A NEW LOOK AT
RETARGETABLE COMPILERS

Patrick William Burke

Dissertation Prepared for the Degree of
DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS
December 2014

APPROVED:

Philip H. Sweany, Major Professor
Mahadevan Gomathisankaran,
   Committee Member
Gayatri Mehta, Committee Member
Armin Mikler, Committee Member
Bill Buckles, Director of Graduate Studies in
   the Department of Computer Science
   and Engineering
Barrett Bryant, Chair of the Department of
   Computer Science and Engineering
Costas Tsatsoulis, Dean of the College of
   Engineering
Mark Wardell, Dean of the Robert B.
   Toulouse School of Graduate Studies
ACKNOWLEDGMENTS

I never could have made this happen without the constant support and encouragement of my wife, Denise. For that, and so much more, I owe her my gratitude and offer my most sincere thank you. I also want to apologize to my children, Ryan and Kaitlin, and my grandson, Korbin, for the impact that this endeavor had on my ability to share time with them over the last decade. You are my legacy and all the hope that I have in this world. Let the good times roll.

I would also like to thank the members of my Ph.D. committee whose tireless support and constructive criticism made this possible. First and foremost, my advisor, Dr. Philip Sweany, was instrumental in recruiting me to take on this endeavor and he has stuck by me every step of the way. Dr. Armin Mikler became my counselor and encouraged me to push through some difficult times early in my Ph.D. studies. Without these two men, I doubtless would have given up before the process ever got fully started. Dr. Gayatri Mehta and Dr. Mahadevan Gomathisankaran provided valuable feedback and hints, especially in the dark world of computer hardware.

Finally, I thank the support staff in the Computer Science and Engineering Department. Rarely did a week go by that I did not have to visit with Sally Pettyjohn, Stephanie Deacon, Genene Murphy, and/or Jaymi Wenzel to handle one crisis or another. You are the glue that keeps this Department running.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgments</td>
<td>iii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
<tr>
<td>Chapter 1: Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1. History</td>
<td>5</td>
</tr>
<tr>
<td>1.1.1. Software Engineering History</td>
<td>6</td>
</tr>
<tr>
<td>1.1.1.1 Soviet Nuclear Missile Attack</td>
<td>6</td>
</tr>
<tr>
<td>1.1.1.2 Patriot Anti-Missile System</td>
<td>7</td>
</tr>
<tr>
<td>1.1.1.3 Year 2000 (Y2K)</td>
<td>8</td>
</tr>
<tr>
<td>1.1.1.4 Expeditionary Combat Support System</td>
<td>9</td>
</tr>
<tr>
<td>1.1.1.5 F-35 Program Performance</td>
<td>10</td>
</tr>
<tr>
<td>1.1.2. Processor History</td>
<td>12</td>
</tr>
<tr>
<td>1.2. Heterogeneous Processors on a Chip</td>
<td>13</td>
</tr>
<tr>
<td>1.2.1. Hy-C Solution</td>
<td>15</td>
</tr>
<tr>
<td>1.3. Retargetable Compiler Simplification</td>
<td>16</td>
</tr>
<tr>
<td>1.4. Overview</td>
<td>17</td>
</tr>
<tr>
<td>Chapter 2: Product Development History</td>
<td>18</td>
</tr>
<tr>
<td>2.1. Software Development Historical Overview</td>
<td>19</td>
</tr>
<tr>
<td>2.2. DOD-STD-2167</td>
<td>24</td>
</tr>
<tr>
<td>2.3. DOD-STD-2167A</td>
<td>33</td>
</tr>
<tr>
<td>2.4. MIL-STD-498 Software Development and Documentation</td>
<td>37</td>
</tr>
</tbody>
</table>
CHAPTER 3 RETARGETABLE COMPILERS 52
3.1. History 52
3.2. Examples 55
    3.2.1. GNU GCC 56
    3.2.2. Production-Quality-Compiler Compiler 59
    3.2.3. LISATek 60
    3.2.4. Horizon 62
3.3. Hy-C 64
    3.3.1. System Specification 64
    3.3.2. Partitioning 66
    3.3.3. CPU Compiler 67
    3.3.4. FPGA Code 67
    3.3.5. ASIC Code 69
    3.3.6. Power/Performance Modules 69
    3.3.7. Optimization Control 70
3.4. Hardware/Software CoDesign 70

CHAPTER 4 ARCHITECTURE DESCRIPTION ESSENTIAL ELEMENTS 72

CHAPTER 5 Hy-C ARCHITECTURE DESCRIPTION 76
5.1. General Description 78
5.2. Texas Instruments TMS320C64x DSP 80
5.3. Instruction Set Description 82
5.4. Memory Description 87
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2.1</td>
<td>DOD-STD-2167/2167A Required Documents</td>
<td>26</td>
</tr>
<tr>
<td>Table 2.2</td>
<td>CMMI Maturity Levels</td>
<td>40</td>
</tr>
<tr>
<td>Table 2.3</td>
<td>CMMI Process Areas</td>
<td>41</td>
</tr>
<tr>
<td>Table 3.1</td>
<td>Matrix Multiply Comparison</td>
<td>69</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>Example TMS320C64x DSP Functional Unit Descriptions</td>
<td>81</td>
</tr>
<tr>
<td>Table 5.2</td>
<td>Hy-C Description of Computing Resources</td>
<td>84</td>
</tr>
<tr>
<td>Table 5.3</td>
<td>Example Hy-C Operator Description for DSP</td>
<td>85</td>
</tr>
<tr>
<td>Table 5.4</td>
<td>Memory Architecture Description</td>
<td>87</td>
</tr>
<tr>
<td>Table 5.5</td>
<td>FPGS Architecture Description</td>
<td>89</td>
</tr>
<tr>
<td>A.1</td>
<td>Example Hy-C Architecture Description</td>
<td>102</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Heterogeneous Processor Component Comparison</td>
<td>13</td>
</tr>
<tr>
<td>Figure 1.2</td>
<td>Hy-C Functional Diagram</td>
<td>15</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Chronology of DOD Software Development Methods</td>
<td>21</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>DOD-STD-2167 Product Development Plan</td>
<td>25</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>DOD-STD-2167A Product Development Plan</td>
<td>34</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>System Development Cycle</td>
<td>37</td>
</tr>
<tr>
<td>Figure 2.5</td>
<td>Agile Development Cycle</td>
<td>45</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Typical compiler structure</td>
<td>53</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Example GCC Instruction Machine Description (subsf3)</td>
<td>56</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Example GCC Instruction Templates</td>
<td>58</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>LISA Example Description for Language Expr</td>
<td>62</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Generic Hybrid Architecture</td>
<td>63</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

Consumers in this increasingly interconnected world constantly demand newer, faster, and more powerful personal computing devices with a price well within the grasp of the average user. Moreover, the consumers have come to expect new hand-held technology improvements every couple of years, coinciding nicely with cellular telephone service agreement commitments. Yet, two-year development cycles are hard enough to orchestrate for either hardware or software, but the concurrent development of both for new product lines within that cycle time seems almost impossible. Necessity being the mother of invention, the electronics industry has, so far, responded accordingly. As future systems become even more complicated, however, maintaining this two year cycle with the current tools will become commensurately harder. My research for this dissertation delves into the feasibility of simplifying the development of a such a system by employing the use of heterogeneous System on a Chip (SoC) composed of existing processor components and using a specialized retargetable compiler to extract increased parallel processing, thus increasing the overall performance of the SoC. This is accomplished through the synergistic distribution of execution among the computing resources via a retargetable compiler, named the Hybrid Computer Retargetable Compiler (Hy-C). Retargetable compilers have been under study since the 1970s, but industry attitudes toward the resultant products has been less than one of universal acceptance. Many practitioners consider existing retargetable compilers as useless, in a practical sense, due to their inherent complexity and the high cost of expert staffing to deal with these complexities. Hy-C is designed to be simple, but powerful. While all aspects of this proposed system will be introduced, the main contribution of this dissertation is that it shows that simple retargetable compilers are a viable option for a significant, and growing, subset of new computer architectures, namely, that of heterogeneous multi-processors on a chip, that include existing proven
processors for which compilers already exist. The dissertation also provides an example of an architecture description adequate to support the initial operation of a simplified retargetable compiler, like Hy-C. That architecture description is only a first cut, but it allows the system to execute as quickly as possible to generate empirical data that can be used to validate or modify the architecture description in accordance with the data. The architecture description design is intentionally easily adaptable to fit this paradigm. A secondary contribution is a discussion of software engineering paradigms used in the military industrial complex that identifies possible sources of software errors in the implementation of those paradigms.

The commercial viability of hand-held devices often depends on such non-technical parameters as size, color, and battery life. The creation of a commercially successful digital device generally requires the orchestration of compromises between price, consumer desires, and the availability of cutting-edge technology. For example, state-of-the-art processor chips have a finite size that will, to some extent, dictate the minimum size of a cellular telephone. Likewise, the battery required to power that processor has a minimum size specification. It stands to reason that a larger battery can last longer, but the designers must limit the size of the battery to fit in a hand-held consumer device. Of course, the phone could be used for days or weeks at a time without the need for recharging the battery, if the battery could be the size of a car battery, but few, if any, consumers would find this a desirable solution. It is also worthy to note that processors, memory chips, and batteries all have heat-dissipation requirements that impose further limits on the required dimensions of the physical device case. Once these physical trade-offs have been made and the final hardware has been selected, the software must then operate as efficiently as possible on the selected platform.

A significant amount of software efficiency can be achieved with traditional compiler optimization techniques. Extracting parallel processing can also greatly improve the execution performance of software. Pipelining instructions provides micro-level parallel execution. Multi-tasking or multi-threading on a multicore processor, on the other hand, allows for full
instruction-level parallelization. True parallel processing requires multiple processors or a multiple core processor that can simultaneously execute instructions that are data independent. My experience in industry afforded me the experience of dealing with broken vendor promises with respect to multicore processors. Marketing personnel from one popular processor contractor touted an eight-fold increase in processing capability over that of the legacy single-core processor with the incorporation of their new eight core multiprocessor. The decision was made to change to that processor for the next planned upgrade. After the switch, unfortunately, the program never realized the promised execution increase and was lucky when execution managed to yield even a two-fold increase. Thus, parallelization became a manual task assigned to the system engineers and software architects to build into the architecture and design of the development system itself, via hardware, software, and/or firmware. Any and all efficiency created in this manual mode was at risk of being insidiously lost with each and every change in the software. Hardware, once manufactured, could not change unless the program and the customer were prepared to absorb an excessive cost and schedule overrun. Any and all design errors had to be changed in software, driving its cost far beyond the initial estimate.

Multicore processing became a topic of much research and, in general, contention for memory access turned out to be the limiting resource[56]. The research also indicated that the performance of the Message Passing Interface (MPI) was degraded by the addition of multiple cores. The level of degraded performance was not constant and was shown to depend upon the characteristics of the application. In general, however, performance degradation showed a strong correlation with the main memory bandwidth. Some level of mitigation was possible through careful design and optimization, but these techniques were, and still are, complicated. Obviously, an automated compiler-driven solution is the preferred method of achieving a fully parallelized solution in software and a software tool named the Hy-C is being developed at the University of North Texas just for this purpose.
The Hy-C retargetable compiler is designed to target not only multiple processing cores, but a heterogeneous set of processors that might include a mix of general purpose central processing units (CPUs), graphics processing units (GPUs), field-programmable gate arrays (FPGAs), digital signal processors (DSPs), and application specific integrated circuits (ASICs)[30]. One unique characteristic of Hy-C that sets it apart from other retargetable compilers is that all of the included processors must be commercially available with associated dedicated compilers. This characteristic allows Hy-C to be implemented with very simple input files in a very short time. Given the 2-year upgrade cycle mentioned earlier, the current trend of using existing processors will become even more dominant in the future. These processors each have unique processing, energy consumption, and thermal characteristics[32][41]. The Hy-C compiler partitions the input source code to the computational devices that will provide the optimal performance in terms of processing throughput and energy consumption. In doing so, it achieves synergistic results which will yield better execution performance and lower energy consumption than the same code executed on a homogeneous processor set. Hy-C could be used, in time, as an engineering tool for hardware/software codesign, which is one component of the Design Space Exploration (DSE) development phase for any program employing new hardware and/or software. The hardware/software codesign concept has different definitions for different development organizations, but has a common goal of establishing the division of required processing between software and selected hardware early enough in the development cycle to avoid rework of hardware in the later phases of development. It is a means of “meeting system-level objectives by exploiting the synergism of hardware and software through their concurrent design[33]).” Ideally, the role of the software components should also be fixed at the end of this phase, but experience has taught me that unplanned modifications to software are sometimes required in the later stages of development to compensate for insufficient hardware, and to be fair, software specifications.

While the discussion, so far, has centered on the domain of hand-held mobile devices,
it is interesting to note that the requirements for those devices are nearly identical with the requirements for current United States Department of Defense (DOD) programs. The DOD strongly desires weapon systems that are light, durable, man-portable, multi-function, error-free, and affordable. In addition, short development schedules are highly attractive since they allow the systems to be fielded and in the hands of the soldiers and make a difference on the battlefield in minimal time. As such, any improvements made for the benefit of the mobile device market will have an immediate positive affect on the critical, and financially lucrative, government contractor industry. I consider both of these to be niche markets, but they are large and important markets, so even small improvements can yield massive savings. Furthermore, according to Markets and Markets, a Texas-based research and consulting firm, the market for heterogeneous processors is going to continue to grow at a rate greater than 20% through at least 2020[10]. This growth rate surely will support the development and growth of new applications and tools for this burgeoning industry.

1.1. History

A discussion of the history behind this paper requires the discussion of two separate areas of study. The first, the history of the success of the software engineering industry, the track record, is necessary to determine the motivation that drives the need for a hybrid computing solution or any other improvements. If current software development paradigms produce sufficiently efficient and error-free software, it stands to reason that this rather drastic proposal would not be necessary. Perhaps, instead, it is just part of the continuing improvement process that follows all products.

The second area of study, processor history, addresses the question of why we cannot simply develop more efficient and powerful processors, just as we have since 1958 when Jack Kilby developed the first integrated circuit at Texas Instruments[4]. If the electronic industry can follow its time tested methods to develop new single processor chips that are more powerful and faster than ever before, why should the industry risk new technology like
heterogeneous processors on a chip? If something is not broken, why fix it?

1.1.1. Software Engineering History

Despite 50+ years of development process and tool improvements, the software engineering industry is still plagued by failure after failure. A simple Google search for “software failures” will produce page after page of examples of truly epic system failures due to software errors. These are not bugs. The use of that term trivializes the grave nature of some of the errors. Errors are errors! Despite all of the existing software development tools and processes, errors are still occurring. The modern software systems are simply too complex for manual handling anymore. Automation plays the critical role in providing software engineers with proper tools to efficiently handle the complexity[3][23].

Hardware engineers had the same problem in the 1960s with the advent of integrated circuits when hardware complexity grew beyond the reasonable grasp of the average or better engineer. That industry responded to the challenge with the introduction of Computer Aided Design (CAD) systems which simplified the task of creating increasingly complex hardware systems while “reducing errors and speeding design time”[1]. With the use of CAD systems, hardware engineers are now able to pack billions of transistors on a single chip. Software is still waiting for such simplification, automation, or automation. Until that happens, increased fidelity of processes will have to suffice to control errors, but it is not working well. A few of the most notable cases of software failures are described below. Each of them had a very simple solution that should have been found to correct the embedded errors long before any systems were fielded.

1.1.1.1. Soviet Nuclear Missile Attack

On September 23, 1983, a software error in a Soviet early warning system in an orbiting satellite reported five incoming nuclear missiles launched from the United States[37]. The standing order for the Soviet military was to immediately launch a counterattack with their own nuclear missiles in retaliation. This policy was known as Mutually Assured Destruction
(MAD) and was adopted specifically to prevent either side from initiating an attack for fear of the dire consequences of the assured counterattack. The commander of that Soviet missile detachment was Lt. Col. Stanislav Petrov. Despite his orders to immediately launch a retaliatory strike, he reasoned that if the United States had really intended to launch a missile attack against the Soviet Union, it surely would have used more than five missiles. Therefore, he concluded that the Soviet Union was not under attack and some sort of error had caused the warning. Within 15 minutes it was clear that no missiles had actually been launched by the U.S., vindicating the actions of Lt. Col. Petrov. Post-incident analysis of the Soviet early warning software confirmed the suspected error. The software had, in fact, identified the reflection of sunlight off high clouds as ballistic missiles being launched from the United States.

The world, as we knew it, would live to see another day because this courageous and intelligent man chose to disobey a standing order. Instead of his actions bringing praise from the Soviet leadership, however, Petrov was forced to retire and lived an unsung simple life on his military pension until his actions were officially recognized with the award of the status of “World Citizen” by the Association of World Citizens at the United Nations in New York in January of 2006. Human intelligence, compassion, and reasoning had saved the day over the mechanical programmed warning of an inanimate computer.

1.1.1.2. Patriot Anti-Missile System

Shortly after the start of the first Gulf War in August of 1991, Iraqi forces deployed Russian built Scud missiles against the American troops and their Coalition allies. The Coalition forces initially had no defense against these missiles, but contracted with Raytheon to modify the Patriot anti-aircraft missile for use in Scud missile defense[66]. The software modification was accomplished in just a few months, and was met with great fanfare in the press and military circles. It was hailed as the savior of the Gulf War and was touted with a near perfect success rate in destroying Scud missiles. Post-war analysis created significant
uncertainty with regard to the actual success rate of the missile as a result of the lack of reliable realtime, automatically collected statistics, as well as, the definition of “success.” While the true effectiveness of this system may never be known, there is no doubt that a software problem was the direct cause of the failure to intercept a Scud missile which targeted a military barracks killing 28 U.S. Army soldiers in Dharan on February 25, 1991. The actual software problem was the accumulation of round-off error over a very long run time in the weapon control software[66]. These errors were exacerbated by the velocity of a missile as a target versus the velocity of a target aircraft, for which the system had originally been designed. Israeli forces were the first to uncover the problem and warned other Coalition forces that very long run times affected the accuracy of the Patriot. The warning included a strong suggestion to periodically cycle the power to the system, which appeared to make the problem vanish. What they had unwittingly uncovered was that recycling the systems reset the accumulated error to zero, avoiding an overflow of the accumulator. Unfortunately for the forces at Dharan, the definition of “very long run times” was ambiguous, at best, resulting in the Dharan fatalities.

1.1.1.3. Year 2000 (Y2K)

This was the error that never was an error, and is one of my favorite stories in software. When I was taking my first undergraduate class in Computer Science in 1974 at the United States Air Force Academy, the instructor warned the students to ensure that all dates used in programming employed four-digit year fields to avoid problems around the turn of the century. Twenty-six years later, the world waited for the impending doom in the first seconds of January 1, 2000. There was no doom. The sad part of this story is that despite billions of dollars spent on consultants on Year Two Thousand (Y2K) throughout the world, the software industry failed in that it could not guarantee that nothing bad would happen in the first second of the year 2000. Unfortunately, based on the history of the software industry as a whole, I fully anticipate a Year Ten Thousand (Y10K) problem after the year 9999.
1.1.1.4. Expeditionary Combat Support System

Military projects, by their very nature, tend to be expensive, revolutionary, and teetering on the leading edge of technology. Most of them include new concepts, visionary and radical hardware, and the intricate software required to pull the systems together. Last, but not least, they generally include safety critical components which can endanger human lives on both sides of the trigger. The Expeditionary Combat Support System (ECSS) Program had almost none of these traits. Instead, it was designed as a program to use a commercial off-the-shelf (COTS) software package to integrate over 200 Air Force logistics software programs running throughout the service into one, single, overarching logistical mega-system[39].

In 2005, Oracle was awarded an 88.5 million (M) dollar contact over its rival, SAP. In 2006, Computer Science Company (CSC) was awarded a contract in the amount of $628M as the lead System Integrator. Six years later, and already $428M over-budget, the Air Force conceded that the Program had not yielded any significant military capability. To make matters worse, the Air Force estimated it would require an additional $1.1B just to obtain 25% of the originally contracted capability! The development of that scant capability would also require an additional 8 years of development time. Facing these staggering numbers, the Air Force decided to cancel the program and rely, instead, on its legacy systems. CSC was given an additional $8M as compensation for shutdown costs, despite its total failure to complete even one of the program requirements.

So, as computer scientists, we must ask ourselves how this tragic waste of time, talent, and money ever occurred and how such a thing can be prevented in the future. Root cause analysis by the Institute for Defense Analysis (IDA) concluded that the root cause was that “the people who began this program had insufficient expertise in what they were buying. As the scope of ECSS is so big, it is possible that nobody in the world really knew how long it would take or what it would cost to acquire this system. It may also be that it is impossible to know how much a large program will cost until the blueprinting – a significant
portion of the cost – is done[21].” In addition, this massive program made a tactical error in its decision to include the implementation of a replacement for all of the more than 200 legacy logistical programs in a single, initial release. Better requirements and better program planning may have been able to successfully implement this program on a phased schedule. Finally, the IDA report indicates that there was also evidence that government or contractor personnel responsible for program management did not fully report shortfalls in program accomplishment. Metrics and measures of progress or goodness are only as useful as the input data is valid. Invalid data, accidental or intentional, leads to invalid actions.

1.1.1.5. F-35 Program Performance

On April 3, 2014, the DOD announced that the F-35 Program would have to push initial operational testing out for at least 13 months due to software testing problems with the early releases of the software[14]. The news report was based upon a Government Accountability Office (GAO) report which was released in March 2014[15]. The scathing GAO report begins by admonishing the DOD for starting the program in October 2001 “without adequate knowledge about the aircraft’s critical technologies or its design”. While the list of technical, financial, and schedule problems on this program is daunting, software assumes the blame because it is, as always, the last developmental item to be loaded onto the system. Software errors still make front page news!

All of these examples and so many more (e.g. National Aeronautics and Space Administration (NASA) Mars Climate Orbiter[9] in 1999 and the Automatic Brokerage Problems on Wall Street[13] in 2012) provide strong evidence that the current tools, practices, and processes employed by the software industry have fallen short of the goal of producing error-free software products. Software errors that result in human deaths are the most egregious of all. A race condition in the Therac-25 radiation therapy machine caused the deaths of at least 6 cancer patients in 1985 and 1986 while the Cobalt-60 radiation therapy machine caused a number of deaths in 2000-2001 [43].
Most of these examples are based on military software programs not because it is the only arena in which software errors occur, but merely because these types of issues, at face value, pose the most devastating potential impact on society. Finding the more mundane examples is as close as the nearest computer terminal or hand-held device used in association with any available search engine. The bottom line is that historic and contemporary software development processes and tools are obviously fraught with opportunities for insidious failure. Our technology-intensive society poses problems requiring increasingly complex solutions. This, in turn, forces more and more complex software solutions. In response to that need, software tools are becoming commensurately more complex and, with that complexity, so rises the risk of unforeseen errors due to misunderstood or unintended implementations of those tools. Processes created to avert these types of errors rely largely on the abilities of the human reviewers assigned to the tasks and it is not cliché to state that all humans make errors. It is reasonable to think, therefore, that keeping solutions as simple as possible rather than adding unnecessary complexity will have a positive impact on future software systems.

The search for simple solutions in software development is certainly not a new idea. In 1987 Barry Boehm identified six “primary options for improving software productivity: (1) getting the best from people, (2) making steps more efficient, (3) eliminating steps, (4) eliminating rework, (5) building simpler products, and (6) reusing components.”[28]” In April of 2014, I attended the 26th Annual IEEE Software Technology Conference 2014 at which Dr. Boehm was a featured presenter. He is still on the soapbox for simplicity. In the Air Force, we called this the Keep It Simple Stupid (KISS) principle. It is my contention that the complexity inherent in the contemporary software development tools, practices, and processes have played a factor in these and many more failures. The over-emphasis of process over product has diverted development time from tasks focused on finding errors in or testing the correctness of the software product, to tasks directly related to the generation of objective evidence (OE) to support automated process tools. Automating the collection of metrics
has created an environment in which engineers are rewarded more for error-free entry of data into the complex metric-collection tool in a timely manner than for the quality or correctness of the software that they create or review. I also propose that simpler, not more complex, product-related software development tools can help reverse this trend. The success of the hardware engineers using CAD tools certainly supports this theory. The use of the simple Hy-C can increase the efficiency of generated software with the automated distribution of code across processing components. Furthermore, if the legacy source code entered into Hy-C has already been proven as correct, Hy-C should be capable of producing equally correct executable machine code. All of these concepts will be discussed in detail in Chapter 2.

1.1.2. Processor History

“Moores Law is a computing term that originated around 1970; the simplified version of this law states that processor speeds, or overall processing power for computers will double every two years” [17]. For the most part, this law has withstood the test of time. From the 1970s through the 1990s, advances in hardware design and materials allowed a steady increase in both clock speed and transistor density every year or two. “The real point of Moore’s Law was not merely delivering improved performance, but delivering improved cost-performance, which meant we got improved performance at a fixed and even reduced cost.” [62] While both increased transistor density and increased clock speed resulted in higher thermal characteristics, simple fans and ambient airflow were sufficient to effectively dissipate heat. Furthermore, the thinner transistors required in the manufacturing of processors of such high density led to significant increases in both power usage and heat generation. The result of this phenomenon was a decrease in the rate of processor performance increases. “Chip performance increased 60 percent per year in the 90s but slowed to 40 percent per year from 2000 to 2004, when performance increased by only 20 percent.”

The initial industry response to this situation was the introduction of chips with multiple, cooler-running, and more energy efficient processing cores instead of one increasingly
powerful core. The individual processors in these chips do not necessarily run as fast as the high-end single processors, but they improve overall performance by handling more work in parallel. Unfortunately, the development of compilers that can efficiently extract parallel execution across multiple processors has been slow to materialize. Thus, parallelization often becomes a manual task assigned to the development team to extract parallelism from a development system. As mentioned earlier, any and all efficiency created in this manual mode will be at risk of being insidiously lost with each and every change in the software.

1.2. Heterogeneous Processors on a Chip

The most recent entry in the processor field combines different types of processors, each with its own processing efficiencies and capabilities within a given niche, to allow efficient processing throughput in parallel with energy efficiency and a corresponding decrease in heat dissipation problems. There are many considerations to be taken into account in the selection of the computing resources for a given heterogeneous processor. One of those considerations is flexibility while another is performance. In an ideal world with infinite resources, one could purchase a device with both absolute flexibility and excellent computing performance. Unfortunately, this is not an ideal world. A comparison between flexibility and performance of ASICs, FPGAs, and general purpose CPUs is provided in Figure 1.1. The choice of the components for a heterogeneous processor on a chip will depend on the application.
environment in which the processor will be used.

Another consideration is the full lifecycle cost of constituent computing resources. Component costs include the manufacturing cost, also known as the recurrent cost of development, and the hardware/software engineering required to establish requirements, design a solution, and implement first items for evaluation. The latter part, the engineering effort, is known as non-recurrent engineering. As the name implies, non-recurrent costs are incurred only once in the system lifecycle while recurrent costs continue throughout the life of the item. Some items have a very high development cost, but that cost is acceptable if the cost can be amortized across a very large production run. ASICs fall into this category. Unfortunately, once manufactured, an ASIC design cannot be altered without incurring the full cost of a complete design lifecycle. Mistakes in an ASIC are very expensive to the manufacturer. The other problem with ASICs is that their total functionality is defined at the time of manufacture. This can be a very big problem if a development organization specifies and purchases custom ASIC devices and later determines that changes in their development system design require additional or different processing support from the ASIC. Design mistakes like this can be very expensive to the ASIC customer. As a computing resource, however, customers who absolutely need high-end processing are willing to accept the risks associated with ASICs, giving up flexibility for maximum performance.

On the other end of the spectrum of computing resources, general purpose CPUs provide nearly complete flexibility as a totally programmable resource but has less performance potential than either ASICs or FPGAs. Therefore, if an error is found in software for a CPU, it can be fixed and reloaded to correct the problem. This is not a cheap option either, however, as the cost of an iteration of software is also very expensive on a large program.

Systems being developed for this type of processor architecture are not turn-key, however, in that the software tasking must somehow be divided in an efficient manner across the constituent processing resources. As is the case described for multiprocessors above, this
parallelization could be achieved through the manual design of a development system, but that is likely an expensive solution. Furthermore, changes in the software could be doubly expensive to ensure that no insidious errors are accidentally introduced. Once again, the obvious solution is to employ an automated compiler capable of balancing processing and energy efficiencies with simple input from the developers. Simplicity and automation are the hallmarks of the Hy-C Retargetable Compiler.

1.2.1. Hy-C Solution

The proposed Hy-C System is depicted in Figure 1.2. Hy-C is a retargetable compiler that facilitates the distribution of executable code across a defined collection of heterogeneous processors taking full advantage of the processing/energy efficiencies of the computing resources. Hy-C provides an efficient execution of the input source code on the fixed set of processors in accordance with the Objectives and Constraints. The processor set shown is one possible collection of processor types that may be typical of a processor set for use in hand-held personal devices. The System Specification contains an architectural description of the processing capability and the relative power-usage/thermal characteristics of each device including both processors and available memory. The other user-supplied file is the Objectives and Constraints which will prescribe the objectives for the run. One possible objective might
be to run as fast as possible with no regard for power consumption. Alternatively, a user might want the system to run with minimal power consumption, regardless of the processing speed. Realistically, users will probably specify something between these two extremes. The Partitioning Process intelligently segments the source code into functional groupings based on the known processing capabilities of the available processors and directs those code segments to the applicable device for execution. If sent to the CPU device, the code is compiled by the native compiler for that processor. Partitioning uses graph matching to determine which code segments are sent to either the FPGA or ASIC. The functionality of an ASIC is well defined. For the initial work using FPGA, defined functions will be programmed into the FPGA. Ultimately some FPGA gate routing code could be synthesized on the fly in later iterations of Hy-C. The processing time and power usage metrics are gathered in the Power Performance modules and all of this data is sent to Optimization Control for performance analysis of the entire system. This function can be used to direct modifications to the partitioning design or, perhaps, to suggest a change in the hardware mix. Each of the Hy-C components will be discussed later in further detail.

1.3. Retargetable Compiler Simplification

The topic of this dissertation is a discussion of retargetable compilers over the years and how modern electronic component advances have opened the door for simpler, domain-specific applications of retargetable technology. Having spent a career in software engineering, I present the topic in its context within the software engineering industry as a tool with some potential for simplifying the software engineering process for important application areas. To demonstrate the simplicity of one potential domain-specific application, some details will be provided about the design of the Hy-C system and an example architecture description file sufficient for inclusion in Hy-C will be discussed. Simplicity is the hallmark of the Hy-C due to the use of off-the-shelf components. This is exactly the market which is currently skyrocketing due to the ever-changing face of both the hand-held electronic device market
and the military-industrial complex. The automatic recompilation of existing source code along with simple user-inputs is expected to increase execution performance of software due to parallelization of the code across the available computing resources, decrease the number of errors through the use of existing, proven application software, and decrease development time due to limited user input and automation.

1.4. Overview

While a brief history of software and hardware development has been introduced, it will first be necessary to dive deeper into the past and current practices, processes, and tools used in product development. Chapter 2 explains how hardware and software development arrived at their current states. This discussion is guided by my 29 years of experience as a software engineer, software test engineer, and software manager working on government programs at two corporate contracting companies. In the end, it will yield the void or flaw in current practices that Hy-C is designed to fill or correct. It will answer the question of why a relatively simple system design solution like Hy-C is needed. Since Hy-C is a retargetable compiler, a discussion of retargetable compiler technology, past and present, will be presented in Chapter 3. Given the number of retargetable compilers available, why is there a need for something new? Once the need for this new compilation system is established, Chapter 4 identifies and discusses the essential elements of traditional architecture descriptions. With the background well established, Chapter 5 provides the details of the architecture description which I have developed to support the Hy-C operation, in its simplicity. One example implementation of this architecture description is provided to support Hy-C processing with the target architecture based on the Texas Instruments Open Multimedia Application Platform™ (Texas Instruments) (OMAP) heterogeneous processor. The fully implemented architecture description is provided in the Appendix. Chapter 6 presents the closing arguments for the dissertation.
Having established that software is error-prone, it is important to discuss the product development paths that have produced these errors. No attempt is made to identify exactly where in the process errors are introduced or how to avoid any specific errors. These are topics of significant research in the field of Software Engineering and nobody, to date, has been able to solve the problems. Instead, the analysis of the current processes and tools identifies significant complexity as a potential contributor to software errors, thus lending support to the argument that a reduction in complexity, when possible, can contribute to less error-prone software[28].

The development of solutions for a given set of system requirements consists of many tasks. DSE refers to the activity of exploring and evaluating design alternatives prior to implementation to optimize desired outcomes in accordance with known system constraints. These constraints may be cost, physical characteristics, (area, size, weight) or they might be functional goals (processing capacity, energy consumption, heat generation/dissipation). DSE is one of the first tasks in a development program because its outcome can add or modify system requirements and definitely impact the final choice of hardware and/or requirements allocation between hardware and software. The proposed Hy-C re-targetable compiler can, in fact, be used as a DSE tool.

Most hardware companies define proprietary processes for development of their products. Being hardware-centric, these processes, in the early days of integrated circuits, tended to center on the creation of drawings and manufacturing processes that produced products in accordance with those drawings. In the 1950s and 60s, companies employed technical artists, also known as draftsmen, to actually create renderings of hardware by hand. In the late 1960s[1], the industry began its transformation to CAD tools which totally revolutionized
the hardware industry. Drawings could now be made at the micron level with the ability to be fed directly into machines for synthesis of the required computer chips. The rendered CAD drawings look just like the finished product. An engineer can look at a drawing and instantly recognize what the finished product is intended to do. The designs are the result of human thought processes, but the outcome of that process is an abstraction that resembles the product under development. The CAD software is even capable of verifying the physical size and shapes to mathematically verify that all designed parts will fit together with an assigned, usually specified, tolerance. The CAD drawing is an abstraction of the design, but the tools used to construct that abstraction allow the automatic generation of most hardware components. This is automated engineering at its finest. Human thought and human innovation is abstracted into a design which automatically generates the desired physical product through the use of tools developed for the purpose of creating the product. Hardware organizations do follow processes that require the creation of documentation in support of the product, but the main flow of the development process centers on creation of the product. Software lacks that type of automation and visual abstraction.

2.1. Software Development Historical Overview

The problems encountered by computer programmers in the earliest days of software development in the 1950s and early 60s were much different than are faced today by software engineers. For example, on May 25, 1961, President John F. Kennedy spoke before a joint session of the U.S. Congress and issued his famous challenge “that this nation should commit itself to achieving the goal, before this decade is out, of landing a man on the moon and returning him safely to the earth.[40]” These words began a national obsession with space travel and putting a man on the moon that culminated with Neil Armstrong’s famous quote of July 20, 1969, when man first set foot on our celestial neighbor: “One small step for (a) man, one giant leap for mankind.” The challenges encountered between those two dates were mind-boggling and set in motion the earliest vestiges of software engineering. NASA
then instituted the skeleton of a government contractor review process for software that is, to a large extent, still in use today.

The daunting task of creating the lunar lander landing system was contracted to the Massachusetts Institute of Technology (MIT) just ten weeks after President Kennedy issued his challenge. MIT started with nothing but a 9-year challenge and managed to create the Apollo Guidance Computer (AGC) which was so remarkable and robust but simple, that variants of it were used all the way through the Shuttle program. “The AGC weighed 70 pounds, consumed 55 watts of power and occupied only 0.97 cubic feet inside the spacecraft. These first digital flight computers were limited to only 36,000 words of fixed memory and 2,000 words of random access memory (RAM), and operated at a 12-microsecond clock speed ... Before the first lunar landing, more than 1,400 man-years of software engineering effort had been expended, with a peak manpower level of 350 engineers reached in 1968”[61].

The computer programs were installed in rope memory[42], which was a grid of small donut-shaped magnets in a mesh grid. Once the computer programmers were done with their code, it was printed on paper in binary (ones and zeros) and handed to the workers at a Raytheon plant where they would tediously weave the binary patterns on the printed papers into the rope memory. To create a “one,” the wire would be sewn through the ring with a long needle. To create a “zero,” the wire would bypass the ring. Needless to say, because of this manual and tedious operation, the code had to be complete, and it had to be perfect, months before it was needed in the lab or on a spacecraft. Even the smallest change to fix a critical error would necessarily result in months of program delays just to remanufacture and test the rope memory.

Looking back on the days of early NASA programs, one must admire the ability of those early engineers to write code that could control the descent and landing of a manned spacecraft on the moon on the first attempt! The sheer complexity of the code, though
simple by today’s standards, combined with the extremely limited computer resources, make this accomplishment seem almost impossible to modern computer scientists. In truth, the software was not perfect, errors were found, but it was sufficiently robust enough to complete the mission.

By the 1970s, software engineers were using flowcharts to represent the design of their programs and using either assembly language or higher order languages to write the programs. These abstractions allowed the industry to handle even more complex problems, but with this complexity came errors. Some systems were now so complex that exhaustive testing of all paths was becoming difficult and time consuming. As memory became more available and processors became more powerful in approximately two year cycles, in accordance with Moore’s Law, the problem spaces became larger and complexity again increased. By the early 1980s, the complexity of the domain space of computer systems had outrun the programmer’s ability to account for all possible contingencies in programs and saw the advent of the use of basis path testing[64]. The budding industry had already accepted the fact that software products could not be exhaustively tested. At around the same time, the size of the average software project had grown to a point at which flowcharts were inadequate for describing software behavior. This ushered in the era of the Computer Aided Software Engineering (CASE) tools and the Unified Modeling Language® (OMG) (UML®)[7].
Until the mid-1990s, all DOD contractors were required to use contractually-specified detailed development methodologies provided via military standards in effect at the time of the award of the contract. Figure 2.1 shows the chronology of all the DOD sponsored or authorized software development specifications or methodologies employed over the course of the last four decades. Prior to 1985, contracts used MIL-STD-1679 for Navy programs and all other military contracts fell under MIL-STD-1644, but these standards are ancient enough that they could contribute little, if anything, to this discussion. In 1985, DOD-STD-2167, Defense Systems Software Development, superseded these documents for all software programs. Contractors complained that the requirements described in DOD-STD-2167 were too closely aligned with the waterfall paradigm of software development and prevented the companies from making sound business decisions with regard to their software processes to make them more relevant and less costly in a changing environment. Revision A of DOD-STD-2167 was released in 1988 to allow the contractors some latitude in the software process, with customer approval, of course. Yet, contractors continued to complain that the Revision A modifications still did not provide them with the necessary freedom to implement cutting edge, cost saving, measures in their development processes.

Technology improvements starting in the 1990s were so fast and so substantial that the military could no longer ignore their benefits. Until this time, most equipment developed for military use had to be “militarized” or specifically built to DOD standards as a deliverable development item. The $600 hammer developed for the DOD under the DOD acquisition rules was widely heralded in the media to represent rampant waste in government procurement programs. Whether or not any hammer actually cost that much, nobody can deny that developing equipment within the bounds of the military specifications incurs a significant increase in cost when compared to comparable items available through commercial procurement. That added cost is a direct result of the required design, development, and testing of any equipment manufactured specifically for a DOD program. Commercial off-the-shelf items
were considered too fragile for the battle environment, so significant costs were accepted to create militarized versions designed to operate in harsh or volatile conditions.

The practice of creating every piece of equipment from scratch under each DOD program came to a screeching halt in 1994. That year William J. Perry, the United States Secretary of Defense, issued a letter which came to be known as the “Perry Memo.” The letter directed “greater use of performance and commercial specifications and standards as one of the most important actions that the DOD must take to ensure we are able to meet our military, economic and policy objectives in the future.” It went even further by requiring the use of COTS equipment unless no such equipment existed in the commercial community. The memo also eliminated the need for most military standards. In response to Secretary Perry’s memorandum, in 1994 the DOD released MIL-STD-498, Software Development and Documentation, which superseded DOD-STD-2167A and some other developmental standards. The expressed purpose of MIL-STD-498 was to bridge the gap for DOD contractors until a commercial standard, or a sufficient company process, was in place to direct the future product developments. This standard only lasted 2 years before the Electronic Industries Association (EIA) J-STD-016, (Trial-Use Standard) Standard for Information Technology Software Life Cycle Processes Software Development Acquirer-Supplier Agreement, was put in place by Institute of Electrical and Electronics Engineers (IEEE) as a model for IEEE-12207, Systems and software engineering – Software life cycle processes. Use of IEEE-12207 which was first released in 1996, or an equivalent commercial development process, is now the standard operating procedure for DOD contracts.

The final product of note in this discussion is the role of the Software Engineering Institute (SEI) with respect to DOD development programs. The SEI was established with funding from the DOD in 1984 for the purpose of advancing software engineering principles and practices and to serve as a national resource in software engineering, computer security, and process improvement. Three years later, SEI first described its Capability Maturity
Model® (Carnegie Mellon University) (CMM®) for software engineering followed by the formal release of CMM® 1.0 in 1991. This model did not tell contracting companies how to do their engineering tasks, it simply provided a list of required capabilities. SEI then established an industry within itself by creating definitions for 5-levels of compliance with the listed capabilities. The overall compliance designation for an organization was set at the lowest compliance level assessed for any individual capability. Furthermore, SEI designated itself as the sole provider of assessment personnel to determine the level of compliance. Finally, given the significant investment that it had sunk into this project, the DOD required all contractors to be assessed at a minimum level of 3 before they could enter into any new contract with the DOD. In 2000, the SEI expanded its scope from just software, to software, systems engineering, and program management, thus involving the complete development process, with the release of its Capability Maturity Model Integration® (Carnegie Mellon University) (CMMI®). All developmental functions now fell under the CMMI® umbrella.

Each of these processes and their implementations will now be discussed to provide clues as to why they might have failed to prevent or uncover software errors before fielding the products. The answer, in my opinion, is much deeper than simple human error.

2.2. DOD-STD-2167

DOD-STD-2167, Defense Systems Software Development, was an attempt by the DOD to implement a single process to guide the development of all software produced as a result of DOD projects. The DOD-STD-2167 Product Development Plan is depicted in Figure 2.2. The plan looks fairly simple consisting of steps/phases familiar to software engineers today: develop requirements, analyze those requirements, create a multi-level design, write the code and test it at low levels, and finally test the compliance of the full software system against the established requirements. Each phase concluded with a formal review, the successful completion of which allowed the contractor to continue work on the program in the next phase. Note that the phases shown in Figure 2.2 do not overlap. The process
was completely serial. Finish one phase, then start the next. Also note that, even though this is a software development document, it depicts the synchronized hardware development cycle, with common software/hardware formal reviews. Not shown on the figure are all of the documents and deliverable items associated with each of the formal reviews. A list of those documents, grouped by type, is provided in Table 2.1.

Some of the listed documents were designed to be finalized and approved at specified formal meetings. Others were meant to be living documents, continuously updated, and approved, throughout the duration of the contract. Finally, the format and contents of each of the documents were specified in excruciating detail in an associated and unique Data Item Description (DID). The use of the DIDs was intended to standardize the documents across all programs supported by the DOD. This proved to benefit the government oversight community far more than the contractors. The standardized document formats made it easier for government managers to oversee and compare the work of multiple contractors across
multiple programs without additional training. To provide some idea of the detailed nature of these DIDs, an excerpt from the DID for the Software Detailed Design Document (SDDD), DI-MCCR-80012A[36] describing the documentation required for each and every temporary, local, or global variable or constant declared in any piece of software related to the product under development is shown below:

- Data item name (project unique identifier)
• A brief description

• The units of measure, such as knots, seconds, meters, feet, etc.

• The limit/range of values required for the data element (for constants, provide the actual value)

• The accuracy required for the data element

• The precision/resolution in terms of significant digits

• For real time systems, the frequency at which the data element is calculated or refreshed, such as 120 KHz 50 Msec, etc.

• Legality checks performed on the data element

• The data type, such as integer, ASCII, fixed, real, enumeration, etc.

• The data representation/format

• The Computer System Unit (CSU) project unique identifier where the data element is set or calculated

• The CSU project unique identifier(s) where the data element is used

• The data source from which the data is supplied, such as database or data file, global common, local common, compool, datapool, parameter, etc. Where applicable, each source shall be identified by its project unique identifier.

The description above did not apply to data retained in any formal data base. Those data were described in yet another document, the Data Base Description Document (DBDD) which was equally painful. Each of the fields listed above has some subjective value for some data elements, that is, variables and constants, in the software under development. Very few variables and no constants, however, require all of the fields to provide a complete description of the function of the data item. Yet, the customer oversight personnel assigned to every program on which I ever worked required every field for every variable. Fields which were deemed unnecessary for a particular variable were simply marked “NA.” A computer program of even a modest 1,000 lines of code could have hundreds or thousands of variables. Most
government systems have millions of lines of code. The size of the SDDD is staggering, especially since the vast majority of the lines are marked “NA.” A reasonable, thinking person should be able to understand how much time is wasted documenting non-applicable fields in this document alone. Surely this has to be considered non-value-added effort which should be eliminated from the process. Yet, rarely were the development plans tailored to eliminate even this obvious example of wasted effort.

Making modifications to the stringent rules of MIL-STD-2167 was certainly an option for the customer personnel. As a matter of fact, tailoring was directed by Paragraph 1.3 of the standard. “Software shall be developed in accordance with this standard to the extent specified in the contract clauses, Statement of Work (SOW), and Contract Data Requirements List (CDRL). Guidelines for applying this standard are provided in Appendix D. The contracting agency will tailor this standard to require only what is needed for each individual acquisition.” Unfortunately, customer personnel were loathe to tailor out any elements of the DIDs, probably because they were not technically competent enough to fully understand the value or purpose of the individual fields. Thus, the Programs tended to be characterized by meticulous document reviews by bureaucrats with more interest in format than content. No matter how complex the actual software might have been, the complexity of the documentation leading to the creation of the software was daunting, indeed, and all too often consumed schedule time that had been planned for the code and phase. It was truly a case of losing sight of the forest for the trees. The product in software development is the code. Requirements and design are important also, but delaying the start of the coding phase for the sake of non-value-added cosmetic manipulation of the documents leading up to the coding phase is absolute suicide for a program schedule.

As a result of this situation, most of the early automation CASE tools for software attempted to automate the documentation portion of the development process. No real attempt was made to automate the creation of the software itself through abstraction. The
creation of software is the codification of human thought, but that human thought has to be brought into the light of day through meaningful visual media as a starting point for the software. The tools created and used by hardware organizations successfully abstracted human thoughts and designs into detailed drawings, easily recognizable as representative of the product. Tools which attempt to provide abstraction of software simply do not provide this clarity of purpose.

One CASE Tool adopted by a program on which I worked in the mid 1990s attempted to simplify the software development process by using embedded textual commands within text files to direct text to specific documents. This looked a lot like the \LaTeX commands look today. Early in the conceptual phase of the program, the documentation might start with an Operational Concept Document describing how the system is intended to be used upon completion and acceptance by the customer. That one file would then be updated with annotated requirements associated with each of the described capabilities or features. That single file would then be logically divided into a number of files, possibly grouped by function. In the design phases, the files would be further divided according to the chosen design, but all of the conceptual text and the associated requirements would still be together. Finally, the actual code would be inserted, breaking the file structure further into individual code modules. Embedded within each of these modules was a description of the high-level purpose, associated requirements, design, and code specific to each code module. If modifications to the code were necessary, all of the supporting documentation would be co-located within the text file for that module and could all be kept in lock-step synchronization.

At face value, this CASE tool looked like the answer to the prayers of the developers and systems engineers alike. Those painful software design documents could now be generated automatically. The engineer merely wrote code to instantiate a variable or database and these data would be automatically inserted wherever applicable in the place and format required for the necessary customer-specified documents. If it looks too good to be true, it probably is.
And it was!

First of all, the program was developing a very technically difficult, state-of-the-art, application. That, alone, would have and should have occupied the full time and resources of the engineering community. Budget and schedule constraints forbade the attendance of all engineers at the full training for the CASE tool. Instead, the program developed a short, one or two day, powerpoint presentation to provide an overview of the tool. Beyond that, they were expected to collaborate with each other to learn all the nuances of the tool. In general, all text for a given document would be encased in paired /begin and /end command tags for that document. A single text file could contain tags for many different documents. When it was time to generate the documents, a document-build tool would find all the text destined for the user-specified document and format the text into the required DID format.

While conceptually simple, it is not difficult to understand how those matched pairs of begin-end statements might get out of order. As the files were broken up to reflect more and more specific functionality, it was incumbent upon the engineers to ensure the integrity of the text across all document boundaries. When a file was divided in the course of the design and code process, each of the files now had to be self contained. Each engineer doing the file-split operation had to ensure that no begin-end pair had been broken across file boundaries. Each individual file had to be self-contained. All too often, this did not happen. The specific export of documentation could not be done on a piecemeal basis, but generation of the complete document was not even possible until the applicable development phase was nearly complete. This allowed a significant number of errors to accumulate in each constituent file, impacting the ability to generate the documents in a timely manner in accordance with the defined schedule. Furthermore, when the miracle occurred and a document was created, it looked like a story cobbled together by 15 authors who had never agreed on the storyline. The information may have been correct, but it was very difficult to read. Worst of all, the document itself could not be edited to correct the awkwardness because it was being pulled
from scores of input files. Therefore, text in each individual file had to be modified to make the document, as a whole, read better.

One absolutely critical aspect of any large product development is a process known as Configuration Management. Once a file of any sort is created and accepted, it is formally put under configuration management control. Every change from that point forward must be meticulously documented in and of itself. The specific changes must be identified, approved by a Configuration Control Board, and only then can the product be changed. At some point, the rules require the approval of the customer before any changes can be made. As such, making cosmetic or grammatical changes to scores of source documents in order to make an "automatically-generated" formal document read nice, is a big, expensive, and time-consuming task.

This CASE tool had never been proven on a large-scale software development program and it was not a cheap. Thus, the program had to go out on a limb to get the customer to approve of the use of the tool. The program management and the customer accepted the risk of the use of the unproven tool because of the promise of automatic generation of any of required documents with the most current design information at the push of a button at any time. The engineering staff noted some problems early in the development and offered to pay for one vendor engineer to reside with the development team to clear these issues. Later, another full-time engineer from the vendor was retained. As we got closer and closer to the scheduled date of the Software Requirements Review (SRR), things got very tense. We could not produce the Software Requirements Specification (SRS) in time to meet the delivery schedule to provide the customer with the time necessary to review the document before the Review. This was explained away as a one-time problem. Once we ironed out all the bugs, the document could be generated at the push of a button at any time. When the date of the SRR loomed, we had to admit defeat, but would fix the problems and let the Customer know when the SRR could be rescheduled. Six months later, the SRR was held
with only a partially available document. Some of the requirements were actually reviewed
from their embedded text files! Needless to say, the customer was not happy. Furthermore,
the six month slip in the Requirements Phase meant that the start of the Preliminary Design
Phase was necessarily pushed back six months. We never did get the documentation problems
settled. The SRS was never completely correct, but was accepted by the customer and design
work began. That phase was delayed for over a year before the customer withdrew support
for the CASE tool and everything was done manually through the end of the program. This is
not the kind of simplification that will improve the software engineering process or reputation.
Under the rules of DOD-STD-2167, work on the next phase of development could not begin
until the current phase was complete and the customer issued the start work order on that
next phase. As such, delaying the completion of any phase had a ripple effect on the entire
schedule.

CASE tools, in general, suffered from a problem with scaling. The tools were tested
before release, but what company can afford to expend the resources to test the development
tool on a program of hundreds of software engineers and years in length? Validation in this
arena is trial by fire. A large program must assume some level of risk in adopting a state
of the art tool with little history. It is a curse of the industry in which we practice our
trade. Developing testing scenarios for this type of tool that might make its introduction into
commercial use a little smoother would be a great topic of research for some willing university
with graduate students who have nothing else to do!

Defense Contractors had many complaints about DOD-STD-2167. First, it formally
adopted the top-down, waterfall method of software development. The word “iteration”
does appear in the text, but only tangentially. Contractors also complained about the re-
lentless rigidity of the software standard[48]. Exceptions to the strict guidance provided in
the standard were almost never granted by the government customers. Other criticism of
DOD-STD-2167 generally fell within three broad areas[53]:

32
(1) It applied to all DOD projects, regardless of size or maturity.

(2) It was based in large part on MIL-STD-1697, but software engineering, as a discipline, had evolved in the 6 years since that standard had been adopted, but the DOD software development standard had not.

(3) Many companies had already invested heavily in proprietary tools and processes for commercial software development. The rigid requirements of DOD-STD-2167 put that investment in jeopardy. Those commercial processes were not necessarily compatible with the development cycle required by the military standard.

DOD-STD-2167 was an attempt by the DOD for simplification within the development community of defense contractors, but it was not a process simplification for the contractors nor did it directly aid in the quality of the delivered software. It was, in fact, a simplification of the development process for the government oversight community dictating that all programs, regardless of size, complexity, or scope, follow the same stringent documentation and formal review process. Other than listing the reviews and documentation required for the software in meticulous detail, the complete, albeit short, list of actual coding standards to which the developed code must conform follow:

- No GOTO statements
- Control Constructs are limited to SEQUENCE, IF-THEN-ELSE, DO-WHILE, DO-UNTIL, or CASE
- Unit modularity: The average length of software units cannot exceed 100 lines of code … no unit can exceed 200 lines of code
- Avoid compound negative Boolean expressions
- Avoid nesting beyond 5 levels

2.3. DOD-STD-2167A

In response to the growing criticism of the rigors of DOD-STD-2167, the DOD updated the standard and released Revision A. The DOD-STD-2167A Product Development
Plan is depicted in Figure 2.3. The major change in Revision A is to break its association with the pure waterfall life-cycle model by allowing an iterative approach to the design and review process in an attempt to accommodate prototyping. Yet, “it is clear that DOD-STD-2167A was not specifically developed with iterative design in mind, even though it is specifically mentioned in paragraph 4.1.1. Regardless, it has been shown that iterative requirements analysis, software design, and system development may be performed under DOD-STD-2167A, if contractors and government procurement officers take the time up front, to tailor the life cycle for each major system development effort[52].”

A visual comparison of Figure 2.2 and Figure 2.3 should yield several distinct differences. The most obvious change is in the initial phases of development. DOD-STD-2167 only had one global phase for the System Concept followed immediately with software and hardware engineering heading down what appears to be mutually exclusive development paths for the remainder of the program until it was time to integrate the hardware and software to-
together. Each developed requirements, design, and products independently, but held common formal reviews. This meant that if either hardware or software development was delayed, the other had to stand down until both were ready for the next formal review. Since customer approval of the formal review was required to begin the next development stage, valuable development time was wasted before each formal review.

DOD-STD-2167A, on the other hand, employed two initial joint phases which produced both the system level requirements and the system level design before partitioning the requirements to either hardware, software, or both. While this may just look like a shuffling of boxes on a chart, it is significant in that it may have been the first depiction of what would, in modern development terms, come to be known as the architecture or the architecture exploration phase. Another significant change is that the hardware and software development phases overlapped, allowing an early start to phases when the previous phase was not yet complete. Of course, proceeding to the next phase before receiving full acceptance of the predecessor phase by the customer was work that was always at risk of needing rework in the event of design changes initiated by the customer at a formal review of the predecessor. The use of some discretion in the selection of the task, or tasks, to be started early, however, could greatly mitigate this possibility. Furthermore, DOD-STD-2167A separated the formal reviews for the hardware and the software, thus allowing products to be reviewed when the product was ready without the need of introducing a schedule gap waiting for the completion of related hardware or software. Finally, some of the formal reviews had annotations allowing multiple, incremental, formal reviews instead of trying to align all of the scheduled tasks to have a single, combined, one-time hardware and software review. This constituted tacit approval for an incremental, or possibly even a spiral, development model.

These changes certainly provided some relief from the rigid waterfall model of DOD-STD-2167 and promoted the use of prototyping, but they did not address the documentation burden in the least. Contractors were still rigidly bound to the DID for each document, as
previously discussed. As such, the customer oversight personnel still focused mainly on the format of the documents. How could customer personnel approve these documents that were only partially completed due to the iterative nature of the development process? This was a major problem on programs on which I worked. The situation was exacerbated by the addition of incremental development for statically defined documents. Prototypes, by their very nature, are investigative tools providing a quick implementation and a rapid test of the feasibility of the concept under test, whether it be hardware or software. The use of a prototype is appropriate when the system requirements are defined, but very complex. The prototype helps break down the complexity and simplify a design by quickly and cheaply testing design alternatives to find a workable solution. Despite the transient nature of prototypes, customer personnel, and sometimes company management, often required the application of the full developmental process in the creation and testing of these one-time-use artifacts. Any savings attributable to the use of the prototype was more than consumed by the creation of throw-away documentation and reviews.

Simplicity is the theme of this dissertation and, in my experience, the achievement of as simple a solution as possible should always be the goal of a developmental system, especially a software system. The more complex the code, the harder it will be to test and maintain. The best software engineers, usually considered those who code the fastest, generally get shuffled from program to program to practice their trade. All too often, once the code is minimally working, the software heroes head to the next program leaving their code to whomever might be left behind to coax it through the remainder of the development program. I have often told students and young engineers to revisit some code that they wrote at the beginning of a semester or a few months ago and see if they are able to determine the functionality of the software just by reading the comments. If not, maybe they need to revisit their embedded documentation or try a simpler approach next time.

DOD-STD-2167A had its faults, but it was an attempt by the DOD to provide flexibil-
ity to the contracting companies. The process, applied with some modicum of common sense, provided all the flexibility needed to simplify the development to a great degree. The mindless implementation, however, both on the customer side and the contractor side, imposed the maximum process and the maximum amount of documentation for each component, piece of test equipment, prototype, and integration tool. As a result, contractors had essentially the same complaints about Revision A as they had with the original version. Once again, however, this standard provided a process and contractors developed automation to implement the process. This did little to increase software quality or prevent software errors.

2.4. MIL-STD-498 Software Development and Documentation

1994 was a landmark year in the area of DOD software development. A new Administration had taken office in Washington, D.C. in 1993 and the new U.S. Secretary of Defense, William J. Perry, used that year to review the acquisition process for the DOD. He then shook up the defense acquisition industry. Seeing the failures of the previous and existing military standards for product development, Sect. Perry abolished the need for them while announcing the policy that DOD procurement programs would instead follow commercial standards for future contracts. DOD-STD-2167A was replaced by MIL-STD-498, Software Development and Documentation, which would only remain in effect until a viable commercial standard was in place. Just two years later IEEE-12207, Systems and software engineering – Software
life cycle processes, was released and MIL-STD-498 was canceled. IEEE-12207 was a near replica of MIL-STD-498, for which the System Development Cycle is depicted in Figure 2.4.

The System Development Cycle represents a dramatic change in the DOD acquisition development paradigm, as compared to those depicted in the preceding military standards in Figure 2.2 and Figure 2.3. MIL-STD-2167 and MIL-STD-2167A provided detailed requirements for how to create products. Development guidance provided in MIL-STD-498 and beyond specifies what has to be done, but relies upon the technical experts of the selected contracting company to specify how those things will be accomplished. The contractor must create a development plan and have it approved by the government oversight personnel in the planning phase of the contract. The process created for each product can, and should, be as unique as the product itself. While the government still possesses approval authority, the responsibility for defining and executing the process is totally in the hands of the contractors. The government still specifies which documents from the list of DIDs are required for a contract, but the contractor is expected to tailor the DIDs commensurate with the complexity, size, or mission criticality of each individual product.

The process depicted in Figure 2.4 should be quite familiar with any modern software engineering practitioner as it describes the basic development cycle employed by most current software development paradigms. Its circular shape reflects the current reliance on cycles of development. Few large, complicated, software products are produced in a one-shot waterfall fashion anymore. They rely, instead, on spirals, iterations, or sprints which build system capabilities in a predetermined sequence of ever-increasing functionality. The sequence of functionality selected for the cycles must be carefully planned to complete within that assigned cycle. The development for Cycle 2 normally depends on proven functionality of features in Cycle 1. Failure of Cycle 1 to complete all design, code, and testing within the allocated time results in Cycle 2 either delaying its start or, even worse, starting its cycle in parallel with the previous cycle. Under the latter scenario, modifications made to the Cycle 1 baseline to
complete its total implementation must be manually inserted into the Cycle 2 baseline. This constitutes additional cost in the form of rework, and quite possibly extending the schedule of Cycle 2 in the process. Then what happens for Cycle 3, or 4, or “n”? The parallel development solution also implies that a program has adequate software resources to fully staff both cycles concurrently, which would be a very unusual circumstance. An even worse scenario would be the decision to delay the full implementation of Cycle 1 to a later cycle. Assuming the cycles have been planned judiciously, by definition there will be no available resources to implement delayed features in future iterations. Besides, the nature of software development in large, complex systems has to assume that changes will be necessary as testing uncovers unexpected design flaws or side-effects. Failure to allocate sufficient resources to a cycle to solve these problems as soon as they appear would be totally irresponsible.

While such an iterative process may appear to add complexity to the casual observer, it actually can have just the opposite effect. In a cyclical development, a program can concentrate on getting an infrastructure in place early which will allow data collection and analysis or prototyping to mitigate risks early. Risks might include such issues as human interface configurations, potential data processing bottlenecks, or revolutionary state-of-the-art processing for which there is no legacy knowledge. Attacking the hard questions first avoids the trap into which many programs have fallen by performing to their calculated schedules early by front-loading all the easy components, only to fall woefully behind as the more difficult tasks are worked. The depicted System Development Cycle includes the terminology of MIL-STD-498, but by substituting phraseology from other modern methodologies, it can represent the majority of modern software development paradigms.

Since MIL-STD-498 only told contractors what to do without specifying how to do it, contractors began searching for available commercial products or developing their own software development tools to comply with the spirit of the tasks specified in the new military standard. In either case, the products generally looked like a retooled DOD-STD-2167A with
<table>
<thead>
<tr>
<th>MATURITY LEVEL</th>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initial</td>
<td>Chaotic, ad hoc, individual heros. Undocumented processes</td>
</tr>
<tr>
<td>2</td>
<td>Repeatable</td>
<td>Process documented sufficiently to allow repetition of steps</td>
</tr>
<tr>
<td>3</td>
<td>Defined</td>
<td>Process defined/confirmed as a standard business practice,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>including Work Instructions</td>
</tr>
<tr>
<td>4</td>
<td>Managed</td>
<td>Process is quantitatively managed in accordance with agreed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>upon metrics</td>
</tr>
<tr>
<td>5</td>
<td>Optimizing</td>
<td>Process management includes deliberate process optimization/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>improvement</td>
</tr>
</tbody>
</table>

Table 2.2. CMMI Maturity Levels

the most objectionable pieces missing. After all, the DIDs were still required, so none of the documentation could be completely avoided. One unintended consequence of the move to commercial practices and standards was that these standards were no longer freely available to the contracting companies. In the case of IEEE-12207, the latest version (2008) is available from the IEEE Online Store for $280 for either a PDF version or a printed copy, or for $408 for both. It is also available to companies with a subscription to IEEE-Explore, which is available within most large companies. This is a relatively small price to pay, so it should not prevent reputable and solvent companies from competing for contracts in the defense industry, but is was, at a minimum, not convenient. Most defense contractor companies, however, chose a different solution for software development via the DOD sponsored Software Engineering Institute.

2.5. Software Engineering Institute

In 1984, the DOD provided funding to Carnegie Mellon University to establish SEI as a Federally Funded Research and Development Center (FFRDC). The purpose of SEI is to help “government and industry organizations to acquire, develop, operate, and sustain software systems that are innovative, affordable, enduring, and trustworthy[18].” To that end, CMM® was released in 1991 with 5 Maturity Levels, shown in Table 2.2[12], and 22 Process Areas (PAs), shown in Table 2.3[8]. Shortly thereafter, the Defense Acquisition Guidebook (DAG)
was updated to require that contractors could only bid on government and military contracts if they were assessed at CMM® Level 3 or higher. Years later, SEI created CMMI® which institutionalized the Process Areas across all engineering and management entities within a company. For the sake of simplicity, I will only discuss the CMMI® attributes with an assurance that the same principles apply to software-only organizations under CMM®.

Basically, CMMI® provides the 22 PAs in Table 2.3. Each PA is then broken down into a list of specific goals (SG) and specific practices (SP). The SGs and SPs identify what needs to be done, but do not provide the process to get it done. Each contractor must provide the processes to fulfill the associated goals and practices. As an example, the CMMI description of the Configuration Management (CM) PA, one of the simplest of all the PAs,
is shown below:

- PA Configuration Management: The purpose of Configuration Management (CM) is to establish and maintain the integrity of work products using configuration identification, configuration control, configuration status accounting, and configuration audits.

Specific Practices by Goal

- SG1 - Establish Baselines
  * SP 1.1 Identify Configuration Items
  * SP 1.2 Establish a Configuration Management System
  * SP 1.3 Create or Release Baseline

- SG2 - Track and Control Changes
  * SP 2.1 Track Change Requests
  * SP 2.2 Control Configuration Items

- SG3 - Establish Integrity
  * SP 3.1 Establish Configuration Management Records
  * SP 3.2 Perform Configuration Management Audits

The determination of the Maturity Level for a given company is accomplished through a very formal, and commensurately expensive, Class A Standard CMMI Appraisal Method for Process Improvement (SCAMPI). SCAMPs actually come in three different versions, Class A, B, and C. Only Class A can actually bestow an assessment of a Maturity Level on a company. The formal Class A SCAMPI generally lasted for about two weeks at a direct cost of approximately $2,000 per day for each member of the SEI Appraisal Team. Leading up to that Class A SCAMPI, however, was a year or two of work meticulously gathering data and following the defined processes in great detail. If a company wanted to verify specific areas with SEI, an appraiser could be hired for a short and informal SCAMPI Class C. If the company thought it was close to compliance overall, it could hire a small team of appraisers
to provide informal feedback with regard to its readiness for the Class A SCAMPI. In my experience, my company used the same appraisers for the informal and formal SCAMPIs, seemingly guaranteeing a successful Class A SCAMPI, for a price. Moreover, though the DOD only required an appraisal at Level 3, most major companies decided that if Level 3 was good, Level 5 would be great! The resources committed to this endeavor were astounding, but companies held this achievement like an Academy Award for Defense Contractors. Again, in my experience, preparation for the SCAMPI always had a negative impact on a program’s cost and schedule.

Officially, CMMI does not specify a single process, not even one. Furthermore, according to the CMMI Frequently Asked Questions Website[8], “CMMI is about improving performance, not about defining practices or performing practices. It promotes greater predictability and confidence that development projects would be successful.” It strives for continuous improvement in the performance of a project. While that may be true, it is certainly a source of confusion for government contract programs which were forced to demonstrate compliance with the Process Areas of CMMI for nearly 20 years. SEI created CMMI and became the sole source of appraisers for determining the maturity level of contracting companies. Compliance was demonstrated solely by the use of OE. Companies had to hire the appraisers to review the OE that had been gathered as a result of the processes, documented in Work Instructions, that were used to comply with CMMI practices associated with specific Process Areas. The companies had used either proprietary processes and tools or commercial processes and tools to create the OE data, so it is easy to understand how companies actually thought the goal of CMMI was process improvement centered on OE data collection and analysis that would lead to performance improvement.

The distinction between “process improvement” and “performance improvement” is subtle, but critical. The SCAMPI appraisers never witnessed the execution of a single process. The appraisal team never looked at an actual software product or a single piece of
manufactured hardware. They only looked at collected data, OE. The data that concerned
the appraisers most was the statistical data gathered with regard to meetings, inspections,
reviews, and testing. If the right number of people were at a meeting and all the action items
from that meeting were closed in an acceptably short time, the appraisers were happy. If prob-
lem reports were being written and closed, the appraisers were happy. If the company could
show how a process was changed as a result of the data, the appraisers were ecstatic! Yet,
the official line from the CMMI website is that it does not care about performing practices.
How was all of this data captured?

From the company perspective, it was all about the process and the data. If a little
data was good, a lot of data was better. Therefore, the process was automated with tools that
collected everything that could be counted and provided standard analysis for each set of data
so it could direct the next changes to the process. Performance was something that process
engineers assumed would be improved if we only made our process better. We automated
the wrong thing. We spent too much time and money collecting data about the process, but
not enough time testing and generating data about the product. The products, the hardware
and software being developed, were still generating errors and were still over budget and
behind schedule. All software, from the simplest throw-away code in the integration lab, to
the most complex autopilot, was created through the same detailed process with little to no
added value to show for that extra effort. We did not automate the creation of the software
product; we automated the processes and the gathering of statistical data relative to the
processes, with little to no regard for the product. We chose “process improvement” and
hoped for “performance improvement” with little correlation between the two.

As of 2012, SEI is no longer directly involved with the administration of CMMI. The
DOD changed the mission of SEI to focus all resources on evolving other technologies of
higher urgency to DOD than CMMI. Carnegie Mellon University, however, did not want to
simply drop CMMI support, so it created a new entity called the CMMI Institute to manage
CMMI. CMMI is not dead, but it certainly does not hold the prominent position in the field of Software Engineering that it once did. The DAG has also rescinded the requirement for contractors to hold a Level 3 CMMI appraisal. Instead, paragraph 4.1.3.1 of the DAG merely states, “The more critical or complex the software acquisition effort, the more important it is to seek developers with demonstrated experience and knowledge.”

2.6. Agile Development

The final paradigm to be discussed is that of Agile computing. The Agile Product Development Cycle depicted in Figure 2.5[38] is not all that different in looks from the System Development Cycle in Figure 2.4. It espouses an obvious cyclical development paradigm, but the diagram does not do this method justice. It was born out of rebellion. It was created by a group of men who coined the Manifesto for Agile Software Development[5] in response to the faulty implementation of all the processes that went before and to the choices made by those processes. The manifesto speaks for itself.

Manifesto for Agile Software Development:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and Interactions over processes and tools
- Working Software over comprehensive documentation
• Customer Collaboration over contract negotiations
• Responding to Change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Each of the four items listed in the manifesto are a direct attack against the traditional, top-down execution of programs. Software processes, tools, stringent documentation, unchanging requirements, and plans that are followed regardless of reality are all assailed. At its core, Agile methods rely on cooperative individuals, good coding techniques, open dialog with a customer, and flexibility to make responsible changes in the schedule or in the product all with the goal of achieving a working product in minimum time with minimal cost without overworking or overstressing the employees.

One implementation of Agile development is called the scrum[38]. Scrum is an incremental product development method that utilizes one or more small groups of often cross-functional dedicated people who self-organize to collaboratively resolve a given problem. In this context, dedicated means that the individuals on the team do not have roles outside the team. It generally consists of fixed-length iterations, called Sprints, which are usually only a week to a month in length which generally try to achieve a working software solution at the end of each Sprint. The team is self-organizing and self-managed and should have the autonomy as to how to reach its commitments. A Product Owner, who may or may not be a member of the scrum team, has the ultimate responsibility for all decisions with regard to the overall product under development. Finally, the ScrumMaster facilitates the scrum activities and artifacts. Most scrum implementations use a short, 15 minute or less, team meeting to bring all the actors together. This meeting is led by the ScrumMaster, who has no management authority over the team, with the expressed purpose of removing impediments to progress and capturing empirical data to adjust forecasts. One critical aspect of the scrum is that the development team arranges its own schedule and makes a commitment to get the chosen list of tasks done within the sprint duration. The Team then has autonomy to
adjust the daily assignments or individual item schedules to support the accomplishment of all assigned tasks within the sprint time limit. The Team is responsible to the ScrumMaster and the Team members to each other. A Sprint Planning Meeting is held at the beginning of the sprint to establish the list of items that need to be finished in the sprint. A Sprint Review Meeting is held at the end of each sprint to demonstrate a working product to the Product Owner and other interested parties. This should not be a presentation, nor should it be a schedule meeting to include unfinished items in the next sprint. It really has to be a meaningful demonstration of one or more planned and proven features.

The use of Agile methods are currently popular in Software Engineering. One solid plus for this methodology is that it seems to be software centric. The daily meetings actually discuss the code development and functionality. Tasks are broken into smaller simpler tasks that can be completed in a building-block fashion in approximately one month intervals. Yet, somebody still has to do all the documentation required for a defense program. There is also an issue with baselines, as generally implemented in defense programs. Changes to the product are generally matters to be handled lawyer-to-lawyer through contractual vehicles. Historically these take months to work out, not hours, days, or weeks, as would be necessary to support the scrum structure. Also, DOD programs, by their very nature, are top-down. In order for Agile to work, the government oversight personnel are going to have to be given significantly more flexibility in the execution of the contract than currently available. Finally, the idea of self-organized team members in the type of companies that generally win defense contracts is far from the norm. Sadly, the scrums and sprints in which I had the pleasure to work were nothing short of daily bloodbaths. The Software Manager, the defacto ScrumMaster, would brandish harsh words toward any individual who was, for whatever reason, unable to produce as promised. There was no mentoring. There was no Team responsibility, just an open task indicating the failure of one individual. This absolutely ignores the respect of each individual which is a hallmark of this development method.
2.7. Software Development Paradigm Assessment

Process is to software as a recipe is to food. The recipe is not the product. It is not the goal. The creation of good food is the goal in cooking, just as the product is the goal in engineering. I have a recipe for Grandma Burke’s banana cake and I watched her bake that cake dozens of times. Despite the recipe and the experience, I have never made banana cake like she did. Apparently there is more to being a good cook than just following a recipe. Likewise, there is more to being a good software engineer than just following a process.

All of the processes and development paradigms discussed in this chapter have strengths and weaknesses, but they all are designed to create good, working software which is compliant with its requirements. They all are designed to start with human thought, an idea which represents a solution to the problem at hand, and to systematically guide that idea through a series of steps which results in an exact replica of that thought in code. Unfortunately, in my humble opinion and based upon thirty years of work in the industry, the process becomes the focus of software development and the generated code must endure endless testing in a futile attempt to ensure correctness. Since exhaustive testing is nearly impossible, however, some errors make it to the field. Thus, the emphasis on process over engineering allows unintentional errors to infect the actual software product only to be uncovered through in-house testing, or worse, upon release to the customer. The software staff was not properly sized to handle both the process support tasks and the design, development, and testing of their own assigned software. Ironically, the process actually forced the finding of errors by declaring that, statistically speaking, no review of software could possibly yield zero errors. Once an error was identified, a lengthy analysis process was required to determine the source of the error, the corrective action for the error, and the suggested change to the process to avoid such errors in the future. Finding real errors of sufficient criticality to warrant all these procedural improvement tasks takes significant time on its own.

Getting eyes on the code via reviews is a critical requirement, but just as important is
the need to get the right eyes on the code. Software systems are complex and getting more so with each passing day. The systems can, and should, be simplified to the maximum extent possible by breaking the code into minimally sized units with a well-defined written interfaces. On the good side, such a breakdown can allow a developer to apply full attention to the creation of the assigned software with full knowledge that it will be fully functional within the context of the full system as long as the defined interface is realized. On the down side, the hidden details of the operation of the full system can allow a well-formed software unit to fail due to an ill-formed specification. Thus, reviews of the code by individuals responsible for the operation of larger segments of the software are absolutely critical. These people, however, are precisely the staff members whose time is the most constrained by process tasking. They tend to be over-tasked, spend too much time in meetings, and become overwhelmed. Available time for code reviews is measured in minutes, not hours. That is how errors pass right through to testing. Under these conditions, errors of commission and omission are injected in every step of the overly aggressive process being executed with an overly aggressive schedule by an overly worked, and generally under-trained, staff.

The emperor has no clothes! The government paid the Software Engineering Institute (SEI) to develop CMM/CMMI and SEI has the sole rights to administer the CMM/CMMI Program and to assess contracting companies on their compliance with the Process Areas. Obviously, the SEI has a vested interest in making the program successful both on behalf of the government and on behalf of the contractors. The government finances SEI to continuously improve the CMMI program and the contractor companies pay to receive SCAPMI assessment for the assignment of CMMI levels. Nobody is willing to stand up and say, “The Emperor has no clothes!” Now that the government no longer requires the use of CMMI, one would think that this situation could get better, but the development programs are still being run by the same people.

The sad truth, to date, is that no process, no tool, no innovation has corrected the
overwhelming tendency of software to be late, over-budget, and error-prone. As evidence that software engineers do have a sense of humor, there is a saying that was coined in 1985 and attributed to Tom Cargill of Bell Labs in an article by John Bentley. He called it The Rule of Credibility and it stated that “The first 90 percent of code accounts for 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time.” I heard some variation of this quote spoken on every program on which I worked in my career. Though uttered in jest, it was a sad reality and a scathing embarrassment to the software industry in general. It is my humble opinion that the cause of this reality is based both in the failed process execution detailed in these pages and in the very nature of the human spirit. Software engineers are intelligent individuals who take pride in their work. Nobody wants to be in trouble. Nobody wants to fail. So, as programming schedules began to slip, creative minds did whatever could be done to feign the appearance that everything was fine. Future work, usually easy future work, could be quietly pulled forward and completed to fill the gaps left by the hard problems which were now late. Thus, the numbers reported to higher management levels appeared good and there were no beatings. The hope was that a solution for these problems would miraculously appear in time to cover the tracks later, but miracles rarely occur in software. At some point, maybe that mythical 90 percent point that Tom Cargill made famous, the truth had to come out and the software organization would be on the hot seat explaining massive schedule delays and cost overruns to management. Yet, delaying the truth until the last possible moment delayed the inevitable “beatings.” Would it not be simpler to have open, non-threatening dialog about problems when they first appear and resolve them with the full knowledge and support of the entire development team instead of allowing the failure to be swept under the rug until it was too late to fix it within the confines of the available schedule? I sincerely believe that such a cooperative environment could resolve most issues and not allow them to fester into long-term problems.
The goal of Hy-C is simple. It is a tool which takes existing code, that is assumed to be tested and error free, and retargets that code for a defined set of existing processors. In doing so, it deals only with the code and avoids the significant time and cost required to retarget software to new technologies in any of the traditional manners described above. While Hy-C is only applicable to a very narrow range of domains, the domain of embedded military applications is most assuredly among them. Retargetable compilation has long been a desire of the U.S. military. Most military-related computing equipment must be fast, energy efficient, and mobile (i.e. small and light), the same characteristics previously identified for modern hand-held mobile applications. Thus, Hy-C, in its simplicity of use, should make it an indispensable tool in software engineering for both of these very lucrative and important domains. Furthermore, it could very possibly reduce both risk and errors by retargeting existing tested code on a new platform.
CHAPTER 3

RETARGETABLE COMPILERS

3.1. History

The concept of a retargetable compiler is not new. The United States military has had an interest in such systems for decades. Military systems are, by their very nature, embedded systems, incorporating state of the art components for specialized purposes. Once completed, some of these systems can realistically be expected to sit on a shelf or exist in standby conditions for decades at a time. Yet, technology forges ahead and as the original cache of spare parts is depleted long before the useful life of the fielded system comes to an end, very difficult and costly decisions must be made. Either the military must pay the exorbitant price required to manufacture another lot of the original parts, if that is even possible, or incur the development cost of system modifications to incorporate newer technology. In the case of upgrades, both software and hardware must generally be modified. Imagine the cost savings that those system upgrades could realize if it were possible to simply recompile and test the legacy software for the new hardware platform! In addition to the military need, an even more pressing issue in recent years has been the explosion of hand-held devices requiring a constant flow of technological improvements on an ever-shrinking area of silicone. Ultimately, these are some of the major triggers driving the need for a retargetable compiler, but no such cost-effective, generic retargetable compiler exists, to date.

So, what makes a compiler retargetable? “A compiler for a fixed programming language is retargetable, if it can be adapted, so as to generate machine code for any processor within a defined class of processors, in such a way that the largest part of the compiler source code is retained.” [45] In other words, a compiler is considered a retargetable compiler if the majority of the legacy source code does not have to be rewritten for each new target processor.
Most, if not all, computer science students and software engineers have been using a retargetable compiler without even realizing it. The GNU GCC compiler is a retargetable compiler that supports 41 different architectures, at last count[16]. Along with being one of the most widely distributed and free software tools, it is living proof that software engineers do have a sense of humor. GNU is an acronym, of sorts, which stands for “GNU’s Not Unix.” GCC is a recursive acronym which originally stood for “GNU C Compiler,” but ever since 2001, the meaning has changed to “GNU Compiler Collection” in response to the large number of source code compilers now available under the GNU umbrella. Instructions for retargeting GCC can be found in the online documents provided by the GNU organization[16].

The general compilation process is depicted in Figure 3.1. Compilers employ a front end to comprehend the source code language and to define the rules by which the Intermediate Representation (IR) is synthesized. The IR is manipulated by optimization algorithms.
which are considered to be machine-independent. A back end then applies machine-specific optimizations, makes register assignments, and selects the appropriate processor microcode to complete the process, culminating in the machine code for the target processor or processor SoC. Obviously, this description does not do justice to the complex operation of a compiler, but the simple description is sufficient for this work.

Making a compiler retargetable can be accomplished by adding a machine description (MD) to the front end and substituting machine-specific modules for some of the general purpose back end components shown in Figure 1.2. If the compiler is being retargeted to an existing COTS CPU, the back end of the vendor-supplied compilers for the CPU can be ported directly into the retargetable compiler. The use of strict modular programming techniques in the creation of the retargetable compiler greatly simplifies the porting of this code. This streamlined development of the compiler saves both time and cost.

In a similar manner, substitutions and additions can modify a retargetable compiler into a heterogeneous retargetable compiler. Once again, a good modular design should allow the relatively simple addition of multiple COTS processors into the heterogeneous compiler using the native compilers supplied with the additional CPUs. The heterogeneous version of the compiler will require a couple additional modules to facilitate the distribution of the code across the multiple computing devices. As a minimum, it requires a module to perform the partitioning of the code and additional information in the MD to provide guidance as to how to partition the code. Generally speaking, this additional information provides some sort of quantitative description of the processing characteristics of the CPUs. If the heterogeneous compiler is used in hardware/software co-design and DSE, it is also desirable to add “hooks” in the software that will allow the collection of whatever metrics are determined necessary to differentiate the processing capabilities of the expected range of hardware configurations under study. This data could be critical in the decision process to select a final heterogeneous configuration.
3.2. Examples

Retargetable compilation can take many forms. Leupers and Marwedel [46] describe at least 36 different retargetable compilers. While each retargetable compiler is unique, they all fall into one or both of two broad categories of retargetable compilers. A compiler can retarget at either the instruction set level or at the micro-architecture level, or both. Instruction level retargeting simply provides a translation of code written for one target processor to executable code for another target processor. Micro-architecture retargeting, on the other hand, includes the translation feature in addition to lower-level processes involving timing, pipelining, and other techniques specifically designed to increase the overall processing performance of the executed code.

One development system that deserves some consideration in this discussion is OpenCL™ because of its acceptance in industry and the fact that it shares some traits with retargetable compilers. “OpenCL is a royalty-free standard for cross-platform, parallel programming of modern processors[6].” It was originally developed by Apple, but was then transferred to the Khronos Group for release as an open standard. OpenCL provides expert system developers with a framework in which computing devices can be described and accessed via a defined application programming interface (API). As such, it is a programming language, of sorts, not a retargetable compiler. This may be an excellent tool for the manual parallelization of a large software program or system, but it does not automatically redirect executable code to any devices. It only responds to the directions provided by the programmers through the API. In addition, it does not automatically extract parallelism and efficiently spread that execution across constituent computing resources, as is the defined purpose of Hy-C.

Descriptions of the architecture or machine descriptions for 4 retargetable compilers are provided below.
(define_insn "subsf3"
  [(set (match_operand:SF 0 "register_operand" "=f")
       (minus:SF (match_operand:SF 1 "register_operand" "f")
                 (match_operand:SF 2 "register_operand" "f")))]
  ""
  "subf\t%0,%1,%2")

Figure 3.2. Example GCC Instruction Machine Description (subsf3)

3.2.1. GNU GCC

As mentioned earlier, the GNU GCC is a retargetable compile in widespread use throughout the software industry. Users are free to use and modify the compiler under the GNU Public License, which, in Machine Description (MD) nutshell, allows free use, modification, and distribution of the compiler with the stipulation that the modified source code must be freely available to all users. The machine description consists of three files[46]:

- The Machine Description file (MD)
- A "C" header file which contains macro files
- "C" source code files with processor-specific support routines

While all three files are required for the required machine description, only the MD is germane to this discussion. The MD file provides the compiler with the available instruction set by mapping GNU Compiler Collection (GCC) IR to the target assembly code. Most definitions consist of a name, a register transfer language (RTL) template, an assembly output template, and matching constraints. Along with instructions, the MD can also define complex instructions, registers, and other hardware details. GCC generally accomplishes retargeting at the instruction level, though the ability to manipulate both registers and hardware details certainly borders on micro-architecture retargeting. The syntax of the modeling language for the MD is very complex. This complexity will be demonstrated with two example instruction descriptions for GCC, along with syntactical explanations for each.

Leupers and Marwedel[46] provide the example shown in Figure 3.2. Learning any new computer language can be difficult, but little in this example would be even remotely
familiar to the average software engineer. This structure is a GCC instruction definition for the subtraction operation (sub) for single precision float (sf) numbers. It further specifies that the operation requires three operands ("3", the end character of the instruction name and each uniquely identified as 0, 1, and 2). All three of the operands in this example are required to be registers and, because of the "f" designator following the register, all must be floating type registers. Register_operand 0 has the additional restriction of being write-only, as designated by the equal sign ("="). Finally, the definition ends with the output template, which is in the form of an assembly instruction with "subf" serving as the instruction name and the three parameter registers.

Once again, the definition in Figure 3.2 provides the necessary information for the subtraction function between single precision float numbers (sf) only. Additional definitions for subtraction would have to be generated for each of the precision modes available in the target processor. GCC defines a total of 52 different modes. Subtraction is not a valid operation for most of those modes, but it certainly would have to be covered for at least some proper subset of the following additional modes, assuming they are applicable to the target processor:

- 1 byte (QF) Quarter floating
- 2 bytes (HF) Half floating
- 3 bytes (TQF) Three-quarter floating
- 8 bytes (DF) Double Floating
- 10 bytes (XF) Extended Floating

Admittedly, the definition for the single precision float subtraction case could, to some degree, serve as a template to create the other variants of the subtraction family of operations, but swapping out the parameters for each individual case would still require a significant amount of knowledge of the both GCC syntax and the target processor and memory architectures. Lin and Chen actually exploited the similarity in the families of instructions to build
an assistance tool to systematically guide a user through the GCC retargeting process. They make the point that “traditionally, developers wanting to retarget GCC to a new target architecture have referenced and refined the available MD files that apply to similar architectures. However, this makes GCC retargeting a trial-and-error procedure.”[47] They then describe an assistance tool that can guide a user through their GCC retargeting methodology. The user needs to be well grounded in the target architecture, however, because the tool can only indicate to the user where target-specific attributes need to be substituted, not what those attributes might be. It provides the user with a template that will work for a family of target processor commands. The user must substitute highlighted template terms with syntactically correct target machine instructions.

An example template from [47] is provided in Figure 3.3. It actually shows the templates for two different, but related, machine instructions, jump and indirect_jump. In this example, the only thing that needs to be replaced with target-processor code are the statements on line 5 (“output assembly code of jump”) and 13 (“output assembly code of indirect jump”). Once again, this helps simplify the retargeting by leading the user through a defined process, but still requires a very knowledgeable user to know the syntax and content of the necessary target assembly code.
Abhijat Vicharl and I.I.T. Bombay prepared a full example of the GCC retargeting process, Writing GCC Machine Descriptions, and provided free access under the GNU Free Documentation License for what they call the Toy Processor[63]. Toy only provides two single integer arithmetic operators (add and multiply) plus a handful of support operators (and, or, not, band, bor, bnot, jump, load, wload, store, and wstore), but it clearly illustrates the depth of knowledge required to implement a new target in GCC. This step by step guide is perfect for this discussion, however, specific use of its contents would require adherence to the full GNU Free Documentation License for this dissertation, making it an impractical choice. Still, it would be of great utility to any number of academic classes in Computer Science.

While the full capability of GCC, in all its complexity, allows a high degree of flexibility for the creation of novel systems, it also limits the manipulation of the MD to a very elite level of programmer familiar with the intricacies of both the MD syntax and the target machine architecture. Benchmark studies have demonstrated that the quality of the code generated by GCC for regular reduced instruction set computer (RISC) and complex instruction set computer (CISC) general purpose processors with a byte-addressable memory is generally acceptable. The same cannot be said, however for applications outside this niche, for example, in the area of embedded DSPs. In this realm, a performance overhead of 400% or more, as compared with hand-written assembly code written by expert domain-experienced programmers, is not unusual[46].

3.2.2. Production-Quality-Compiler Compiler

The Production-Quality-Compiler Compiler (PQCC) research project at Carnegie Mellon University (CMU) was headed by R.G.G. Cattell. The design target for this compiler in the early 1990s was the fast and efficient backend processing (from IR to target machine code)[31]. The machine model for this product views the machine as an “instruction set processor.” The machine definition consisted of 7 components:
Machine operations
- Data types
- Storage bases
- Storage access models
- Operand classes
- Instruction fields
- Instruction formats

Each instruction in PQCC was represented as a tree expression with an input assertion and a corresponding output assertion. The tree representation was especially efficient for the code generation phase since a simple “tree matching” algorithm could be used to traverse the source code tree to match the instruction patterns to determine the required code sequences.

While PQCC never became a commercial success, in and of its own accord, it did form the basis, in part, for the compiler work at Tartan Labs, which was founded by Bill Wulf, one of Cattell’s associates at CMU.

3.2.3. LISATek

Not all academic research translates into a commercially viable product, but LISATek[46], the brain child of Peter Marwedel and Ranier Leupers at RWTH Aachen University of Technology in Germany, is certainly one of the the commercial success stories in the field of re-targetable compilers. LISATek is a complete re-targetable compilation system serving as an electronic design automation (EDA) tool which was spawned by their research on the Language for Instruction Set Architectures (LISA) language for re-targetable compilers. Developed within academia, LISATek can generate all tools and models necessary for software development and for the integration and verification of embedded processing devices within a SoC environment. The generated tools include high-performance simulation models, target assemblers, linkers and synthesizable RTL code[11]. As an added feature, it even produces (Very High Speed Integrated Circuit) Hardware Description Language (VHDL) and Verilog.
code that can be ported directly for the synthesis of ASICs. The system uses LISA to provide a complete description of the full Instruction Set Architecture (ISA) of the proposed target system, but it also acts at a micro-architecture retargeting level. After years of academic research on LISA and associated products, spawning scores of academic papers, the LISATek system was sold to CoWare in 2003 for commercial productization and distribution.

One very important technology advancement included in LISATek is that it allows for cycle-accurate modeling, even for pipelined architectures. This detailed accuracy provides a capability to investigate and quantify the detailed performance potential of a viable or proposed DSP for time-sensitive processing systems. In addition, it “includes elements, which facilitates the language use as well as generation of fast compiled simulators[46].” LISATek embeds all of the tools into an integrated graphical user interface (GUI) environment and supports the direct translation of LISA models to synthesizable VHDL and Verilog RTL models. This capability greatly increases the efficiency of the generation of the modeled hardware. It allows the simulation of the system to operate either in an instruction-accurate model for early development, or in a more fine-grained cycle-accurate mode, which includes detailed pipeline timing simulation[19].

Of course, such detailed results from the LISATek model requires the inclusion of commensurately detailed modeling data into the model. Figure 3.4 shows a LISA specification for arithmetic expressions where only addition of numbers, not variables, is allowed[49]. This level of complexity for the instruction descriptions along with the need for low-level syntactical knowledge of the LISA language itself almost guarantees the need for a significant staff of electrical engineers, systems engineers, and software engineers dedicated to the navigation of the intricacies of LISATek, thus negatively impacting its cost effectiveness. Furthermore, at least one published evaluation of LISATek noted its lack of automation, model description errors, and VHDL generation errors as significant weaknesses. In addition, the same publication found that the products generated by LISATek were bound to the specific mod-
Figure 3.4. LISA Example Description for Language Expr

eled architectures and were “not suitable for performance estimation, in general [51].” After spawning numerous research papers and books, this product moved to the commercial world, but has not has not gained any widespread notoriety in this arena.

3.2.4. Horizon

The Horizon Retargetable Compiler is the result of a research project at the University of Colorado and was authored by Robert Mueller, Michael Duda, Philip Sweany, and Jack Waliki in the 1980s. The expressed goal of the Horizon Project was to create a compiler that could simultaneously produce highly optimized microcode and retargetability across a variety of processor types focusing on resource usage as the primary issue [50]. Horizontal architectures use control words of variable length to produce as much parallelized processing
as possible by packing micro instructions into a single control word, all to be performed in a single clock cycle. As such, Horizon retargets at both the instruction architecture level and the micro-architecture level.

Since resource usage is the primary consideration, the character of the target machine description for Horizon is quite a bit different than that for other retargetable compilers. It uses a target machine description that provides the following:

- Resource setup and hold time
- Control word field encodings
- Machine microoperations
- Data dependency graphs
- Target machine data paths

Horizon has been retargeted to three different microarchitectures, a pipelined signal processor, and several commercial graphics processors with enough success to “permit its use in rapid architecture prototyping and evaluation during the early architecture phases[50].”
3.3. Hy-C

Hy-C is a retargetable compilation system which retargets strictly at the micro-architecture level, since it uses native compilers of constituent processing resources to produce the necessary executable code for the assigned code segments. The specific implementation under development combines one or more general purpose CPUs with an FPGA fabric, as shown in Figure 3.5. Later enhancements will include the addition an ASIC. One very likely target may be the configuration of the Texas Instruments OMAP™ hybrid processor which includes a single ARM Cortex-A8 processor (StrongARM), a TI-64X DSP (C64x), and an ARM hardware accelerator (WimpyARM). Descriptions of each of the computing resources are provided in the following subsections.

3.3.1. System Specification

The System Specification is a user-supplied file which contains the architectural description of the computing resources and memory that allows the Hy-C system to be classified as a retargetable compiler. The architecture descriptions for new processors, sometimes called MDs, associated with most retargetable compilers is far from simple, as described above for GCC and LISA T ek. Each requires a significant investment in terms of cost, staff, and time to complete the retargeting. Developing the architecture description requirements for these systems is a daunting task. It requires in depth knowledge of general processor architectures and syntax to support the descriptions of the behaviors of each supported operation. This complexity and its associated cost prevent the routine retargeting of these systems. This is, perhaps, what has kept LISATek from becoming a booming technology in the electronics industry. The complexity is, unfortunately, an unavoidable by-product of retargeting to the machine code level for the new machines.

One premise of Hy-C is that not all retargetable compilers need to retarget new processors. By using existing CPUs in its architecture, Hy-C can pass source code directly to existing native compilers for those blocks of code partitioned to that device. While this limits
the utility of Hy-C to a subset of possible heterogeneous processors, this is exactly the type of processors that are being considered for large-base consumer lines, like cell phones and tablets. In addition, the technology holds great promise for military and government systems. In times past, government contracts have pushed the cutting edge of technology to maintain a technological edge over adversaries. While this investment in technology has triggered some of the most innovative and capable weapon systems on the planet, their development has come at an exorbitant price. Today’s budget for new military systems can no longer support this type of risk nor open-ended budget in development and the government insists, instead, on building system with minimal risk using already proven technologies. There is an implication in this new paradigm that looks at reusing hardware while creating innovation through software. Therefore, a retargetable compiler like Hy-C in concert with one or more heterogeneous processors is a perfect match to provide both commercial and military systems with innovative technologies with minimal risk and greater performance than can ever be realized with a single-single processor solution. Thus, the domain is certainly not universal, but the payoff is potentially enormous! If it makes retargeting to those platforms very easy and cost-effective, it will translate into commercial success. The approach with Hy-C is to keep retargeting simple. The Partitioning Function passes functional blocks of code to those native compilers to produce the necessary machine code for execution on the associated processor. In a like manner, Partitioning passes functional blocks of code to the FPGA Coder to synthesize executable code for those devices. The full description of the Hy-C architecture description file is presented in Chapter 4.

To support this simplified version of retargeting, the Hy-C architecture description file simply needs to provide execution time, delay time, and energy consumption parameters for each computer operator, for each processor, and for each type of memory. The Partitioning Function will use these data to determine the most efficient routing for each functional block of code as defined by the System Objectives and Constraints. While determining the time and
energy requirements may not be trivial, it is not nearly as difficult or error prone as generically describing the full processing behaviors of each instruction.

The other task of the Partitioning Function is to provide synchronization code to the Program Controller to ensure the proper sequencing of the code, start to finish, in accordance with the intended sequencing of the original source code. The result of this partitioning will be the achievement of the optimal parallel execution for the input source code.

3.3.2. Partitioning

A compiler that partitions computation among a hybrid chip’s heterogeneous processing elements provides a major component of our code-generation strategy. The partitioning of tasks among multiple dependent resources is an NP-complete problem[34]. However, researchers have successfully employed many heuristics to obtain solutions, in reasonable time, for specific instances of the problem. In the case of Hy-C, tasks must be partitioned among heterogeneous resources on-chip. This requirement, along with other constraints, makes the partitioning compiler difficult to design. However, in addition to using algorithms and techniques employed in hardware/software co-design, Hy-C uses the design, algorithms, and code of existing compilers, namely Rocket[59] and Low Level Virtual Machine (LLVM)[44].

The hardware/software co-design community has a long history of partitioning computation among heterogeneous resources. However, according to Brandolesse et al. [29], most suffer from one or both of two problems. Either they focus on too narrow an application area or they require too much “pre-design” by domain experts, which often leads to the same problem, i.e. too narrow a focus for a wide range of applications. In fact, much of the hardware/software literature describes different heuristics used to find acceptable solutions in a vast search space. Similar approaches to finding acceptable solutions to NP-complete problems, such as instruction scheduling and register partitioning relying on genetic algorithms [25][24][57] and simulated annealing[58] have been used.

However, for this particular partitioning problem, we have had good success with an
algorithm that builds Data Dependence Graphs (DDGs)[20] for each function of the C code and determines which available computing resource best fits the parallelism available in that DDG. Parallelism, in this case, is defined as the number of nodes in the DDG divided by the time-sensitive critical path of the DDG.

3.3.3. CPU Compiler

A hallmark of Hy-C is that it has been kept as simple as possible for the user. By utilizing off-the-shelf processors for which commercial compilers are readily available, Hy-C only requires that the user provide an estimate of the processing time and heat generation characteristics for use in extracting execution time and power-usage efficiency in the compilation process. Other retargetable compilers, GCC and LISATek, for example, require large staffs of knowledgeable engineers along with a large budget and ample schedule time to choreograph the retargeting of a new processor. This is not the efficiency model required by successful telecom companies in this fast-changing industry!

In terms of speed and thermal efficiency, the CPU is assumed to be the slowest processor and it will probably produce the greatest amount of heat for a given execution of source code.

3.3.4. FPGA Code

Generating on-the-fly, application specific FPGA code is certainly a hard problem, but solutions for converting code written in C to VHDL for execution in FPGAs, which could simplify this problem, are currently being studied[27]. In the short term, however, the initial version of Hy-C will employ the use of some predetermined functions whose execution on FPGAs are known to be efficient. These functions will be hand-coded using VHDL, Verilog, or C (along with a VHDL/Verilog converter) to produce the FPGA “code.” Standard compiler techniques will be used to build a representative DDG describing the code sequence that can be assigned to the FPGA. The Partitioning Function will then use subgraph matching
techniques to choose source code that will be directed to the FPGA for execution. Later versions of Hy-C will tackle the custom-generated FPGA code solution.

At this point, the main impact of FPGA code is that it will allow more parallelism than is inherent in any fixed CPU processor. As an example, consider matrix multiplication. Since the innermost loop contains no loop-carried dependence, the only limit to parallelism is the number of multiply-accumulate functions that can be started at any point in time. Texas Instruments family of DSP chips allow up to eight distinct operations to complete in one cycle. In theory, these chips allow for reading four source operands (for two loop iteration’s multiply-accumulates), two multiply-accumulates for different loop iterations, and for two writes of values for yet two more loop iterations, completing two inner-loop iterations per cycle. Is it possible to do better? For a fixed CPU, it is only possible to do better if more functional units are available.

In contrast, consider using a hybrid architecture’s FPGA to perform multiply-accumulates in parallel. Instead of being limited by a fixed processor’s hardware, it would be possible to compute as many inner loop iterations in parallel as we can program in the FPGA. So the question of interest becomes: With the available FPGA resources, how many parallel sets of two loads (for input values), one multiply-accumulate, and one store can be executed in parallel?

To answer this, the results of a matrix multiply in which the innermost loop was pipelined in FPGA was compared to the results of one executed in a CPU. An assumption was made that all data be prefetched into FPGA (Xilinx Vertex5 XC5VLX220, the most powerful FPGA available at the time of the study). The process to compute matrix \( c[i][j] \) was parallelized and pipelined so that it could be completed in one clock cycle. The results are summarized in Table 3.1.

Calculations were limited at \( N=32 \) since a size of 64 would have exhausted the resources of the FPGA being used. Still, it is apparent that using the FPGA produced an
improvement in inner loop performance by a factor of 25. And with significant improvement in size, speed, and programmability of FPGAs recently, the improvement would be much greater.

3.3.5. ASIC Code

An ASIC is shown in Figure 1.2, but the initial Hy-C will not actually support ASICS. In general, FPGAs and ASICS have comparable performance characteristics. The choice of one over the other is largely a matter of development budget, schedule, and deadlines. Unlike FPGAs, which can be reprogrammed time and time again, ASICS are manufactured with the gate logic hard coded in the design of the device. In other words, they are not programmable at all after manufacture. There is an expensive non-recurring engineering cost associated with ASICS to manufacture the proper logic into the logic gates. This takes both time and significant cost. The benefit, however, is that the manufacture of the device is considerably cheaper than that of an FPGA. As such, the non-recurring cost of the up front engineering can be negated by the savings in manufacturing if a sufficient quantity of devices is needed to affect this saving. On the other hand, once the device is manufactured, there can be no modification without another investment in the development cost of a new ASIC. Therefore, system designers must consider all of these factors in determining whether or not to include ASICS in the final heterogeneous processor.

3.3.6. Power/Performance Modules

The three Power/Performance modules are depicted in Figure 1.2, one for each constituent processor. The purpose of these modules is to gather performance metrics for each

<table>
<thead>
<tr>
<th></th>
<th>N=2</th>
<th>N=4</th>
<th>N=8</th>
<th>N=16</th>
<th>N=32</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (ns)</td>
<td>110</td>
<td>875</td>
<td>5,500</td>
<td>42,000</td>
<td>350,000</td>
</tr>
<tr>
<td>FPGA(ns) = n^2 \times \text{clock} = FPGA(ns)</td>
<td>28</td>
<td>109</td>
<td>635</td>
<td>2,549</td>
<td>13,865</td>
</tr>
</tbody>
</table>

Table 3.1. Matrix Multiply Comparison
of the respective devices for analysis by the Optimization Control module. The metrics collected must include such things as the execution time of code on the device and the power consumed by the device in the execution of the assigned software. Other metrics may be required to satisfy specific customer needs identified in the Objectives and Constraints file.

3.3.7. Optimization Control

The Optimization Control must evaluate the collective performance data produced by Hy-C from the processing of a run of user-supplied source code. The data generated by the Optimization Control Function can then be used to modify the partitioning algorithms or, possibly, to provide evidence that a different hardware configuration may be more efficient. The data might also suggest that the user change the objectives and/or constraints. If, for example, the run indicates that one device is using an excessive amount of power, thus generating excessive heat, the partitioning algorithm might have to change to favor reduced power consumption over processing time performance.

3.4. Hardware/Software CoDesign

No discussion of retargetable compilers would be complete without at least a cursory discussion of Hardware/Software Co-Design because this is the development phase in which a retargetable compilers are particularly useful. As mentioned earlier, however, there are many types of retargetable compilers and the Hy-C retargetable compiler, which is a critical component in this dissertation, is an exception to the rule. Since Hy-C simply distributes software application code across a number of static hardware components, there is no need for hardware design at all. It is not difficult, however, to envision a time in the future when a fully operational Hy-C retargetable compiler could be used within the context of the Hardware/Software CoDesign process to evaluate the processing of a variety of candidate sets of hardware to definitively identify the most efficient configuration.

The proliferation of hand-held personal electronic devices dictates a quick time-to-market development process. At the same time, the rate of hardware technological advances
makes pure hardware solutions obsolete within a matter of a few years, as evidenced by the rapid evolution of cell phones from 2G to 3G to 4G and now 5G, with new product baselines doubtless to follow. But, according to DeMicheli and Gupta:

The trend toward smaller mask-level geometries leads to higher integration and higher cost of fabrication, hence the need of amortizing hardware design over large production volumes. This suggests the idea of using software as a means of differentiating products based on the same hardware platform. Due to the complexity of hardware and software, their reuse is often key to commercial profitability. Thus, complex macrocells implementing instruction-set processors (ISPs) are now made available as processor cores[33].

In other words, powerful retargetable compilers that generate efficient code are needed in this short cycled development arena in order to get products on the shelves for hungry consumers before the target architectures are made obsolete by the next generation of electronic components. Typical two or three year development cycles of the past are incapable of keeping a product at the cutting edge of technology. The systems today are orders of magnitude larger and more complex than any that have preceded them. Their development demands a new paradigm, one designed to mitigate problems associated with that complexity. The process of hardware/software co-design allows for just such the concurrent development of both, by explicitly considering the trade-offs between implementation technologies and the interaction between components of different types. It is a perfect match for the use of an agile, cyclical development process which allows the incremental development of critical software elements concurrently with the associated hardware components in spirals of ever-increasing capability. If the schedule is perfectly tuned, the software will be developed concurrently with a skeleton version of the hardware prototype allowing the concurrent testing and verification of a system-level solution before finalization and manufacturing of the production hardware. At the same time, the next iteration of both can be designed and moved to full implementation if, and only if, testing of the prior cycle validates the specified system-level operability for the cycle.
CHAPTER 4

ARCHITECTURE DESCRIPTION ESSENTIAL ELEMENTS

Architecture descriptions for traditional retargetable compilers are extremely complex. Some of the examples in Chapter 3 provided a very small window into the complexity of the descriptions necessary for each and every instruction for each included device. The examples chosen for inclusion were the simplest examples available, but it is a stretch of one's imagination to think that any novice would consider them simple. While I espouse employing the most simple solution possible in all situations, there is currently no relief available for this complexity when dealing with full compiler functionality. Computers can only do what they are programmed to do, so detailed instruction translations are a necessary evil. “Despite efforts to make compilers easier to retarget, it is still a task that requires an expert both in the compiler project and target architecture domains ... One would expect it to be easy to retarget a compiler, since the software team just needs to write a new back end for an existing compiler project. But, in practice, this task requires developers to understand not only the target processors, but also the compiler API, which is a considerable effort and is likely to add a prohibitive delay into the design exploration phase.[22]” As previously discussed, tools to automate this process are being created, tested, and used with varying degrees of success right now, with more on the horizon. Until the miracle occurs, however, we must muddle through with whatever is available today.

The examples in Chapter 3, combined with additional text in the cited references, provide a fairly extensive compilation of the elements required in traditional retargetable compiler architecture/machine descriptions. In general, the description files must provide the compiler with all the processor, register, and memory physical configurations in addition to a description of the manner in which those devices communicate.

Special purpose retargetable compilers, like Hy-C, however, can drastically change the constituent elements of the associated architecture description. Since the components
of the Hy-C system are off-the-shelf processors, the native compilers are employed in the actual compilation process, negating the need for instruction level retargeting details. The traditional compilers, for all their complexity, do not provide the information necessary to allow Hy-C to perform its critical Partitioning Process. That functionality requires parameters to allow the estimation of processing time, including all expected propagation delays, and power consumption, which should also be a measure of the generated heat that must be dissipated by the device. Minimizing power consumption has a dual benefit of extending available battery life and reducing the harmful effects of heat generation.

Another unique feature of Hy-C is the inclusion of FPGAs as a component for realtime code synthesis. While they have been widely used in industry as stand-alone components of systems, their inclusion as components of a heterogeneous processors on a chip and the direct compilation of high-level source code into formats compatible with the on-the-fly synthesis of FPGA gating instructions is in its infancy. This represents a significant departure from the historical hardware development paradigms and provides a great potential for significant efficiency increases for the processors. The plan is to initially use FPGAs with preprogrammed functions associated with DDGs segments that we expect to find in the application IR. Once this operation is fully understood, Hy-C will transition to full on-the-fly creation of FPGA “instructions.” The inclusion of ASICs is a stretch goal for Hy-C, but not until the use of FPGAs is fully understood. The assumption is that the use of the ASIC resources will mimic those of the FPGAs, but the power consumption and processing time would be significantly less.

A consolidated list of essential architecture description elements is provided below. Those items marked with a single plus sign (+) apply to both traditional retargetable compilers and Hy-C. Those marked with double plus signs(++) apply only to Hy-C. Those with no asterisks at all are only needed in traditional retargetable compilers and are of no use to Hy-C.
+Processors

• For each type
  
  • For each Operator
    
    • +Cycles per execution
    
    • +Energy consumption (predicted energy/thermal characteristics with use)
    
    • +Pipeline stages/offset (how long between start of new pipelined events)
    
    • Target Processor Assembly Language equivalent

+Registers

• Program Counter

• General Purpose [0 .. n]

• Stack Pointer

• Address Register (indirect addressing) [0 .. n]
  
  • Stack Memory [y .. z]

+Memory

• +Type of memory (RAM/ROM/Program/Data/shared/how to move from M1 to M2)
  
  • +Distributed Shared Memory (DSM) or Message-Passing Memory (MPM)
  
  • +Propagation delay (cycles to latch on write) or Read time
  
  • +Read time
  
  • +Energy consumption (predicted energy/thermal characteristics with use, read and write)

+Size

• +Pipeline stages/offset (how long between starts of new pipelined events)

+Communication Between Devices

• +type
• ++Propagation Delay

++ASIC

• ++DDG (the match for the Code Selector)
• ++I/O addressing/handling
• ++Energy consumption (predicted energy/thermal characteristics with use)
• ++Performance estimation (time/clock-cycles)

++FPGA

• ++DDG (the match for the Code Selector)
• ++I/O addressing/handling
• ++Energy consumption (predicted energy/thermal characteristics with use)
• ++Performance estimation (time/clock-cycles)
CHAPTER 5

HY-C ARCHITECTURE DESCRIPTION

The main theme of Hy-C is the KISS principle. It provides a relatively easy mechanism for enabling software engineers at all levels to make use of the advantages of a retargetable compiler for heterogeneous processors on a chip utilizing existing and proven computing resources. While change and innovation can lead to quantum leaps in product capabilities, uninformed change for the sake of change alone can spell doom for a product, or even a product line. Among the most poignant demonstrations of a product failure for no good reason was the debacle of the New Coke rollout in 1985. Coke already held the lead in the soft drink industry, but was concerned about the increasing popularity of Pepsi. The leaders of the Coca-Cola company decided to make a pre-emptive strike and developed New Coke, a slightly sweeter version of the old classic. The plan backfired and New Coke was pulled off the shelves within three months[55]. Change for the sake of change can be risky business. While the soft drink industry is obviously different than the computer industry, they both vie for market shares in the lucrative consumer market. An even more applicable example, previously discussed in Chapter 1, was when a Program on which I worked changed from a single-core to an eight-core processor with the promise of realizing a massive increase in performance. That increased performance, for a number of reasons, never materialized and the change cost the Program both time and money. It is, therefore, fitting to consider the commercial utility of heterogeneous processors on a chip before expending significant investments of time, talent, and treasure on the development of support systems for these devices.

Having established the need for retargetable compilers in concert with heterogeneous processors throughout this dissertation, the final question to be answered is whether or not there is sufficient volume in the niche markets serviced by Hy-C to warrant its continued development. In other words, is there sufficient need for heterogeneous multiprocessor which contain legacy commercially available processors to support the development of a product
like Hy-C? There are several facets to the answer of this question.

First of all, the cellular phone market currently depends on a two-year contract system, which drives the need for newer, faster, and more-connected personal electronic devices every two years, or faster, to keep a customer’s business for the next contract period. While this arrangement could change at any time, it has been the customary term of service for the major cellular phone services for at least the last decade. The bottom line is that nearly everyone I know upgrades their phones when they start a new contract. Some even update them more often. Therefore, having upgraded devices every two years is critical to the survival of a cellular phone company. Unfortunately, this is a very fast turn around for new hardware and software. Anything that can be done to hasten the pace of development of new technology will be valuable to a personal electronic device manufacturer.

Secondly, peripheral devices and other non-CPU, dedicated, special purpose processors are also being developed alongside the processors. The synergistic effect of the addition of a new video device in conjunction with the power of a commercially available DSP for inclusion in an heterogeneous processor could very well provide the additional wow-factor that could define the difference between a customer signing on for two more years or the loss of that customer to a rival provider. Developing novel features for a product line using only existing, proven processors could greatly simplify the process of creating the new capability while minimizing the risks inherent in any new product development. This is precisely the application field for which Hy-C is intended. Hy-C could even go beyond just providing basic compilation services for the new heterogeneous processor. It is designed to actually optimize the execution of the new feature’s software by parallelizing the execution over the full set of available processing resources.

Next, the inclusion of reprogrammable firmware devices, FPGAs, as components to these new heterogeneous processors allows a level of flexibility never before realized in consumer electronics. Each download of software could include algorithmic updates to further
utilize this relatively untapped resource, and their inclusion in a retargetable compiler like Hy-C could exploit their efficiencies in real time.

Finally, any of the reasons listed above to support the development of Hy-C for the personal electronic device market are equally applicable to the global defense market. DOD budgets are not as open-ended as they once were. Today’s military acquisition programs rely on proven capabilities, technologies, and components to reduce costs and minimize the risks of new product development. This market, in and of itself, could justify the development of a product like Hy-C. The combination of the commercial consumer electronic market and the military acquisition market makes the decision to move ahead with Hy-C development an easy decision.

5.1. General Description

So, having described a rationale for both retargetable compilers, in general, and a more simple version like Hy-C, in particular, it is now time to provide the details of the simplified version of an architecture description for Hy-C. The Hy-C architecture description only needs to provide a few parameters about the general architecture of the processor along with parameters for determining execution timing estimates and energy consumption estimates for that execution. The Partitioning Function of Hy-C will use that information to guide the decision as to which of the available processors should be used to execute each functional block of the source code to produce the overall fastest processing and/or most energy-efficient execution of the source code, depending upon guidance from the user-specified Objectives and Constraints file. The content and usage of the architecture description file will be demonstrated with a non-trivial example for a Texas Instruments TMS320C64x DSP.

Before beginning this discussion, however, it is important to understand that the stated purpose of this example architecture description is merely as the first step of the overall Hy-C development. The performance parameters have been set to values with sufficient justification to serve as initial estimates to allow Hy-C to be built and provide a reasonable gross
estimate of relative performance between the constituent processing elements for use by the Partitioning Function. Justification is either supported through direct references to processor documentation or references to other related literature. Once Hy-C can be executed, those estimates will either be validated or modified, in accordance with the collected empirical data. It may even be necessary to substitute defined numbers with representative functions to provide a more detailed estimate. Increasing the fidelity of the processing estimates is listed as future work for this topic.

There are many different ways to represent and create the actual architecture description file or files. Given that the parameters in the file(s) are expected to change over time in accordance with empirical results of actual Hy-C execution cycles, my preference would be to maintain the architecture description as a single, combined, external Microsoft Excel file. Spreadsheets are easy to understand, manipulate, and read into executable programs. After making the necessary changes, and recording the changes for good configuration control, a user can simply write the contents out to a *.csv (comma separated values) file and create an associated function in a program to read the data into whatever variables or tables make sense for the application. While there is some effort to maintain compatibility between the spreadsheet and the input function, my personal experience has been that this technique is very user-friendly. This is especially true when the users, including the subject area experts who provide some or all of the data, are not necessarily software engineers. The technique came in very handy in a lab situation in which realtime data was being monitored. The limit values of a set of variables being monitored on a test set were loaded at the start of a run and the data was continuously monitored and automatically flagged above and/or below specified values. Quite often in an integration scenario, integration was started with all limits open wide, just to get all the pieces and parts to play nice together. Once the system was marginally operational, the limits were tightened to specified, and beyond, values to test the robustness of the system without having to continually make code changes. I could defi-
Definitely see the utility of doing the same for Hy-C integration in which parameters assigned to different architecture fields might need to be modified to more accurately portray the true system operation. Alternatively, the architectural description data could be stored within a table in the executable Hy-C implementation code and recompiled for each change. For the purpose of this dissertation, the data for the example DSP is portrayed in this chapter in four separate tables to provide clarity for discussion. There is no reason that, as a matter of simplified implementation, all four of them could not be merged into a single file for loading, as described above. As a demonstration of how such a file might be configured, a single-file version is provided in the Appendix. The choice of the implementation method is at the discretion of the user.

One premise of the Hy-C architecture description is that it starts with values pulled directly from the specifications, but it can then be modified as more is actually learned about the execution of the individual instructions or components. As such, the data is admittedly over simplified awaiting validation and informed modification in accordance with the analysis of empirically-collected data. Each set of data is specifically targeted to the specific computing resource or component, so the values loaded in the tables below are just the starting points in a realtime discovery process. The rationale for the initial values for the chosen DSP, memory access, and FPGA are discussed in the following sections.

5.2. Texas Instruments TMS320C64x DSP

The Texas Instruments TMS320C64x/C64+ DSP executes up to eight 32-bit instructions per cycle. It consists of 64 general purpose 32-bit registers and eight functional units, including two multipliers and six arithmetic logic units. The eight functional units can be divided into two groups of four and each of the four types of functional units (L, S, M, and D) supports different functionality and, therefore, different instructions. A summary of the types of fixed point instructions that are supported by each of the different Functional Unit
types is provided in Table 5.1[60]. The DSP Reference Guide actually contains a total of 59 processor instructions, each with its own several-page description of allowable operations and limitations. Fortunately, however, Hy-C does not require these low-level details because the DSP compiler will perform its own compilation of source code assigned to the C64x by the Partitioning Function. Hy-C only needs enough data to estimate the performance of the DSP with regard to its timing and power usage. For a single issue CPU, this would be as simple as a number for the instruction clock counts and another number for the estimated power usage. In this case, however, even though the DSP does not support actual pipelining, it does essentially utilize a very long work instruction, given that it can support the execution of 8 operations per clock cycle. To estimate both the power usage and the performance of this type of processor, the specific functional units that support individual DSP instructions must somehow be made visible to Hy-C to accumulate time/power estimates used by the Partitioning Function for code assignments to the most efficient processing elements. The Reference Guide simplifies this task by providing a table listing each instruction and its applicable Functional Units. To make it even simpler, the 59 DSP instructions have been analyzed and manually mapped to just 21 applicable “C Language Operators” or pairs of operators.
5.3. Instruction Set Description

Whereas other retargetable compilers have been shown to require such detailed architecture and instruction-level descriptions that their authors must fully understand both the nuances of the retargetable compiler and the intricate details of the retargeted processor in order to write instruction transformations to the new processors, Hy-C requires only those parameters necessary to estimate processing time and power consumption for the commercially-available computing resources used in the heterogeneous processor. The initial data for these data must be gleaned from the reference manuals which describes the available operators for the computing resources. A reference manual generally describes both the parameters for the operators and the number of clock cycles required to complete that operation. The Hy-C Partitioning Function then uses this data to estimate the performance of each available computing resource to determine the optimal assignment of functional code blocks to those resources. A separate, but equally simple, set of data is required to address the number of clock cycles required to move data to and from the various types of memory available in the system.

The clock cycle counts provided in the architecture description file represent the minimum number of cycles required for the execution of the associated operation, but the raw sum of the counts for an executable block of code does not necessarily represent the sequential amount of time required to execute that block. If, for example, the execution of a functional block is interrupted by the processor or has to wait for non-cached data, the realtime execution of the functional block could be lengthened, but Hy-C cannot take this added delay into account. The sum of the counts merely provides the minimal execution time for the functional code block. The realtime execution time of the overall program can be decreased if the computing resources include the ability to execute instructions in parallel. The Partitioner must keep track of the realtime clock cycle count that converts directly to actual execution time by multiplying the sum by the clock cycle interval, the reciprocal of
the clock frequency. The estimate for the execution of the functional block of code is based solely on the sum of the specified clock cycles required for each of the constituent operations in the block, including known associated memory operations. Extended execution time of the block of code due to runtime resource contention cannot be reliably predicted by Hy-C. This type of fidelity would require a simulator which can provide cycle-accurate modeling, like LISA Tek.

In a similar manner, energy units are provided in the architecture description to account for energy consumption. Just as in the case of runtime estimation, Hy-C accounts for energy consumption by maintaining a running sum of the energy units. Theoretically, the total energy consumption could be calculated by using the power rating for the various devices on the chip, but this conversion is not actually necessary to compare the energy performance of the individual resources. The running sum of energy units is sufficient for that comparison. This method also assumes that the heat dissipation requirements for heat generated by the execution of the code will be proportional to the sum of the energy consumption units.

Admittedly, this method of assessing runtime performance and energy consumption is not exact, but it is sufficient to discriminate between the relative performance of an individual block of code being executed on two or more candidate processing resources. Hy-C does not attempt to estimate the incipient static power consumption because it is assumed to be relatively stable across all execution scenarios. The performance noise caused by unforeseen resource contentions will most probably occur in any execution scenario and, thus, can be expected to be relatively similar between any execution of the code, even on a different mix of processors on the same heterogeneous device. Therefore, these extraneous delays inherent to code execution can be ignored. Closer estimates of execution time and energy usage can be simulated, as it is in the cycle-accurate processing simulation in LISA Tek[46], but that added marginal fidelity comes at a high cost in terms of complexity, time, and money. Hy-C is a simple solution.
<table>
<thead>
<tr>
<th>NAME</th>
<th>CLOCK SPEED</th>
<th>POWER</th>
<th>FORMAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>StrongARM</td>
<td>600 MHz</td>
<td>6</td>
<td>2 core</td>
</tr>
<tr>
<td>TMS320C64x</td>
<td>400 MHz</td>
<td>4</td>
<td>2D, 2M, 2L, 2S</td>
</tr>
<tr>
<td>WimpyARM</td>
<td>100 MHz</td>
<td>1</td>
<td>1 core</td>
</tr>
</tbody>
</table>

Table 5.2. Hy-C Description of Computing Resources

Once again, the time and power factors supplied in the example architecture description are based solely on data in or inferences derived from the vendor-supplied reference guide for the selected processing resource. It is fully expected that these factors will be fine-tuned in accordance with experimental, empirical data collected in the execution of Hy-C. Separate testing of each operation in an isolated test case with an instrumented processor could adjust the “clocks” and “power” values to get better fidelity in the estimation of the performance of the system or components under test.

Table 5.2 provides the general operating characteristics required by Hy-C for some example computing resources, including the Texas Instruments TMS320C64x DSP[60]. All of the data is self-explanatory except the Power figures in the third column. Power consumption by a processor is defined by the equation:

\[ P = CV^2f \]

In this equation, \( P \) is the dynamic CPU power consumption, \( C \) is capacitance, \( V \) is voltage, and \( f \) is frequency. This equation is germane to this discussion only in that it shows that power consumption \( P \) is proportional to frequency \( f \), since voltage can be assumed constant on a chip. While this calculation may not be precise, it is considered sufficient for the initial architecture description. Since Hy-C only needs to compute relative power usage between the various processing resources, it is acceptable to use the lowest frequency as the base number and make the other numbers multiples of that base number as the initial estimate. Once the system is operational, empirical measurements will provide
Table 5.3. Example Hy-C Operator Description for DSP

<table>
<thead>
<tr>
<th>INST</th>
<th>CLOCKS</th>
<th>STAGES</th>
<th>POWER</th>
<th>ENCODING</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>L or S or D</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>L or S or D</td>
</tr>
<tr>
<td>*</td>
<td>4</td>
<td>0</td>
<td>16</td>
<td>M</td>
</tr>
<tr>
<td>/</td>
<td>10</td>
<td>0</td>
<td>40</td>
<td>S</td>
</tr>
<tr>
<td>F*</td>
<td>10</td>
<td>0</td>
<td>40</td>
<td>S</td>
</tr>
<tr>
<td>F/</td>
<td>10</td>
<td>0</td>
<td>40</td>
<td>S</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>S</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>S</td>
</tr>
<tr>
<td>&lt;</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>L</td>
</tr>
<tr>
<td>&gt;</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>L</td>
</tr>
<tr>
<td>&gt;=</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>L</td>
</tr>
<tr>
<td>&lt;=</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>L</td>
</tr>
<tr>
<td>==</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>L</td>
</tr>
<tr>
<td>!=</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>L</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>S or L</td>
</tr>
<tr>
<td>!</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>S or L</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>&amp;</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>L</td>
</tr>
<tr>
<td>^</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>L</td>
</tr>
<tr>
<td>Call</td>
<td>10</td>
<td>0</td>
<td>40</td>
<td>S</td>
</tr>
<tr>
<td>br</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>S</td>
</tr>
</tbody>
</table>

fine-grained estimates for future use. Thus, the math is very easy for this example. The lowest frequency is that for the WimpyARM at 100 MHz. Thus, the base number of 1 is assigned to that processor for Power usage. The Power usage index for the TMS320C64x is $400/100 = 4$. And the Power usage index for the StrongARM is $600/100 = 6$. Remember that these figures are only intended to produce relative power numbers, not actual power figures.

The Hy-C architecture description for the TMS320C64x instruction set, actually for the modified operator set, as described earlier, is show in Table 5.3. The “shaded” blocks are informational only and are not included in the actual architecture description file. Each computing resource needs 5 columns for its individual parameter description. Descriptions of
the entries in each of the columns follows:

(1) Column 1 (INST ... INSTRUCTIONS) provides the “C-type” operator or operator combination being described.

(2) Column 2 (CLOCKS) provides an estimate of the minimum number of clocks that will be required to complete the operation. Most of the clock counts have been pulled directly from the TMS320C64x DSP Reference Manual. The exceptions to that rule are the clock estimates for “/” (divide), “F*” (floating point multiply), and “F/” (floating point divide). The Reference Manual does not provide any division operations nor any floating point operations, both of which are expected to be necessary in a DSP. Therefore, an assumption is made that division and floating point operations are going to have to be handled in software via a function call. Since the timing for the “Call” operation is provided in the Reference Manual at 10 clocks, it is assumed that 10 clocks should be sufficient for calls to a software function to handle these missing operations.

(3) Column 3 (STAGES) would provide the depth of the available pipeline capability, if it exists in the associated processor. Since the TMS320C64x does not support pipelining, all of the entries are 0 (zero).

(4) Column 4 (POWER) provides the estimate of the power consumed by the instruction, not including any latent, across the board, power consumption consumed by the DSP in an idle state. Since this is a description of the operations for the TMS320C64x, simply multiply the clock count in Column 2 by the POWER factor provided for this processor in Table 5.2. In this case, simply multiply the number in Column 2 by the number 4 (from Table 5.2).

(5) The entries in Column 5 come directly from the table in Appendix B (Mapping Between Instruction and Functional Unit) of the TMS320C64x Reference Manual[60].
5.4. Memory Description

Accounting for the impact of memory access is a critical component in any system, but especially in a computer system designed to maximize parallel processing[56]. Since this is the purpose of the Hy-C, it is critical that memory processing be considered in the estimation of the performance estimates for instruction processing. As such, the Hy-C architecture description includes table entries to address both processing time and power utilization associated with memory access, both reads and writes, for each type of available memory. Shared memory will be utilized for passing data between the processors, but each processor is responsible for its own caching of data for optimal performance. Each processor has dedicated L1 and has access to off-processor Random Access Memory.

Table 5.4 provides the timing and power data associated with stores and loads to, from, and between registers and memory. Once again, the timing data is taken from the TMS320C64x0 Reference Manual. The Power index is calculated the same as described for the instruction set. Since this is the TMS320C64x processor with a Power Index of 4, the power consumption for each of the rows is computed by multiplying the number of clocks times the applicable Power Index of 4, in this case.
5.5. FPGA Description

The inclusion of FPGA resources for on-the-fly realtime use is not a new concept as its inclusion in realtime systems was mentioned in a paper by De Michelli, et al, as early as 1996[33]. There is scant evidence in engineering literature, however, that proclaims the routine inclusion of FPGAs in this on-the-fly role in any systems. Knowing that this would be a difficult problem, planning for the Hy-C program assumes that the initial instantiation of FPGA resources will contain predetermined functions that are known to be economically executable in FPGAs, such as, matrix multiplication and fast Fourier transfers. In fact, the problem space would have to be known so well as to predict exactly what size of matrices might be expected in order to provide performance estimates for that particular matrix size. Thus, the number of predetermined functions might be quite high, depending on the expected variability in the input data. Lacking any suggestion to the contrary, the only manner in which estimates of the speed and power usage of these devices can be acquired is through execution of a similar model in some sort of simulator. MATLAB is a tool often used in engineering industries to predict the performance of systems, so it might be a good fit for this application also. Modeling of this sort would produce a time figure in absolute time, probably microseconds or milliseconds. In order for this data to be useful for performance comparison, Hy-C would either have to convert this absolute time to equivalent clocks or Hy-C would have to convert the clock totals derived from the simulated execution of the other processing resources to an equivalent absolute time.

I was able to find an EDN Network article which indicates that FPGA manufacturers provide power/performance spreadsheets with the purchase of their products[65]. It is directed toward hardware designers who must accurately measure the power and heat dissipation requirements during the design of new boards, but “none of the requirements is trivial.” The same paper describes an elaborate estimation process that the designers use during the course of the design effort to refine their estimates in the final product. This level
of estimation is far more than is needed for the purpose of Hy-C and it is only an off-line estimate which would not work for on-the-fly estimation. Lacking an actual FPGA or test stand, I can only suggest that the initial Hy-C implementation restrict its FPGA usage to estimates derived from off-line simulations.

An example architecture description for the FPGA functions is shown in Table 5.5. Note that the TIME provided for this table is in microseconds, not clocks. Furthermore, the Power Index provided is used slightly differently than for the other components. All of the other tables provide the actual power value to be used to determine the relative efficiency of the component. In the case of the FPGA, however, the “power” value must first be multiplied times the “time” value only after the absolute time has been converted to an equivalent number of clocks. It should also be noted that the power index is significantly lower, but at least an order of magnitude, than that of the other components. The FPGA is by far the most energy efficient component shown in this chapter.

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>TIME</th>
<th>POWER</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatMul10x10</td>
<td>60</td>
<td>.1</td>
</tr>
<tr>
<td>FFT10x10</td>
<td>15</td>
<td>.1</td>
</tr>
<tr>
<td>Malloc</td>
<td>5</td>
<td>.1</td>
</tr>
<tr>
<td>Free</td>
<td>3</td>
<td>.1</td>
</tr>
</tbody>
</table>

Table 5.5. FPGA Architecture Description
CHAPTER 6

CONCLUSION AND FUTURE WORK

Our modern lives are unavoidably intertwined with technological advances, and that trend is not about to reverse itself any time soon. Every day brings new opportunities in the electronics industry which drive changes in the very way we live our lives. Hand-held devices do not represent the end goal of this electronic technological revolution. On the contrary, they are but the start of a process which will bring electronic devices into every niche of our lives. What teenager today does not carry a cell phone at all times? We already have the technology to place a radio frequency identification (RFID) chip under the skin of dogs and cats to allow properly equipped veterinarians to positively identify our pets if they are lost. How long will it be before we decide to do the same to Grandma who has Alzheimer’s Disease and frequently walks away from her room at the nursing home? Tiny electronic devices are already giving hearing back to the deaf. Can artificial sight be far behind? These devices are miraculous, though the ethical considerations of such applications have yet to be determined in everyday life, but that is a topic for someone else’s dissertation.

So, what do all of these innovations have in common? All of them are going to be implemented with some degree of software, and the innovations are going to come fast and furious, creating the need for more and more powerful software in less and less development time. It is exactly this type of environment which will benefit the most from innovative heterogeneous processors and retargetable compilers, and those compilers will need fast and correct architecture descriptions to feed the consumer needs in a timely fashion.

The computer industry has already spawned scores of multicore processors and associated compilers. It took years of research and trial and error to design compilers that make efficient use of the multiple cores, and there is still a long way to go. The current hot market is for personal hand-held devices, such as, cell phones and tablets. Improvements in these devices are quick to catch the eye of discerning consumers. I remember the first mobile phones
in the 1970s. They were heavy and large with poor reception and almost no battery life on a battery the size of a cigarette package. We affectionately called the brick, for obvious reasons. They were also very expensive, but they were quickly abandoned with the delivery of the first cellular phone. The old flip phones only lasted a few years until the iPhone made its debut. No longer did the consumer have just a wireless telephone! The iPhones came with a camera, games, contact lists and so much more in a light, compact case! Every couple of years the new enhanced versions have come on the market with more speed, more memory, smaller size, and more applications (apps). That 2-year enhancement period is significant in that it coincides nicely with the typical minimum 2-year service contract offered by most cell phone providers. To not have a new high-end “toy” ready at the end of the service contract is a sure way to lose that customer to another provider. The research for this dissertation implies that this will not change in the foreseeable future, and that software is going to have to be more responsive than ever to get a product on the market quickly and error-free.

According to a July 2, 2014, report issued by Markets and Markets, a Texas Market Research Company and Consulting Firm in Dallas, the need for increased performance and improved efficiency in the mobile phone market is the major trigger driving the growth of the heterogeneous processor market[10]. It identifies possible areas of application as consumer, telecommunication, automotive, model driven architecture, and medical devices. The same source also indicates that “the total Heterogeneous Mobile Processing and Computing Market is expected to reach $61.70 billion by 2020, growing at a CAGR of 20.75% from 2014 to 2020.” Compound Annual Growth Rate (CAGR) is an investment term that provides a statistical measurement “of the year over year growth rate of an investment over a specified period of time[2].” In other words, it is the level amount of growth that an investment would have had to make each year during the reporting period to produce the terminal value. It can sort of be thought of as an average growth rate, though it is not equivalent to an average of the actual overall growth figure. An estimated growth rate of 20.75% certainly seems
like an economic indicator supporting the continued development of heterogeneous processor capabilities. The reason for that growth rate is not difficult to discern. Today I watched news coverage of a Malaysian Airliner reportedly shot down over Ukraine. The wreckage of the airplane was strewn about along a 9 mile long debris trail and in every video report from every site along that path I saw people of all walks of life taking their own videos of the scene with wireless phones. Personal electronic devices represent the ultimate international must-have item. The market for personal electron devices is ever-changing, but it is here to stay.

6.1. Software Engineering

Accomplishing the task of software keeping pace with hardware improvement in personal electronics will not be easy, especially in light of the jaded history of software engineering portrayed in Chapter 1. Chapter 2 reviewed the processes, tools, and procedures employed by members of the software engineering community over the course of the past 3 or 4 decades in an attempt to find a possible common thread that might have contributed to past failures in software products. The software development specifications, up to and including DOD-STD-2167, failed to recognize the evolving nature of massive software systems. These early specifications portrayed a single waterfall process that began at one end with a system specification and ended when the system was built and tested. It was taken for granted that all requirements were complete and correct on day 1, never in need of revision thereafter. Unfortunately, the complexity of most of the DOD programs was so vast that nobody could know all the details at the beginning. These programs began as dreams and expected the dreams to come true from good effort on the part of the contractor, but this was rarely the case. Most development programs were actually executed like a cook peeling an onion. Only after prototyping or designing one piece could you fully understand how it fit in the overall scheme of the program. Then you would attack the next layer. At each step, as lessons were learned, the contractor and government alike were forced to reassess the overall program progress and the work left to be done. Needless to say, this resulted in unacceptably long
delays in fielding the desired systems and, since time is money, at a cost significantly above that of the original contract price.

DOD-STD-2167A replaced DOD-STD-2167 mainly to acknowledge the existence of the cyclical nature of large program development, both hardware and software. Specifically, it encouraged prototyping as a means for determining the real program requirements and then it allowed hardware and software to develop in an independent fashion until the time for their formal integration. Unfortunately, not everyone on the government side got that clarification memo and development under Rev A was not much different than under the strict rules of the original specification. In both cases, it was obvious that the personnel in charge of oversight of the contractors were much more concerned about the quality of the ponderous list of required documents than they were about the software itself. As such, elaborate CASE tools were built or purchased for the sole purpose of automating the construction of the required documents. The actual software was still being written by hand in some dark corner of a lab somewhere with scant little visibility as to the status or quality of the actual software product.

MIL-STD-498 was the DODs last stand on governing the software process. This standard was quickly replaced by the use of commercial standards and, in reality, by the use of the CMM/CMMI processes in the development of software products. In this realm, continuous improvement of the software process, or the software performance, depending on the source, was the overall goal of the process. Certification at CMMI Level 3 was mandated by the DOD for any company to compete for a DOD contract, so an industry was born within the software engineering community. The certification assessment process was contracted to a single source, the Software Engineering Institute at Carnegie-Mellon University, and the process was expensive and time-consuming for the contractors. Moreover, certification had to be accomplished every 3 years. As such, all software tools written or purchased during this era were for the sole purpose of collecting and analyzing metrics on everything that could
possibly be measured in the product/software development process in order to produce data to improve the process. Thankfully, this era is also waning as the DOD is currently relying on commercial or contractor in-house standards to accomplish the software engineering task.

So, how did so many software errors slip through the cracks to be released in fielded products? First of all, the answer lies not in the methods or processes themselves. Any of the above-mentioned standards was more than capable of producing quality, error-free, software. The problem with the processes was the implementation of the processes by contractor and government personnel alike. All of the standards and processes allowed, and even encouraged, tailoring. Tailoring was a process by which a program started with the fully-defined process for all parts of the program and then systematically removed any unnecessary process for individual products for which a greater level of process made no sense. For example, throw-away code used in integration to extract internal data for comparison certainly should require no process except archival within a configuration management systems. Yet, every program on which I worked required the fullness of the complete process on even this kind of code.

Secondly, the tools generated or purchased to accompany the software processes were directed at collecting data to support the process, not at improving or ensuring the quality of the software itself. Early tools tended to be automated documentation generators, that never quite worked as advertised. CMMI tools were tool to gather random statistics about the process with, again, little regard for the quality of the actual software product. How did errors in the software make it through the software processes? The simple answer is that the processes, and especially the automated tools associated with the processes, never emphasized the need to look for the errors.

A process is much like a map. The lines, colors, and symbols on a map have meaning. In the hands of someone skilled in the knowledge of the meanings of those lines, colors, and symbols, a map can lead you straight to your desired destination. In the hands of someone without the requisite skills, a map is just a piece of paper you will be carrying when you stop
at the next gas station to ask for directions. Software development is the codification of human thought and needs to be done by skilled software engineers to create good software. My assertion in this dissertation is that the complexity of the processes and the unfortunate misdirection of attention, by the chosen processes, from the actual tasks involved in physically creating the software itself allowed errors in the code. Reducing complexity of the design, the process, and/or the automated tools could have been instrumental in preventing or finding errors before they were released in the final product.

6.2. Retargetable Compilers and the Hy-C Solution

The idea of retargetable compilation has been bandied about the software industry for at least 40 years. The DOD was among the first organizations to recognize the value of retargetable compilers to solve a problem that was fairly unique to the military. Weapon systems, especially strategic deterrent weapon systems, were developed with state-of-the-art technology at the time they were manufactured, but then would remain in storage or on active alert for decades waiting for circumstances that might warrant their use, but hoping those circumstances would never materialize. Over time, components in these systems wore out or malfunctioned and had to be replaced. While the DOD programs do buy spare parts as addenda to their initial contracts, nobody can know how many of which parts will be needed as replacements 2 decades from now. This, all too often, placed the DOD in the unenviable position of having to choose between a rock and a hard place: either manufacture new parts to the old specifications, if the machinery to do so was even available, or redesign the system to use currently available parts. The existence of retargetable compilers, in the minds of the military leadership, would allow the old software to be recompiled for the new, and available, devices. As good as this sounds, it is fraught with problems and retargetable compilation for this problem has never been developed.

On the other hand, the current mega-market for hand-held digital devices has spurred a renewed interest in retargetable compilation to maintain a 2-year cycle for major break-
throughs in the hand-held electronic product lines. Historical program data implies that creating new hardware and new software with full product testing within a 2-year period using current development methodologies is infeasible. The concurrent design of new hardware and new software always has some elements of risk as designs change to match reality or available schedules. It is in this arena where retargetable compilers are seeing a resurgence as tools in the hardware/software codesign process. The available retargetable compilers are very complex and require skilled engineers who are well versed in both the intricacies of the syntax of the retargetable compiler and in the architectural and syntactic nuances of the target processor. It is an expensive process, but if it significantly shortens the development time, and provides some modicum of flexibility along the way, it just might be worth the money to maintain overall profitability.

The Hy-C Retargetable Compiler is offered as a possible candidate for decreasing the complexity of concurrent software development in a very small niche in the field of consumer hand-held electronics. Heterogeneous processors consisting of a number of diverse processor types on a single chip are entering the market for use in consumer hand-held electronics. While some of the processors for these heterogeneous chips may be of new designs, it is not unreasonable to assume that some chips will be built with a lower risk of failure by using commercially available processors already in wide use in the industry. Hy-C exploits the use of the native compilers for these off-the-shelf processors, negating the need for detailed remapping of microinstructions. Instead, it simply requires an estimation of the processing time required for each instruction along with an estimation of the energy consumed by the execution of the instruction. Existing retargetable compilers, in all of their complexity, cannot perform the simplified processing of Hy-C to efficiently distribute processing among the computing resources. Furthermore, Hy-C will provide an additional benefit of the inclusion of FPGA and, eventually, ASIC resources to make execution even more efficient.
6.3. Contribution

The major contribution of this dissertation is the introduction of the concept of the Hy-C Retargetable Compiler for heterogeneous processors on a chip. It is envisioned for use with heterogeneous processors when the computing resources on the device happen to be existing processors with their own native compilers. The required architecture description for these devices is limited to essentially an estimate of the processing time and power-consumption for each operation resident in each processor. Armed with little more than the Programmer’s Handbook for the CPUs, a relative novice could provide the necessary data in a very short time allowing a very fast evaluation of the configuration under test for a given input program.

The other contribution with regard to Hy-C is the research into the seamless incorporation of FPGAs and/or ASICs into heterogeneous processors. While the initial integration of a FPGA will be with one or more preprogrammed functions, the ultimate goal is to create the necessary gate routing on-the-fly to maximize the overall utility of the firmware devices. The techniques offered in this dissertation are theoretical in that the other components of Hy-C are not yet available. I am sure that the techniques will be refined as a result of empirical data derived from the execution of the completed Hy-C System, but I am confident that the numbers provided will provide sufficient reward to validate the system utility.

The other contribution of this dissertation is a very personal one. Having spent thirty years of my professional life as a practitioner of software engineering for various government contractors, I have witnessed far too many failures and watched far too much money being wasted on misguided software programs. Most practitioners blamed the “process,” while management and the government oversight personnel worshiped the process of the week as the cure for all that was wrong with software. For most of my professional life, I was a staunch member of the former group, fighting the vestiges of process wherever I could. As a result of my research for this dissertation, however, I have come to see “process” in a different light. As I spent time reviewing the nuances of each of the processes that I
described in Chapter 2, I have come to the realization that, while no process is perfect for every software project, most of the recent processes could have been effective for most any project if, and only if, the company engineers and their government oversight partners had, in fact, performed their assigned duties of sufficiently tailoring the processes for each product on the program. Furthermore, both sides need to be significantly more flexible in the application of the process on a daily basis. The programs need to be executed by engineers who can think, reason, and make good and timely decisions. The lawyers and bureaucrats need to stay out of the development labs.

In my first year of service to this great Nation as a freshman at the United States Air Force Academy in 1973, I had to memorize the following quotation: “Never tell people how to do things. Tell them what to do and they will surprise you with their ingenuity.” These great words were spoken by General George S. Patton. He was an Army commander, not an engineer. Every action of the men under his command from the clothes they would wear, to the food they would eat, to the arrangement of their drawers, and the fighting of their wars, was covered in military regulations 5 inches thick. And, yet, General Patton admonished the officers under his command to “let them surprise you with their ingenuity.” My guess is that those remarks were directed at men in the struggles of war. Men who had to “think out of the box” long before that became a cliché of American business. In many ways, development of military hardware and software mimic the actions of war. To win the war, diverse groups of warriors must accomplish their mission in any manner that they can to support the final goal of winning the battle. So, too, should engineering organizations take pride in getting their part done using whatever means is available to support the overall goal of getting the product into the hands of the warrior. We need to take a look at history before we commit ourselves to the same old rut that has been followed for generation after generation. Following a process is good and necessary, but wasting time, schedule, and effort on useless tasking demanded by any process should be questioned.
6.4. Future Work

The most obvious future work for the described architecture description and Hy-C, in general, is the implementation of the full Hy-C functionality. This dissertation describes the steps necessary to take the first step and prepare the architecture description, once the actual components have been identified. To date, the creation of the architecture description is a manual process. Adding automation to this process, possibly in the form of a web-driven interface, could both ease the tedium of the creation of the architecture description and eliminate a source for error. Regardless of how the initial architecture description is created, the values will have to be validated or modified in accordance with collected empirical data to increase the fidelity of the description for any given processor configuration. Experience with actual performance data will then lead to better methods for initial estimates.

Besides the architecture description, there is still so much to be done to bring the Hy-C project to fruition. A significant amount of research has gone into the programming of the FPGAs, but more is needed to produce working prototypes. Even more work will be necessary to provide the capability to create on-the-fly gate-level interconnections to implement FPGA circuits to configure the device for a specific software function with better performance than executing the corresponding code on a microprocessor.[33]. This is a concept with significant commercial viability, but with a long road ahead of it.
APPENDIX

INSTRUCTION SET ARCHITECTURE DESCRIPTION FOR HyC
Chapter 5 described the details of the Hy-C architecture description with a detailed description and explanation of the elements required for the inclusion of the Texas Instruments TMST MS320C64x DSP as a constituent computing resource in a hypothetical heterogeneous processor. Exactly how the architecture data is entered into Hy-C is at the discretion of the developer. One method might be to manually embed the data into tables or other data structures within the code itself. Changes to the parameters as a result of the analysis of empirically collected data could then be manually reprogrammed into Hy-C and the code recompiled. Another possibility, one that I have personally used on large-scale integration projects in the course of my career, is to enter the data into an external data file to be read by the Hy-C program as part of the initialization sequence. This method allows the parameters to be quickly changed without any recompilation. The data can be stored and manipulated via a user-friendly Microsoft Excel spreadsheet and saved to a *.csv (comma-separated variable) file to be readable to the Hy-C program. Therefore, the complete description for the DSP is presented here in the form of a continuous, uniformly dimensioned table that could be implemented in the form of a spreadsheet.

The table shown below is a simple concatenation of the four tables provided in Chapter 5. Since those tables have different numbers of columns, the concatenated table below has quite a few blank cells, indicated by a single dash (-) in the center of the cells. These cells can be ignored by any input program. Similarly, any cells highlighted in gray are included here for informational purposes only and should probably be deleted before converting to a *.csv file in preparation for input into Hy-C. They would typically not be included in the actual input file, though they could be included if the Hy-C input function were designed to ignore them. Once again, this is an implementation issue left to the discretion of the developer.
Table A.1: Example Hy-C Architecture Description

<table>
<thead>
<tr>
<th>NAME</th>
<th>CLOCK SPEED</th>
<th>POWER</th>
<th>FORMAT</th>
<th>INST</th>
<th>CLOCKS</th>
<th>STAGES</th>
<th>POWER</th>
<th>ENCODING</th>
</tr>
</thead>
<tbody>
<tr>
<td>StrongARM</td>
<td>600 MHz</td>
<td>6</td>
<td>2 core</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TMS320C64x</td>
<td>400 MHz</td>
<td>4</td>
<td>2D, 2M, 2L, 2S</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WimpyARM</td>
<td>100 MHz</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Continued on next page
Table A.1 – continued from previous page

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>CLOCKS</th>
<th>STAGES</th>
<th>POWER</th>
<th>ENCODING</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>L</td>
<td>-</td>
</tr>
<tr>
<td>Call</td>
<td>10</td>
<td>0</td>
<td>40</td>
<td>S</td>
<td>-</td>
</tr>
<tr>
<td>br</td>
<td>6</td>
<td>0</td>
<td>4</td>
<td>S</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SOURCE</th>
<th>DESTINATION</th>
<th>CLOCKS</th>
<th>STAGES</th>
<th>ENCODINGS</th>
<th>POWER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Areg</td>
<td>5</td>
<td>0</td>
<td>D</td>
<td>20</td>
</tr>
<tr>
<td>Memory</td>
<td>Breg</td>
<td>5</td>
<td>0</td>
<td>D</td>
<td>20</td>
</tr>
<tr>
<td>Areg</td>
<td>Areg</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>4</td>
</tr>
<tr>
<td>Breg</td>
<td>Breg</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>4</td>
</tr>
<tr>
<td>Breg</td>
<td>Areg</td>
<td>2</td>
<td>0</td>
<td>D</td>
<td>8</td>
</tr>
<tr>
<td>Areg</td>
<td>Breg</td>
<td>2</td>
<td>0</td>
<td>D</td>
<td>8</td>
</tr>
<tr>
<td>const</td>
<td>Breg</td>
<td>1</td>
<td>0</td>
<td>S</td>
<td>4</td>
</tr>
<tr>
<td>const</td>
<td>Areg</td>
<td>1</td>
<td>0</td>
<td>S</td>
<td>4</td>
</tr>
<tr>
<td>Areg</td>
<td>Memory</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>4</td>
</tr>
<tr>
<td>Breg</td>
<td>Memory</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>TIME</th>
<th>POWER</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatMu10x10l</td>
<td>.1</td>
<td>-</td>
</tr>
<tr>
<td>FFT10x10</td>
<td>.1</td>
<td>-</td>
</tr>
<tr>
<td>Malloc</td>
<td>.1</td>
<td>-</td>
</tr>
<tr>
<td>Free</td>
<td>.1</td>
<td>-</td>
</tr>
</tbody>
</table>
REFERENCE LIST


[23] Don Batory, Why (meta-)theories of automated software design are essential: A personal


Dept. of Computer Science, Univ. of Ill. at Urbana-Champaign, Urbana, IL, 2002, see http://llvm.cs.uiuc.edu.


[60] Texas Instruments, Inc., *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide*, July 2010, Literature Number: SPRU732J.


[64] Arthur H. Watson and Thomas J. McCabe, *Structured testing: A testing methodology*
