Reconfigurable Systems: A Potential Solution to the von Neumann Bottleneck

Damian Miller

A Senior Thesis submitted in partial fulfillment
of the requirements for graduation
in the Honors Program
Liberty University
Spring 2011

Acceptance of Senior Honors Thesis

This Senior Honors Thesis is accepted in partial
fulfillment of the requirements for graduation from the
Honors Program of Liberty University.

____
Mark Shaneck, Ph.D.
Thesis Chair

____
Terry Metzgar, Ph.D.
Committee Member

____
Monty Kester, Ed.D.
Committee Member

____
James H. Nutter, D.A.
Honors Director

____
Date

Abstract

The difficulty of overcoming the disparity between processor speeds and data access speeds, a condition known as the von Neumann bottleneck, has been a source of consternation for computer hardware developers for many years. Although a number of temporary solutions have been proposed and implemented in modern machines, these solutions have only managed to treat the major symptoms, rather than solve the root problem. As the number of transistors on a chip roughly doubles every two years, the von Neumann bottleneck has continued to tighten in spite of these solutions, prompting some computer hardware professionals to advocate a paradigm shift away from the von Neumann architecture into something entirely new. Many have begun advocating the relatively new technology of reconfigurable systems, popularly known as morphware. The difficulty with adopting a new architectural paradigm, however, is that developers on both sides of the software-hardware spectrum must start from scratch, creating entirely new operating systems, hardware peripherals, application software, and user interfaces, all of which must seem familiar to the end user, yet still take advantage of the improvements morphware has to offer. With this in mind, this thesis builds off of the fundamental theory and current implementations of morphware to describe the processes and products necessary to develop and deliver morphware to the average user as a viable alternative to current technology.

Reconfigurable Systems: A Potential Solution to the von Neumann Bottleneck

The Von Neumann Bottleneck

Through the years, a variety of problems have plagued the development of faster, smaller, and cheaper computer hardware. Generally, the faster and smaller the component, the more it would cost. Fortunately, over the years computer hardware has improved roughly according to the predictions of Intel co-founder Gordon Moore, who predicted that the number of transistors that would fit on a single integrated circuit would double every two years [1]. However, in order to reap the full benefits of this rapid growth, developers have been faced with the task of overcoming one of the oldest, most pervasive problems in computer hardware— the von Neumman bottleneck.

The von Neumann bottleneck arises from the fact that CPU speed and memory size have grown at a much more rapid rate than the throughput between them; thus, although memory may hold a lot of data that needs to be processed, and the CPU may be using only a fraction of its computational power, the limited data access speed prevents the computer from doing its work any faster [2]. Although a number of options have been proposed to help alleviate this bottleneck, including cache memory and branch predictor algorithms, even the best of them are probabilistic. Thus, none of these solutions can provably in all situations solve the problem of the von Neumann bottleneck. For example, cache memory alleviates the von Neumann bottleneck only if the data needed by the processor is stored in the cache; if not, the processor must still wait for the data to be retrieved from main memory or external storage [3]. Similarly, branch predictor algorithms only help to lessen the von Neumann bottleneck if they predict the program flow correctly; if not, the next instruction(s) and all necessary data must be retrieved from slower memory while the processor waits [3]. Furthermore, although some of these options have met with great success as temporary solutions, they have done very little, if anything, to correct the growing discrepancy between processor speed and memory access speed shown in Figure 1 [3]. While these solutions may be adequate for many current applications, as the gap between processor and memory access speed continues to grow, the von Neumann machine may eventually become impractical for many applications that require a high degree of processing efficiency.

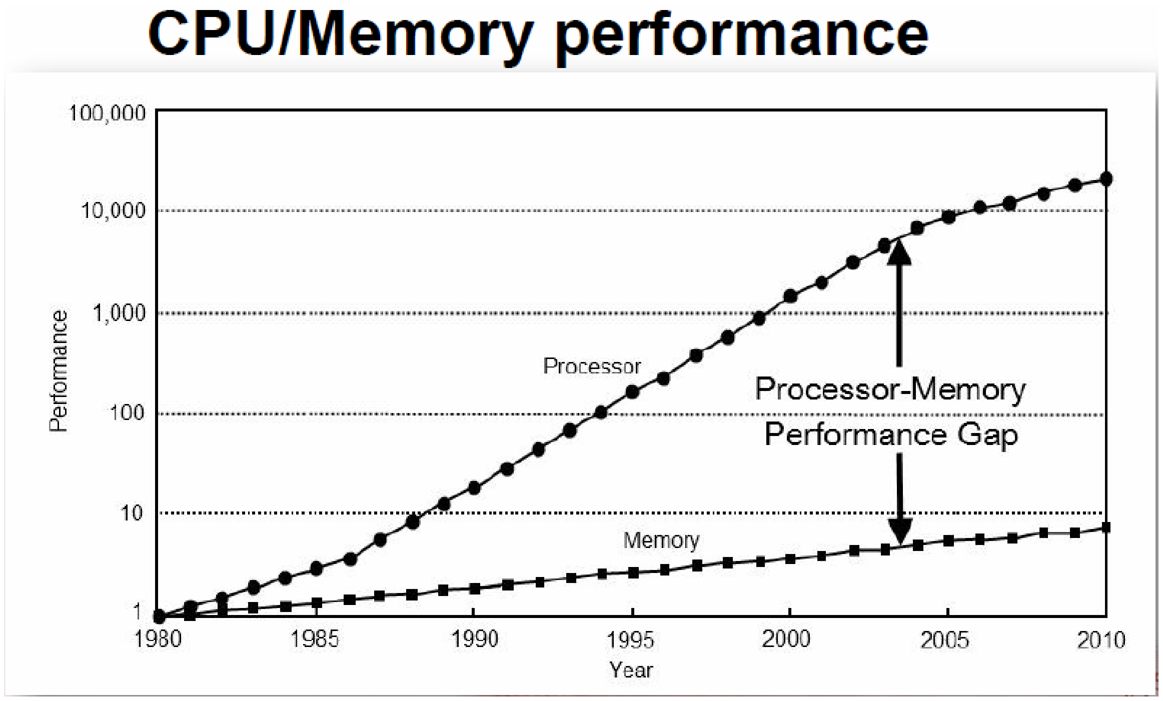

Figure 1: This graph demonstrates the approximate difference in growth between processor speeds and memory access speeds over the last three decades. Performance units on the left are in mhz [4].

Paradigm Shift to Morphware

Because the problem is inherent in the architecture of von Neumann machines, some professionals believe the solution is not to add new features to alleviate the symptoms, but rather to switch to a completely new architecture that does not have the same weaknesses as von Neumann's design [5]. One of the ideas submitted as a possible replacement for the von Neumann architecture is the reconfigurable system, otherwise known as morphware— the idea of using hardware circuits which may be physically reconfigured in order to transfer and/or modify data.

Theoretical Advantages

In addition to eliminating the von Neumann bottleneck, the concept of reconfigurable systems has several strong theoretical advantages over the traditional von Neumann paradigm. The primary advantage is flexibility— reconfigurability allows the system to be altered to perform a variety of different functions, instead of being hardwired to perform a single specific function [6]. The second advantage is fault tolerance. Reconfigurable systems are naturally more robust than their von Neumann counterparts, in that a faulty section of a circuit may be avoided by reconfiguring the rest of the circuit to bypass the bad sector [6]. This functionality allows the system to continue operating without a hitch, even if one of its components fails, allowing a level of reliability usually unmatched in a von Neumann system. The third and final advantage is efficiency in creating new systems. When new functionality which is not offerable by software is required for an application, a von Neumann system must either be upgraded or completely replaced with a new system that offers the desired functionality—a process which costs significant time and money. However, in many cases, reconfigurable systems are able to partially or completely change their organization in order to provide the new functionality, usually in a matter of minutes or even seconds [6]. These significant advantages will become clearer to the reader as reconfigurable systems are explained in more depth in Section 2.

Once a solid conceptual understanding of morphware has been established, the remainder of this paper will focus on the two current hurdles reconfigurable systems must overcome to be viable in today's market—a unified system of configware development and a familiar user interface (provided by an operating system).

Components of Reconfigurable Systems

This section of the thesis will explain the conceptual underpinnings of morphware and establish concrete definitions for the components used in reconfigurable systems.

Data Stream Processing

The first step in adequately understanding morphware is to establish some important terminology. Von Neumann machines may be understood as instruction stream processors running software (instructions and data); the processor runs through the instructions in the software, which access data from memory and move, modify, or delete it in order to accomplish some task [7]. Rather than an instruction stream, however, the reconfigurable systems paradigm is formulated around the idea of a ***data stream processor***—instead of fetching and processing instructions to operate on data, the data stream processor operates on data directly by means of multidimensional ***systolic arrays***, in which each cell of the array is a processor that operates on and stores data independently of the other cells [8]. In a data-stream based system, execution of a program is not determined by instructions, but rather by the transportation of data from one cell to another—as soon as a unit data arrives at a cell, it is executed [8]. Transportation (i.e., which data unit needs to be sent to which cell/array port at which time) is handled by ***flowware***. In a sense, flowware is the "data scheduler" of a data-stream based system [9].

It is this data-driven architecture that allows reconfigurable systems to eliminate the von Neumann bottleneck. Rather than all data being processed through one or several large hubs (the processors), each cell of each systolic array functions like a miniature processor. The data is then transformed as it is transported through the systolic arrays, thereby eliminating the need for costly downtime as data is transported to and from a single processor.

Configware and Reconfigurable Systems

Morphware **(reconfigurable systems)** is built on the idea of making a data-stream based system reconfigurable—that is, instead of hardwiring the connections between cells in a systolic array, morphware utilizes configuration code called ***configware*** (stored in a “hidden” RAM to prevent tampering or overwriting) to reconfigure the hardware circuits before runtime [7, 10]. Which parts of the circuits are reconfigured is determined by the system's ***granularity***—the smallest functional block of circuitry that is reconfigurable by the configware's mapping tools [11]. Most morphware systems fall into one of two classes of granularity—***fine-grained*** reconfigurable systems (smaller functional blocks) or ***coarse-grained*** reconfigurable systems (larger functional blocks) [11]. The simplest way to make a data-stream based system reconfigurable is to make the data-stream processor the reconfigurable unit. In such a system, configware is responsible for reconfiguring the connections between data-stream processors in the system, as well as configuring each processor to perform a specific task. However, the flowware is still responsible for the transportation of data between cells in the systolic array [7]. This idea of a reconfigurable data-stream processor (along with some applications for hardwired data-stream processors) forms the basis of the ***antimachine***

paradigm, one of the leading candidates for possible replacement of the von Neumann machine [7].

Reconfigurable Data Paths

Another method of implementing coarse-grained reconfigurable systems (which is a little more finely-grained than a reconfigurable data-stream processor) is implementing the systolic array as a ***reconfigurable data path array (rDPA)***, with each cell being a ***reconfigurable data path unit (rDPU)*** [8]. Each rDPU is multiple bits wide (i.e., 16-bits, 32-bits), and may be configured by the configware to perform a specific function. In such a system, the rDPU is the smallest reconfigurable functional unit—hence its classification as coarse-grained morphware [8]. The configware also handles pre-runtime reconfiguration of the interconnections between each rDPU cell in the array. Thus, since the configware can configure the connections between each rDPU such that there is only one path for the data to follow (to the next rDPU that needs to operate on it), there is no need for flowware in an rDPA-based reconfigurable system [10].

Due to their ability to fit entire words into a single rDPU, coarse-grained reconfigurable systems are ideal for large computations and algorithms that require wider data paths [11]. This optimization is largely a side effect of the shorter interconnections between each rDPU in coarse-grained systems, which is usually done at the expense of flexibility [11]. Fortunately, in most of the applications that use coarse-grained reconfigurable systems, the necessary algorithms and word sizes are known in advance, thus allowing each rDPU to be optimized to perform exactly the functions expected of it, while still leaving room for flexibility [11]. However, in many applications, the necessary algorithms are not known in advance, necessitating a great deal of flexibility not provided by coarse-grained morphware. For such applications, fine-grained morphware may be a better option.

Field-Programmable Gate Arrays

The primary implementation of fine-grained reconfigurable systems is the ***field-programmable gate array (FPGA***; also known as ***reconfigurable gate array, rGA)***, basically a programmable integrated circuit containing many ***configurable logic blocks (CLBs)***, each of which may be programmed directly by the system's configware [12]. FPGAs do not require flowware for much the same reasons as rDPAs—the configware itself manages the connections between CLBs so that the data can only follow one path. Depending on the function of the FPGA, each CLB may be configured to perform anything from a simple AND or XOR gate to a complex mathematical function [12]. Additionally, each CLB usually includes some form of memory, whether that be a simple flip-flop, or a larger block of memory [12]. At runtime, placement and routing configware is used to place a program’s logic circuits and memory into the proper CLB on the FPGA, and then route it to another CLB or a connecting bus [12]. Historically, FPGAs are much slower and less efficient than their von Neumann counterparts, application-specific integrated circuits (ASICs), although FPGAs have been quickly catching up in recent years [12]. However, their strength lies in their reprogrammability, which allows them the flexibility to adapt to new applications in situations where a normal ASIC would need to be completely replaced [12]. Figure 2 provides a visualization of an FPGA with CLB “island” architecture.

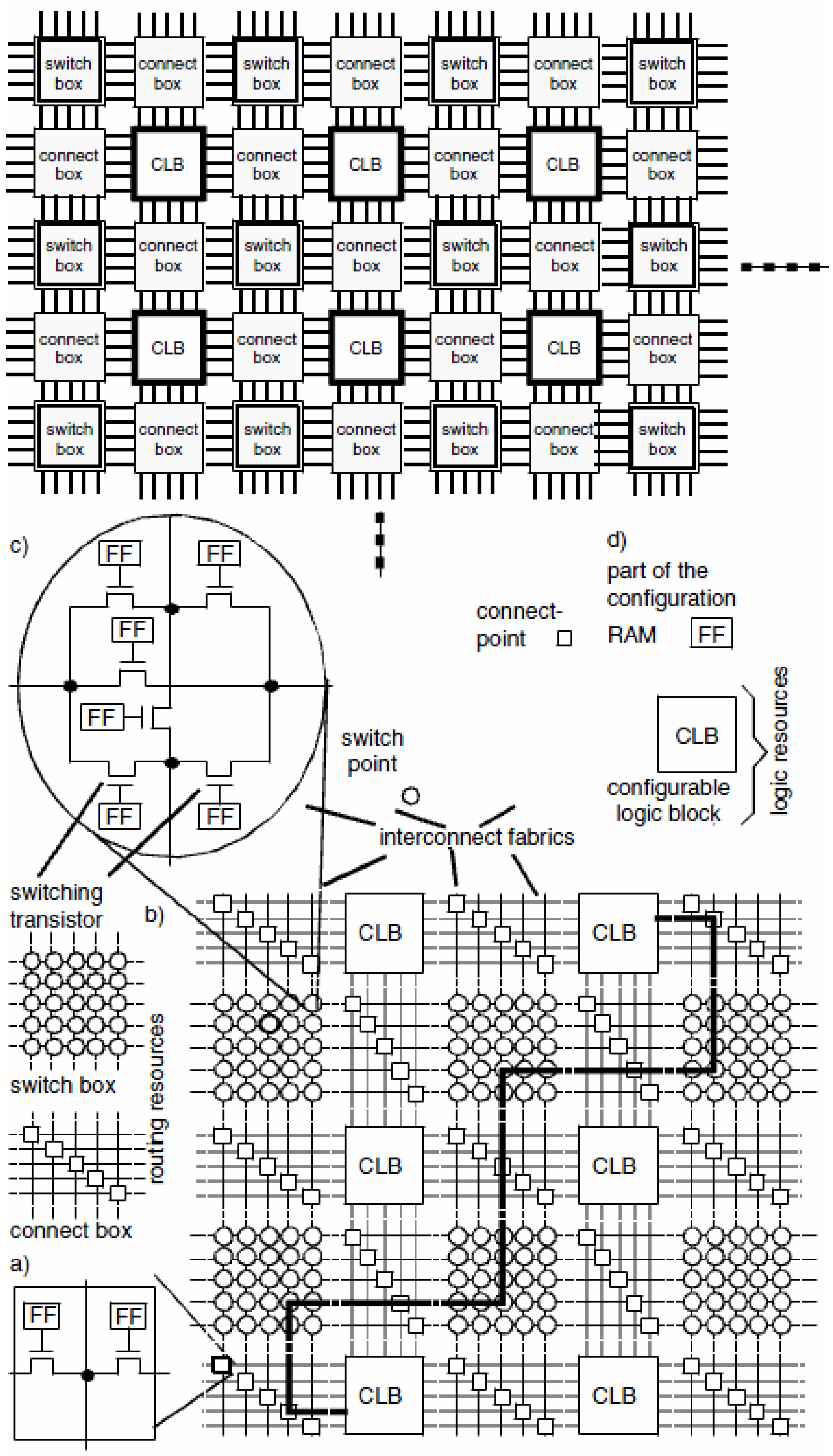

Figure 2: A visualization of an FPGA with CLB "island" architecture. The thick black line in the bottom half of the image is an example of a configware routing program [12].

Implementation

Hybrid Computing

Now that we known the components of morphware and related concepts, we can discuss the actual implementation of the concepts discussed above. As of now, all current implementations of reconfigurable systems fall under one of two categories: hybrid computers or fully FPGA-based computers. Thus, in the interest of finding a way to make reconfigurable systems viable in today's marketplace, the remainder of this thesis will focus on these two categories of reconfigurable systems. Hybrid computers are fundamentally von Neumann machines which integrate reconfigurable technology in some way. For example, some hybrid computers have adopted hybrid-core computing, which is a method by which one or more CPUs in a multiple-CPU configuration is replaced with an FPGA [13]. Another popular method for implementing hybrid computers is to add one or several FPGAs to the system on a PCI (Peripheral Component Interconnect) or PCI Express card, thus allowing the von Neumann machine to utilize it for specific applications that need to be sped up, similarly to the discrete graphics chips utilized in modern video gaming machines [13]. Although both methods are still just as susceptible to the von Neumann bottleneck as any von Neumann machines, as well as being less scalable and less energy-efficient than their counterparts, the hybrid computer solution is still desirable because it allows the average user to experience acceleration of certain common applications without having to give up a familiar interface (i.e., Windows or Macintosh OS) [13].

FPGA-Based Computing

On the other hand, fully FPGA-based computers completely discard the classical von Neumann architecture, replacing all CPUs and interconnections between them with FPGAs, except possibly using a single CPU to interface over a network with other traditional von Neumann machines [13]. These systems completely remove the von Neumann bottleneck, and are generally more scalable and energy-efficient than their hybrid and von Neumann counterparts [13]. Thus, although FPGA clock speeds are currently slower than those of their ASIC counterparts, the removal of the data access bottleneck perpetuated throughout von Neumann systems more than makes up for the difference in speed, making FPGA-based systems significantly faster than traditional von Neumann machines [13]. The major downside to fully FPGA-based systems, however, is their complete incompatibility with the current "big three" operating systems—Windows, Macintosh, and Linux. The issue of providing a familiar operating system interface with comparable functionality to Windows and Linux for reconfigurable systems is addressed in Section 6 of this thesis.

Current Implementation Examples

Cray XD1. One of the most well-known hybrid computers is the Cray XD1. The XD1 was initially created by OctigaBay Systems Corporation, but was branded with the Cray name when Cray bought out OctigaBay in 2004 [14]. The XD1 is divided up into chassis and racks, with up to 12 AMD Opteron 64-bit CPUs per chassis, and up to 12 chassis per rack [14]. The system supports multiple-rack configurations, thus allowing theoretically infinite multiples of 144 CPUs to run in parallel on a single machine [14]. Although the potentially massive bottleneck in such a machine is largely alleviated by proprietary hardware built on top of its fundamental von Neumann architecture, the XD1 does utilize FPGA technology in the form of application accelerators, which are basically chips or cards similar to a GPU (graphics processing unit) or PPU (physics processing unit) designed to enhance resource-greedy or data-driven applications [14]. The Cray XD1 is compatible with both 32- and 64-bit Linux operating systems [14]. The major improvement of the XD1 over similar von Neumann machines is its interconnect speeds. While the Intel Xeon server line is capable of 1 GB/s output from the processor, its data access speeds are only a fourth of that; the XD1, on the other hand, is capable of 8 GB/s output from its processor, a speed which its reconfigurable interconnections are fully capable of handling [23].

COPACOBANA. A famous example of a fully FPGA-based reconfigurable system is COPACOBANA, the Cost-Optimized PArallel COde Breaker, a co-project of the German Universities of Bochum and Kiel [15]. COPACOBANA was designed for use in parallel computing problems, specifically those related to cryptanalysis [15]. According to the developers (and current owner company, SciEngines), COPACOBANA is capable of cracking any symmetric key encryption method with up to a 64-bit key [15].

Because of its fully-reconfigurable design, COPACOBANA successfully eliminates the von Neumann bottleneck for the majority of its code-breaking applications. However, COPACOBANA's reconfigurable design comes with a number of significant limitations which prevent it from becoming the standard for cryptanalytic computing and force the reintroduction of a data access bottleneck for large applications. Its major limitation is its extremely small onboard memory—it only has several hundred kilobits of memory for each of its 120 FPGAs, along with a small amount of user-inaccessible memory dedicated to the machine's control board. Thus, for applications requiring extensive amounts of memory, it must be connected to external RAM or a hard disk, which reintroduces the bottleneck and significantly slows down the speed of its calculations [15]. Nonetheless, when not using external RAM or hard disks, COPACOBANA is much faster and much more energy efficient than similar von Neumann or hybrid machines [15].

Other implementations. Several other reconfigurable systems designed for specific applications have demonstrated excellent results in terms of calculation speed and operational efficiency, especially when compared to the results of traditional von Neumann machines designed for similar purposes. One hybrid reconfigurable system proposed in 2005 by Zhuo and Prasanna for high performance linear algebra calculations achieved speeds of 2.06 GFLOPs on a single Cray XD1 chassis reconfigured to their specifications; when ported to a reconfigured 12-chassis installation, the same series of calculations ran at 148.3 GFLOPs [24].

Another experimental reconfigurable architecture designed in 2000 for data-parallel, computation-intensive applications (such as cryptography) is MorphoSys [24]. When running the International Data Encryption Algorithm (IDEA), MorphoSys achieved speeds as fast as just 4.5 clock cycles per ciphertext block; in the same year, it took an Intel Pentium II processor 153 clock cycles per block (scaled to match MorphoSys' clock speed) to run the algorithm [25].

These examples of reconfigurable computers, along with a host of other implementations for a wide variety of applications, have demonstrated almost universal improvement in both speed and efficiency over their von Neumann counterparts. In many cases, these implementations are several orders of magnitude faster than the fastest von Neumann machines designed for those applications. However, in order to provide those improvements in speed and efficiency to the average user, developers must first provide robust configware with functionality comparable to that of today's software and an operating system with the services, components, and familiar interface the average user has come to expect.

Configware Development

Once we reach an understanding of morphware and the progress made towards market viability in its current implementations, the next step is to examine what needs to be done in order to make the functionality of reconfigurable systems comparable to that of current computer hardware. The best way to do this is in light of current development standards in the software industry—what can be taken from software development standards and applied to configware to ensure comparable functionality and quality?

Necessary Functionality

Although a number of very powerful reconfigurable systems implementations currently exist (see Section 3), most of them are specifically designed for a specific type of application (e.g., COPACOBANA is a code breaker). In order to ensure current market viability, a more all-purpose reconfigurable computer must be designed which can compare in range of functionality to current von Neumann systems. Such a design must begin at the hardware level, requiring configware that both allows the user to take advantage of the improvements inherent in the reconfigurable architecture, and reproduces the capabilities of current hardware. Note that since both implementations of reconfigurable computers discussed in this thesis (hybrid computers and fully FPGA-based computers) utilize FPGAs as the basis for their reconfigurability, there is no need to address flowware development, since neither FPGA-based computers nor hybrid computers utilize flowware for their reconfigurability.

Configware and Software Development

Although various methods for developing configware have been put forth, including a method for concurrent development of software and configware for certain kinds of reconfigurable systems, configware engineering is generally performed by the same rules and standards as software engineering [16, 17]. This is because a number of the same parameters that determine quality software apply to configware as well; for example, clock speed (the time it takes to complete a task) is vital to all real-time systems, be they von Neumann- or morphware-based [18]. Another important design consideration common to both configware and software is parallelism, a concept which is central to configware development due to the highly parallel nature of the morphware paradigm [18].

Although these and other design considerations make the software life cycle ideal for configware development, configware engineering naturally emphasizes certain phases differently than software engineering [12]. Specifically, configware engineering emphasizes testing more than most software life cycle models. While the worst-case scenario for a software malfunction is usually a system crash, a configware malfunction may have the potential to permanently damage or destroy the system hardware itself [12]. However, despite configware engineering's emphasis on testing, the other phases of the software life cycle apply equally well to configware [12].

Primary Focus of Configware Development

However, in addition to the emphasis on testing, the configware life cycle may differ somewhat from the software life cycle in the primary concerns of its individual phases. While the requirements phase of the software life cycle is generally focused on gathering detailed information about the expectations of the client, the requirements phase of the configware life cycle must not be solely focused on client expectations, but also on the hardware implementation of the system's reconfigurable accelerators. For software, the design phase involves creation and implementation of data structures to hold data and algorithms to modify the data; for configware, the design phase is focused solely on the implementation of mathematical algorithms to reconfigure the structure of the system hardware, with little to no consideration given to data structures. Although there are further minor differences between the software and configware life cycles in the remaining three phases, compared to the first two they are relatively insignificant. Basically, the differences between the software and configware development life cycles are present because software is primarily concerned with the manipulation of data, while configware is concerned with the manipulation of hardware.

Configware Life Cycles

Section IV determined that configware development should be based on currently existing software life cycle models, with an appropriate extra focus on testing. The next step is to examine each of the major software life cycle models in order to determine which one(s) are suitably testing-oriented, while also being able to handle the low-level development necessary for proper configware engineering.

Trial and Error Model

Although the Trial and Error Model is not as much a software development model as a way of describing poor programming, its effects on configware development must be addressed nonetheless. Trial and error programming often has severe negative consequences for software engineering; however, those consequences are even worse when applied to configware development. As stated previously, configware testing can be expensive in terms of hardware, since a single configware malfunction has the potential to severely damage system hardware. Any life cycle model that not only lacks good initial design to minimize bugs before testing, but also expects malfunctions to occur even after deployment, is fundamentally unacceptable for configware development. Not only does this include trial and error programming, but also any design methodology, policy, or practice which allows developers and programmers to implement a system without proper requirements analysis and design as a prerequisite. Lazy design methods may (occasionally) eventually result in a passable product in the software industry, but the same methods will never succeed in the realm of configware.

Waterfall Model

The second software engineering model is the Waterfall Model. Unlike trial and error programming, the Waterfall Model is easily adaptable to include an emphasis on testing. However, the aspect of the Waterfall Model that makes it most suited to configware engineering are its feedback loops, which provide ample opportunity during and after each development phase to determine the viability of the configware given the hardware implementation of the system's reconfigurable accelerators [19]. In addition, the Waterfall Model's straightforwardness makes it ideal for complicated hardware programming, since it allows the developers to devote less time to following procedure and more time to understanding the requirements of the system hardware [19]. Overall, the Waterfall Model is one of the most configware-friendly software life cycle models.

Prototyping Models

The next two life cycle models are the prototype-based models—Rapid Prototyping and Prototype Evolution (or Incremental) [19]. Although these models are both well-suited to software development, they are somewhat redundant for configware development. Configware differs from software in that, although software's viability may be proven pre-implementation through a proof of concept prototype, configware's viability must be *proven mathematically* before implementation. If even a small portion of configware code is omitted for the purpose of generating a prototype, the reconfigurable accelerator executing that code will not function properly, and depending on the portion of code left out, may even be damaged. Thus, there is no purpose to adopting a prototyping model for the configware life cycle, since any configware "prototype" must be fully functional in order to execute properly.

Reuse-Based Model

The final life cycle model examined here is the Reuse-Based Model, which is fairly neutral when applied to configware development. On the one hand, the Reuse-Based Model does promote building a domain of reusable resources, which is as beneficial for configware programming as it is for software programming [20]. However, due to its focus on reusing software resources, it may be more difficult to adapt to hardware programming, since hardware resources generally improve enough every few years that old hardware is replaced regularly [20]. Thus, the Reuse-Based Model is only optimal for configware programming in situations where the same hardware implementation is reused over multiple generations of a system, in which case an established domain of configware code would be useful [20].

Spiral Model

The next life cycle model is the Spiral Model, which is admirably suited to configware development for much the same reasons as the Waterfall Model. The Spiral Model has a number of advantages over each of the life cycle models discussed in this thesis, specifically in the way it provides repeated, consistent, rigorous testing throughout every step of the development process [19]. Although the Waterfall Model has many strengths, it simply cannot equal the rigorous level of testing provided by the Spiral Model. Likewise, although the Prototyping and Reuse-Based Models do offer some interesting hardware resource-building possibilities, their approaches to testing during the early life cycle phases are not quite rigorous enough for configware development.

There are two major features of the Spiral Model that distinguish it from the other life cycle models discussed here. First of all, it naturally emphasizes testing throughout the development process, which is necessary in any case when dealing with hardware programming [19]. Secondly, its cyclic structure allows for ample feedback during each phase of development, which promotes repeated viability consideration in much the same way as the Waterfall Model [19]. The Spiral Model’s only potential weakness is its complicated structure, which may not lend itself easily to hardware programming [19].

A Spiral Configware Life Cycle Model

Although most of the aforementioned software life cycle models may be applied to configware engineering with minimal modification, certain models (specifically, the

Waterfall Model and the Spiral Model) may be preferable to others simply for their ingrained focus on proper testing throughout each phase of development. However, in the interest of ensuring the robustness and reliability of configware for future mass-market reconfigurable systems, it is best to select the most robust, testing-oriented software life cycle model for adaptation into a configware life cycle model. With these criteria in mind, the Spiral Model is the best software development life cycle model for configware engineering, since it naturally incorporates rigorous testing before and after each cycle, throughout each life cycle phase.

Although the Spiral Model is more complicated than the Waterfall Model, its benefits to product testing through the development cycle easily make up for this minor weakness. Furthermore, the complicated structure of the Spiral Model may easily be offset by requiring familiarity with the Spiral Model and reconfigurable systems as a prerequisite for all configware development team members. Ultimately, requiring use of the Spiral Model, rigorous testing and maintenance policies, and an experienced development team will provide a solid basis for the development of the reliable, robust configware necessary to make reconfigurable systems viable in today's market.

Morphware Operating Systems

Once basic hardware functionality is guaranteed by robust, reliable configware, the next step is to design the interface between the hardware and configware and the user— a reconfigurable operating system. For hybrid reconfigurable systems, this is not an issue, as they may be designed to support current operating systems due to their fundamental von Neumann architecture. However, it is much more of a challenge for fully FPGA-based reconfigurable systems, as there are currently no personal operating systems like Linux or Windows designed for them. With that in mind, the rest of this thesis will focus on the important tasks, services, and components necessary for a reconfigurable operating system comparable to those for current von Neumann systems.

Operating System Tasks and Services

Loading and executing programs. Developers of operating systems for reconfigurable computers must take two major factors into consideration when implementing the program loader. First of all, unlike loading a program into RAM on a von Neumann computer, loading a program into an FPGA causes it to be executed immediately. Thus, programs in reconfigurable systems may not be pre-loaded into the same FPGA while another program is executing in that FPGA, as with von Neumann systems [21]. Morphware’s speed increase over traditional von Neumann systems helps to offset this issue somewhat, as does increasing the parallelism (number of FPGAs) of the system [21]. However, developers are still trying to find a solution to this issue for applications that require many programs to be loaded very frequently (i.e., modern personal computing) [21].

Secondly, programs in reconfigurable systems involve both logic circuits and embedded RAM, which upon loading the program, must be placed on the FPGA and routed to another circuit or bus [21]. The placement and routing algorithms used by the FPGA are usually fairly limited due to the tradeoff between time and optimization during program loading; the more time the FPGA has to load the program, the more optimized the logic circuit’s path to its destination, and vice versa [21]. This is a somewhat more of a problem than the previous issue, since it limits programs from being loaded and executed immediately following one another, which is the most obvious solution to the first problem. Programmers are still working on a way of overcoming these two issues [21].

Program scheduling. Program scheduling in reconfigurable systems is also very different from its von Neumann counterpart, primarily because morphware programs lack the traditional fetch, decode, and execute cycle. Thus, preemption in the traditional sense is not possible, except for preemptive deletion on some FPGAs [21]. As a result, programs on most reconfigurable systems will continue to execute indefinitely unless the program's designer specifies a point of completion within the program. On a reconfigurable system, multiple programs may run concurrently so long as the FPGA(s) have enough space to allow for more partitions or the programs are small enough to fit into the remaining space [21]. Thus, if there is enough space for all the programs to run in, there is no need for scheduling; however, the operating system must still provide scheduling services for cases in which there is not enough memory for all waiting programs to run [21].

Virtual memory. Although virtual memory on reconfigurable systems is conceptually similar to virtual memory on von Neumann machines (e.g., the program is broken up to fit into several smaller available spaces in memory), there are a few added considerations which make it more complicated. The major difficulty is that dynamically partitioning a program to fit it into leftover (unpartitioned) space on the FPGA is extremely difficult due to the fact that each chunk of the program must output intermediate results that can be entered with the next chunk of the program elsewhere on the FPGA [21]. This situation is complicated by the fact that locality of time and reference are currently inapplicable to reconfigurable systems due to the two-dimensional design of FPGAs [21].

Cache management. Cache management in FPGA-based reconfigurable systems is handled in much the same way von Neumann machines handle it. However, instead of traditional cache memory, FPGA-based systems utilize a hierarchy of high-speed FPGAs and RAM memory. The operating system is then responsible for placing and routing circuits to optimize the access speeds of the FPGA/RAM caches and ensure that the fastest available cache memory is used at all times [21].

Input and output. In reconfigurable systems, programs may process input and output directly through direct memory access (DMA) capable hardware, which the program appropriates for the duration of its operation [21]. This does not result in any interrupts or preemption due to the aforementioned lack of a traditional processor cycle.

Interprocess communication. Although there are many possible methods for implementing interprocess communications on a reconfigurable system, perhaps the simplest method is to connect applications through the wires in the FPGA [21]. In such a system, results of program operations would be stored in embedded RAM or registers for later recovery. Such a system is desirable because of its similarity to the system call interface implemented in most von Neumann computers. However, currently application programmers have not agreed upon a standardized interface for hardware-implemented interprocess communications [21]

Operating System Components

This section explains the four major components of a morphware operating system—circuit allocation, dynamic partitioning, circuit placement, and routing. Figure 3 demonstrates how these four components interact with one another (circuit allocation is divided into detecting space and blocking in the figure). When a program is entered into the reconfigurable computer, the operating system must detect whether or not there is space in any of its FPGAs for the program to be placed [21]. Until it finds space for the program, the operating system must keep blocking the program from being executed [21].

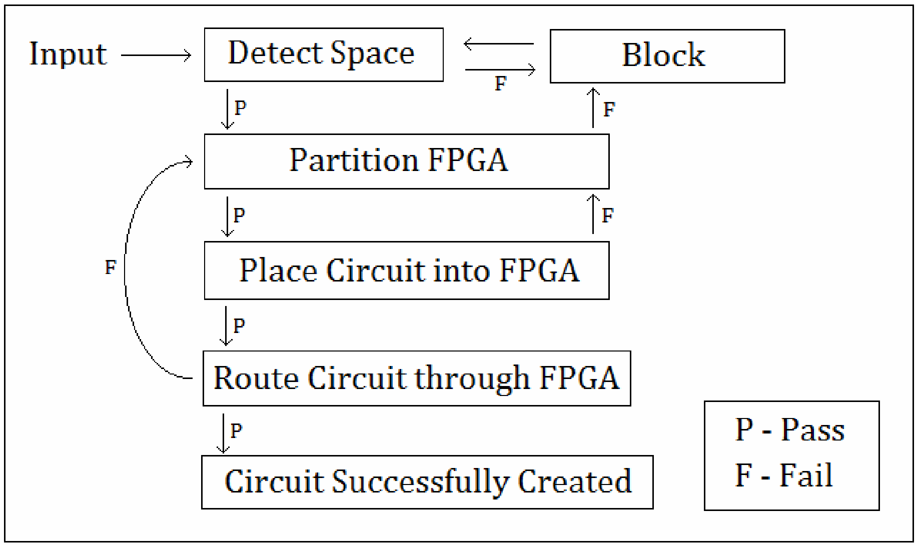

Figure 3: Morphware operating system architecture (edited from [21])

Once space becomes available on one of the FPGAs, the operating system must tell the configware to partition off the CLB(s) or parts of the CLB(s) necessary for the program to execute [21]. Then, the operating system tells the configware to place the circuit into the partitioned CLB(s) and route the path the program will take through the CLB(s) to its destination [21]. Once the circuit is successfully created, the operating system allows the configware to release the program into the FPGA, and the program executes [21].

Circuit allocation. At any given time, some of the nodes in an FPGA will be allocated to various programs, and the remainder of the nodes will be free. Circuit allocation is the component of the operating system that determines which nodes are free and manages the allocation of various program circuits in an FPGA [22]. There are two primary methods for implementing this function. The first method utilizes an operating system table that is updated with available space information every time a new circuit is added to an FPGA [21]. In the other method, the operating system would check the FPGA for sufficient free space each time a new circuit is allocated [21].

In either method, if there is not enough space to allocate the circuit, the operating system may handle it in two ways—by blocking the new circuit and waiting for space to be freed up, or by splitting up (partitioning) the new circuit to fit it into the available space [21]. Although the second option makes the operating system significantly more complex, it is preferred for efficiency [21]. The specifics of dynamic partitioning are explained in the next section.

Dynamic partitioning. Dynamic partitioning begins with a representation of the reconfigurable application as a task graph, in which nodes represent functions and directed edges correspond to data dependencies between those functions. The goal of dynamic partitioning is to break up that graph into more manageable chunks, either for simplicity or ease of allocation. Dynamic partitioning algorithms almost always demonstrate a tradeoff between minimization of communication between partitions and program runtime [21]. To date, most of the partitioning algorithms written for FPGAs are designed with the former goal in mind; however, in some cases this goal must be compromised when program runtime is of high importance.

Circuit placement. Once a reconfigurable application has been partitioned and allocated, it must be placed into the FPGA. While allocation merely reserves a chunk of the FPGA for the circuit, placement is the component of the operating system that determines the direct correspondence between nodes on the task graph and cells in the FPGA [21]. As with partitioning algorithms, placement algorithms always involve a tradeoff between FPGA area efficiency and application runtime [21]. Although most placement algorithms tend to focus on maximizing efficiency of placement, in many applications these algorithms need to be adjusted to maintain a reasonable runtime.

Routing. The final major component of reconfigurable operating systems is routing, the process of establishing an electrical circuit between the source CLB and the program's destination—the physical implementation of the circuit [21]. Routing, like the previous two components, inherently involves a tradeoff between path efficiency and placement speed—the more efficient the path, the longer it takes to place it into the FPGA. However, there are two major methods of minimizing this tradeoff. First of all, creating a library of "pre-routed" blocks that implement the most common circuit paths allows the operating system to select one of those blocks and implement it quickly, without calculating the same electrical path every time. The other method involves limiting the possible connections the routing algorithm is allowed to make (e.g., only allow nearest neighbor connections), thus drastically decreasing the number of tests that must be made before the algorithm either fails or succeeds. This method will decrease the number of successful connections, but in many applications the significant runtime speed increase is worth it [21].

Conclusion

Robust, reliable configware, together with a familiar operating system interface with functionality comparable to the "big three" operating systems used in today's personal computers, will enable reconfigurable systems to compete with their von

Neumann counterparts in the consumer market. In addition, placing reconfigurable systems on the same ground with von Neumann machines as far as general purpose functionality will allow for a much clearer comparison between their technical capabilities. Rather than viewing reconfigurable systems as special-purpose supercomputers for very narrow, specific applications, consumers will come to view them as another alternative to the current offerings on the market. Given enough time to realize morphware's technical superiority, consumers will gravitate towards it as costs go down and speed and efficiency goes up. If such a trend occurs and holds true, it is entirely possible that reconfigurable systems will completely replace von Neumann computers within the next several decades.

Current Sources

[1] Intel Corporation. (2005). *Moore's Law* [Online]. Available FTP: ftp://download.intel.com/museum/Moores_Law/Printed_Materials/Moores_Law_2pg.pdf

[2] H. N. Riley. (1987, September). *The von Neumann Architecture of Computer Systems* [Online]. Available: http://www.csupomona.edu/~hnriley/www/VonN.html

[3] Boston University. (2010). *The SyNAPSE Project* [Online]. Available: http://cns.bu.edu/nl/synapse.html

[4] J. L. Hennessy, D. A. Patterson, and A. C. Arpaci-Dusseau. (2006). *Computer Architecture: A Quantitative Approach* [Online presentation]. Available: http://1.bp.blogspot.com/_0sB_kfTI7ig/TOPt-HhRfwI/AAAAAAAAB_0/FB8E-yXGzMY/s1600/ScreenShot132.png

[5] E. Mang, I. Mang, and P. R. Constantin, "Reconfigurable computing – a new paradigm," *Journal of Computer Science and Control Systems*, vol. 2, no. 2, pp. 22-27, 2009.

[6] J. Lyke. (2009, July 6). *Introduction to Reconfigurability and Reconfigurable Systems* [Online]. Available: http://www.ece.unm.edu/reconfigurable/MembersOnly/rs101

[7] R. Hartenstein. (2001, June). *What is Morphware?* [Online]. Available: http://morphware.de/

[8] R. Hartenstein. (2001, June). *Data-stream-based Computing* [Online]. Available: http://data-streams.org/#data

[9] R. Hartenstein. (2001, June). *What is Flowware?* [Online]. Available: http://flowware.net/

[10] R. Hartenstein. (2005, June). *What is Configware?* [Online]. Available: http://configware.org/

[11] Z. ul-Abdin and B. Svensson, "Evolution in architectures and programming methodologies of coarse-grained reconfigurable computing," *Microprocessors and Microsystems*, vol. 33, no. 3, pp. 161-178, May 2009, doi:10.1016/j.micpro.2008.10.003

[12] R. Hartenstein, "Morphware and configware," in *Handbook of Nature-Inspired and Innovative Computing: Integrating Classical Models with Emerging*

Technologies, A. Y. Zomaya, Ed. New York: Springer Science + Business Media, Inc., 2006, pp. 343-386, doi:10.1007/0-387-27705-6_11

[13] K. Bondalapati and V. K. Prasanna, "Reconfigurable computing systems," *Proceedings of the IEEE*, vol. 90, no. 7, pp. 1201-1217, July 2002, doi:10.1109/JPROC.2002.801446

[14] *Cray XD1 Datasheet* [Online], Cray Inc., Seattle, WA, 2004. Available: http://www.hpc.unm.edu/~tlthomas/buildout/Cray_XD1_Datasheet.pdf

[15] *COPACOBANA FAQ* [Online], SciEngines GmbH, Kiel, Germany, 2008. Available: http://www.sciengines.com/copacobana/faq.html

[16] K. Ben Chehida and M. Auguin, "A SW/configware codesign methodology for control dominated applications," in *Proceedings of the 16th IEEE International Conference on Application-Specific Systems, Architecture Processors*, Samos, Greece, 2005, pp. 56-61, doi:10.1109/ASAP.2005.10

[17] R. Hartenstein, "Trends in reconfigurable logic and reconfigurable computing," in *Proceedings of the 9th International Conference on Electronics, Circuits, and Systems*, vol. 2, pp. 801-808, December 2002, doi:10.1109/ICECS.2002.1046294

[18] C. Vickery, "Configware in the computer science curriculum," *IEEE*, submitted for publication.

[19] N. Bezroukov. (2009, August 12). *Software Life Cycle Models* [Online]. Available: http://www.softpanorama.org/SE/software_life_cycle_models.shtml #Waterfall Model

[20] K. C. Kang, S. Cohen, R. Holibaugh, J. Perry, and A. S. Peterson, "A reuse-based software development methodology," Carnegie Mellon Software Engineering Institute, Pittsburgh, PA, Spec. Rep. CMU/SEI-92-SR-4, Jan. 1992.

[21] G. Wigley and D. Kearney, "The first real operating system for reconfigurable computers," *Australian Computer Science Communications*, vol. 23, no. 4, pp. 130-137, January 2001, doi:10.1145/545615.545612

[22] G. Wigley and D. Kearney, "The development of an operating system for reconfigurable computing," in *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Rohnert Park, CA, 2001, pp. 249-250, doi:10.1109/FCCM.2001.43

[23] *Cray XD1 Overview* [Online presentation], Cray Inc., Seattle, WA, 2004. Available: http://www.telegrid.enea.it/CrayXD1overview.pdf

[24] L. Zhuo and V. K. Prasanna, “High performance linear algebra operations on reconfigurable systems,” in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, Washington, DC, 2005, pp. 1-12, doi:10.1109/SC.2005.31

[25] H. Singh, L. Ming-Hau, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho, “MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications,” in *IEEE Transactions on Computers*, Irvine, CA, 2000, pp. 465-481, doi:10.1109/12.859540