of its path and the medium barrier, the Pencil Beam algorithm is used to determine its new trajectory, and the particle is placed back in the queue. If the distance does not cross a medium barrier then the particle is moved that distance and a Monte Carlo interaction is chosen at random to be applied to it. (The possible interactions are discussed in the following sections) Depending on the interaction, this might result in new particles being added to the queue. As particles drop below a threshold of energy level, or leave the area of interest, they are also removed from the queue. (See Figure A.2 on page 120)

A.4 Possible Monte Carlo Interactions

The possible interactions depend on the energy level of the photon. The energy levels are classified as high (>1000 KeV), intermediate (10-1000 KeV) and low (<10 KeV). [Rogers 2002]

High: Interactions can occur with the nucleus:

- **Compton scattering:** the photon changes direction and loses energy to a free or outer shell electron which is set in motion.
- Pair production: (E>1.022MeV) the photon disappears and an electron and positron share its energy.
- **Photodisintegration:** (E>10MeV) the photon disappears and an electron and positron share its energy.

Intermediate: Interactions can occur with an electron cloud.

• **Photoelectric effect:** the photon ejects an inner shell electron and is completely absorbed.

- **Compton scattering:** the photon changes direction and loses energy to a free or outer shell electron which is set in motion.
- Thomson scattering: the photon is redirected by a free electron. No energy is lost.

Low: Interactions can occur with an atom or a molecule.

- Raman scattering: the photon interacts with an atom or molecule. It loses energy and changes direction.
- **Rayleigh scattering:** the photon interacts with a whole atom or molecule or any particle smaller than the wavelength of light. It does not lose any energy, but it changes direction.

For cancer treatment, the intermediate interactions are the most interesting since they are high enough energy to affect the cancer without seriously harming the patient.

A.5 Interactions by type

The possible interactions can also be classified according to their types: inelastic, elastic or absorption. The following tables list the common interactions by their types.

Name	Nobel Prize	Interaction occurs with:
Compton scattering	1927	Free or outer shell electrons
Raman scattering	1930	Atoms/molecules

Inelastic (energy loss, change of direction)

Absorption (deposit all energy)

Name	Nobel Prize	Interaction occurs with:		
rotation modes,				
vibration modes,				
level transitions				
photoelectric effect	Einstein	The photon ejects an inner shell electron and		
	1921	gets completely absorbed		
pair production		High energy photon E>1.022 MeV Photon interacts with nuclear/electric field to produce electron-positron pair. Positron loses energy and annihilates at rest with another electron, producing 2 photons with energy E=0.511 MeV		
photodisintegration		High Energy Photons E>10 MeV. Photon is absorbed by the nucleus. Neutron and photon emitted		
Elastic (no energy loss, change of direction)				

Name	Nobel Prize	Interaction occurs with:
Rayleigh scattering	1904	Whole atom/molecule or any particle smaller than the wavelength of light
Thomson scattering	1906	Free electron

Knowing all the possible interactions for a particular particle with a specific energy level, and the probability of each interaction, the system can then calculate a randomly determined path for the particle and all the particles set in motion by the initial particle. Figure refunique:monteExample on page 121 shows the path of a high energy photon using this method.

A.6 Depositing radiation on the particle's path

Now that the path of each particle is defined the dose can be calculated. To calculate the dose the amount of energy the photon or electron loses as it travels through a unit of a particular medium is calculated based on the stopping power of the medium. The stopping power of a photon within a medium is the sum of the stopping power caused by collisions with other particles following Coulomb's law¹ (Coulomb collisions) and the energy lost as the surrounding atoms are elastically displaced as the photon passes by. This value can be calculated as a ratio of the chemical makeup of the material and the density of the material. For accurate calculations of a given medium, use the calculator provided by the National Institute for Standards and Technology (NIST) calculator provided at

http://physics.nist.gov/PhysRefData/Star/Text/method.html

A.7 Calculating the Radiation of a Beam

Since the radiation deposited by a single radioactive particle can be calculated, the procedure can be repeated for each particle in a beam. An average beam might trace the path (or 'history') of 10,000.000 particles for a single beam, and each treatment might involve several such beams irradiating the tumor from different angles. To make this massive calculation possible, the Royal Jubilee Hospital Cancer Clinic in the case study uses is a single main computer to coordinate the calculations, and 12 sub-computers processing a batch of particles at a time. Ideally, already at this very high level, each sub-computer would have a dedicated FPGA.

¹The magnitude of the electrostatic force between two point charges is directly proportional to the product of the magnitudes of each charge and inversely proportional to the square of the distance between the charges.



Figure A.2: Flow chart of particle ionizing calculations



Figure A.3: The path of a high energy photon

A 10 MeV photon (dotted line) is incident from the right on a slab of lead. A pair production event occurs and produces an electron (solid line) and a positron (long dashed line). The electron scatters many times and loses energy to low energy particles. The electron gives off a bremsstrahlung photon near the bottom of the figure. The positron also scatters many times and loses energy and then annihilates in flight (if it were at rest, the two 511 keV photons would be given off at 180 degrees to each other). The 511-keV photon coming back towards the surface Compton scatters, creating another electron and scattering, apparently at 90 degrees. The figure is an EGS Windows output from an EGS4 simulation. [Rogers, D.W.O. 2002]

B FPGA

A field-programmable gate array (FPGA) is an integrated circuit designed to be configured and reconfigured by the end user after manufacturing: thus the name *field-programmable*. Notable Quote:

Any sufficiently advanced technology is indistinguishable from magic.

- Arthur C. Clarke

As shown in Figure B.1 an FPGA is made up of an array of logic blocks that can be connected via interconnect resources. Each logic block of the FPGA is capable of implementing a 4-input function (through the use of a look-up table), or acting as a small register.

The interconnect is contains programmable switches that connect the logic blocks to each other. The I/O cells connect the FPGA to an external device for communication. To implement a logic circuit in the FPGA, first partition the logic circuit into smaller components, such that each piece can be implemented by a single logic block. Then place each of the smaller logic circuits within logic blocks on the FPGA architecture in such a way that they can later be connected to reassemble the original logic circuit. Each of the components within the FPGA (logic blocks, the interconnect and I/O cells) are re-programmable. This allows the user to repeat the process of creating circuits and mapping them to the FPGA. This configuration by the end user is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC).



Figure B.1: A conceptual field programmable gate array (FPGA).