Advances in Adaptive Computer Technology

by

Dr.-Ing. Andreas Koch

Tech. Univ. Braunschweig
Department for Integrated Circuit Design (E.I.S.)
Germany

Submitted on 1. December 2004

Abstract

The tremendous progress in microelectronics no longer linearly translates into increased computer performance. While we can build chips with billions of transistors, the processor architectures implemented on them still follow patterns developed in the mid-1940s for vacuum-tube based systems.

As an alternative, we propose to use the silicon real estate to implement processors following a different paradigm: Instead of temporally distributing a computation across shared compute units, it is distributed spatially across dedicated compute units, increasing the parallelism of the computation. Conventional processors and reconfigurable compute units complement each other well: The computation-intense kernels of an application can be spatially mapped to the reconfigurable unit, while the less critical or unsuitable parts remain on the CPU. Together, both processors form an adaptive computing system.

Practical experiments have shown the advantages of adaptive computing systems for a variety of application fields: Very regular signal processing algorithms can profit as well as heavily control dominated applications. Gains can be achieved both in terms of compute performance as well as reduced power consumption.

Lacking, however, are abstractions and software tools to make the potential of adaptive computers accessible to applications programmers. The complexity involved in developing an automatic compiler targeting adaptive computers is significant: In addition to conventional compiler technologies, issues of automatic hardware synthesis both at the architectural and logic level also have to be considered now. Further intricacy is involved in partitioning the application between the conventional and the reconfigurable processor, and generating communications interfaces both in hard- and software. A completely novel topic even in hardware design is the full exploitation of reconfigurability, which in itself can occur in various forms.

This work presents some of the progress that has been made in all of these areas. It compares and contrasts different architectures for adaptive computers and discusses their effects in an applications context. To this end, sample problems from three different domains have been implemented and evaluated.

The experiences gained in this manner have influenced our research on a third-generation tool flow for automating the mapping from a conventional high-level language to a program running on an adaptive computer. We describe specific areas of progress, ranging from abstract models of computation down to the efficient composition of datapaths from hardware operators.

Contents

List of Figures

1 Introduction

Never before in the history of mankind has so much progress been made in so short a time as in the area of microelectronics. Especially the application of this technology to the construction of computers and communication systems has influenced our culture and way of living on a global scope. From roots in military computation [1] to the spreading of ever-more powerful personal computers into both business and private settings, from increasingly capable telecommunications grids to the Internet and wireless networks covering even the most remote areas of the planet, microelectronics systems have literally become ubiquitous.

Their increasing capabilities (high speed, large memory size) and better environmental adaptability (small form factor, low power consumption) can fundamentally be traced to continuous improvements in the areas of chip fabrication and materials technologies. One of the more important characteristics optimized in these fields is the minimum linewidth of the actually fabricated integrated circuit. This is the size of the smallest features that can be manufactured using current process technology.

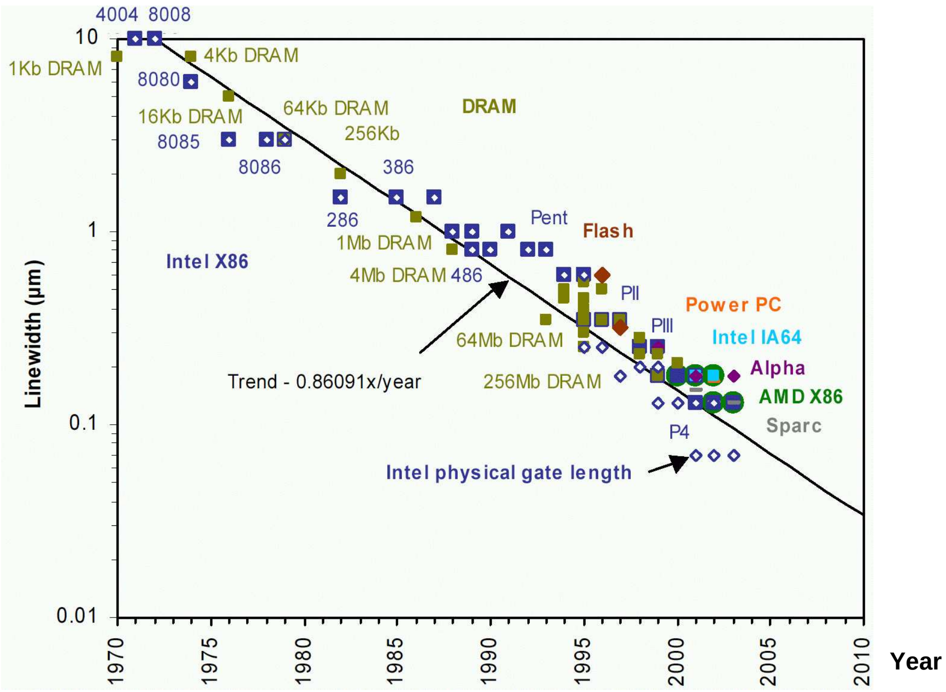


Figure 1.1: Shrinking of IC linewidths [2]

The minimum linewidth has dropped from 100um in 1959 to 2um in 1983 and is currently at 90 nm (Figure 1.1. The next step, a geometry of 65 nm, is expected to be reached by

2007 [3].

1.1 Moore's Law

The shrinking of transistor geometries down into the so-called Deep Sub-Micron (DSM) range allows the integration of ever-higher numbers of transistors onto a single chip. This development, shown in Figure 1.2, has been characterized in 1965 by Gordon Moore, one of the later founders of Intel, as being exponential [4].

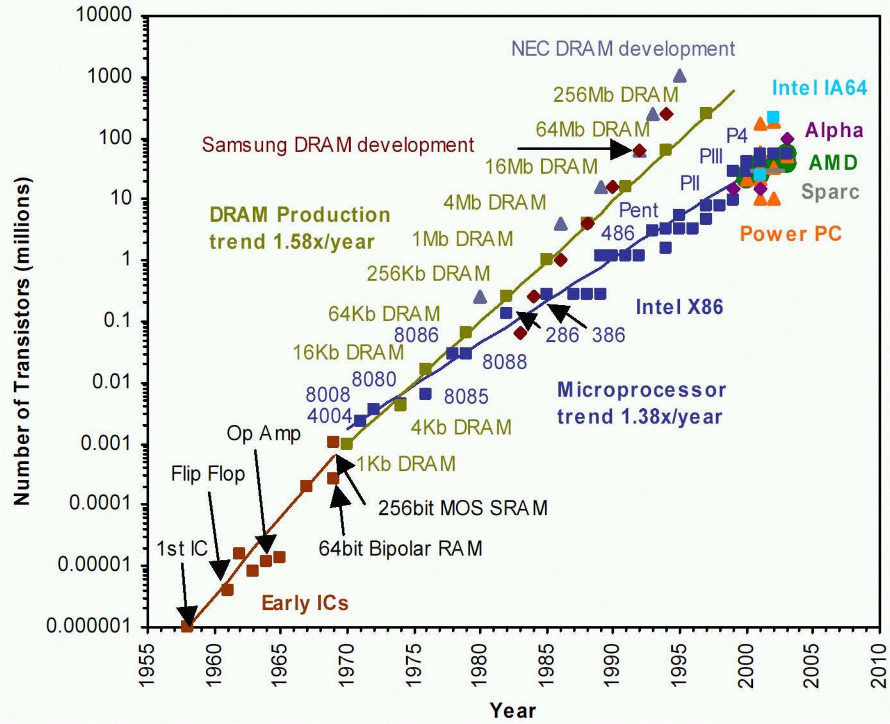


Figure 1.2: Moore's Law [2]

Initially, he predicted the doubling of the number of *minimum-cost transistors* on a chip every year, In 1975, he refined his statement to predict a doubling of the *maximum complexity* of chips (no longer considering the minimum-cost aspect) every two years [5]. The shrinking of device geometries also leads to shorter switching times of the circuits, thus allowing ever faster circuits, with peak clock frequencies of digital circuits currently being in the range of 3-4 GHz. Combined, these effects allow ever more complex, faster, or cheaper chips to be implemented.

1.2 Chip Technology Limitations

However, all of these advances do not come without a heavy price.

Fab Cost The costs to actually set up a chip production line for a recent process technology has increased from USD 6 million in 1970 to more than USD 2 billion (10^9) in 2002. For the upcoming 65 nm processes, it is expected to reach as high as USD 10 billion [6]. It will be increasingly difficult to amortize the cost of the production line over the lifetime of the technology.

Mask Cost The costs to actually tool the production line for the fabrication of a specific circuit design have also risen exponentially. For the 250 nm process prevalent in 2000, a mask set cost on the order of USD 100,000. For a currently state-of-the-art 90 nm process, these costs increase to over USD 1 million [7]. To give an impression of the complexity of these mask sets: For a 90 nm microprocessor, 20-25 individual masks are required, described by a data set 200GB in size. Due to their complexity, only 50-70% of the masks created are even suitable for chip production, the rest contains unfixable errors [8]. Even worse, for each design error discovered after fabrication, a new mask set has to be produced. Even for the established 130 nm process technology, 50% of all designs require one or more corrections (re-spins). This situation is expected to deteriorate further for the smaller geometry processes [7].

Design Complexity Both the sheer numbers of available transistors, as well as the increasingly difficult problems faced to actually use them, burden the development of a complex state-of-the-art chip. In 2000, the then-typical design of 5 million gates in a 250 nm technology required an effort of 5 man-years. In contrast, a typical 90 nm design has 80 million gates in 2004 and needs over 200 man years for completion [7]. Note that the chip complexity and the design effort are not proportional. In the DSM arena, many of the abstractions and simplifications used both by human designers and their support software tools (Electronic Design Automation, EDA) no longer hold. Especially in the interconnect, formerly ignored capacitive, resistive and now even inductive effects void many of the prior assumptions. Since these effects are still poorly understood, designers can no longer rely on accurate simulation and verification software tools to "bulletproof" their design prior to fabrication [9]. Instead, costly chip re-spins are often required.

Yields The new processes are also becoming more difficult to optimize for high yields. When the 130 nm technology was introduced in 2002, only those chip manufacturers making a very limited number chip designs (such as mass-produced CPUs) were able to fine-tune the fabrication processes to arrive at yields above 70%. Foundries having to cater to a wider variety of customer designs were only able to achieve yields in the range of 10…20% [10]. No such data is currently publicly available for the 90 nm process, but the situation is expected to be comparable or even worse.

Power With the increasing numbers of transistors on a chip, the power consumption grows, too. For the prevalent CMOS circuit technologies, where a current ideally flows only during switching activity, this has been countered for some time by lowering the supply voltage from the 5V used for decades down to currently 1.2V. Further reductions are necessary due to the shrinking transistor lengths and planned

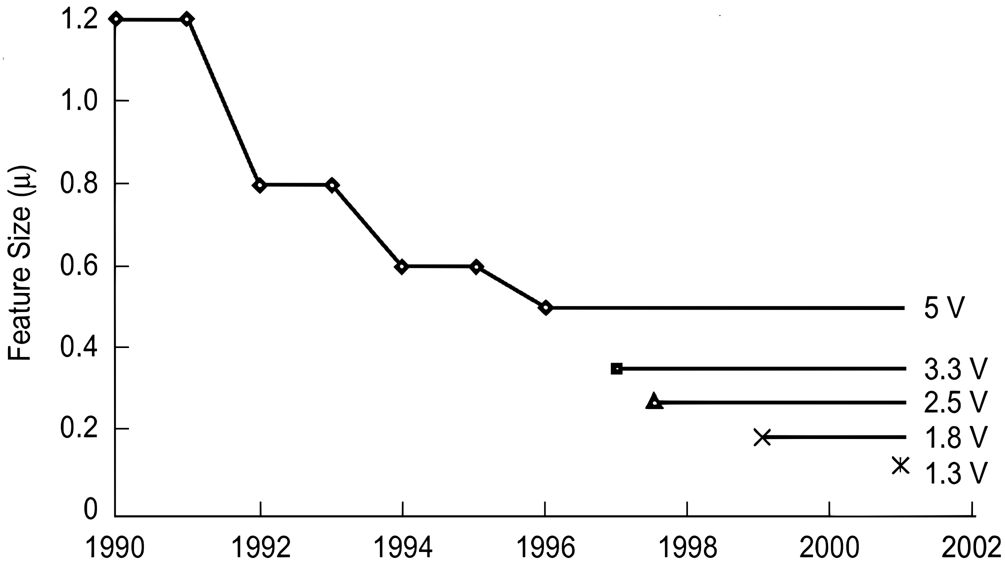


Figure 1.3: Lower supply voltages [11]

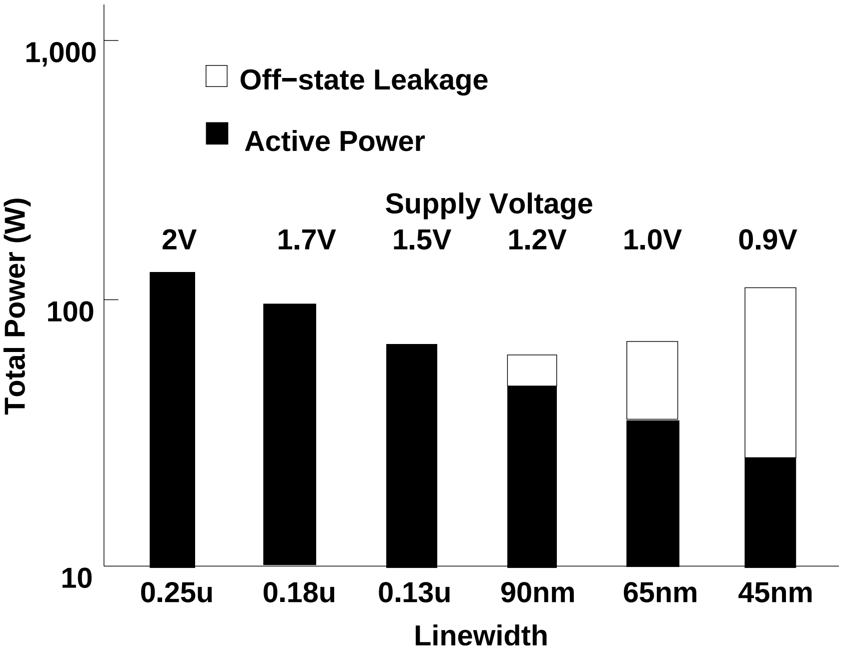


Figure 1.4: Increased off-state leakage current [12]

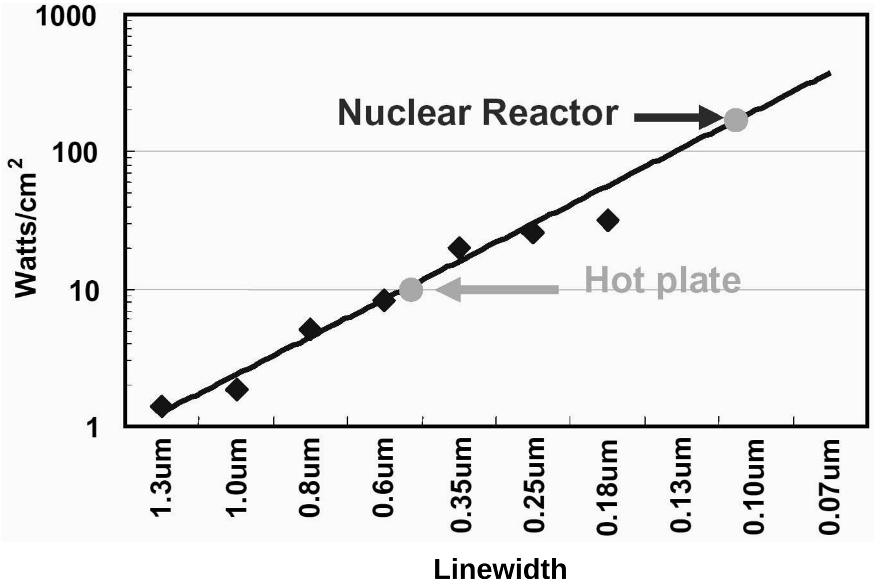


Figure 1.5: Power density vs. shrinking linewidth[13]

to go to 0.7V in the next few years. However, the sheer number of switching transistors has begun to overwhelm the gains achievable in this fashion. Even worse, with device geometries becoming smaller, the transistors are never able to reach a completely switched-off state. This so-called *leaking* leads to a standby power-consumption that begins to dominate over the dynamic switching power [12]. All effects combined (larger numbers of leakier transistors packed into smaller area) also directly lead to significant cooling problems. The power density (as W/cm^2) of common chips has already exceeded that of a hot plate at a comparatively coarse feature size of 600 nm and continues to grow [13].

Technological advances that can be exploited to follow Moore's Law still exist, of course [14]. They includes mask lithography using extreme ultra-violet light (EUV, 13 nm wavelength) [15] and Alternating Phase Shift masks [16]. Materials improvements such as strained silicon (increased electron mobility) [17] and low-k dielectrics (reduced capacitance between metal layers) hold significant promise [18]. However, the difficulties discussed above are still expected to apply and most likely become more serious [19].

1.3 Processor Architecture Development

The increase in computer performance we have experienced in the last three decades was achievable by exploiting advances both in chip fabrication technologies (discussed above) as well as architectural innovations. The latter influence what computation structures are actually realized on the chip(s).

One of the major breakthroughs was the move from sets of complex, but slow instructions (CISC) to sets of much simpler, but very fast instructions (RISC) [20]. All modern microprocessors follow this scheme for their internal operation. Sometimes, due to backwards compatibility concerns, a complex instruction set has to be presented to the programmer. These externally visible opcodes are then translated on-the-fly by the processor core into simpler instructions (sometimes called uOps) that are actually executed in a RISC-like manner [21].

Another innovation was the increase of the number of instructions executed at the same time, lowering the count of cycles-per-instruction (CPI) below 1.0. Since 1985, the most common technique for this is pipelined execution, which allows to simultaneously process multiple instructions in an interleaved fashion, each at a different stage of execution. The basic idea has been refined since with concepts such as dynamic scheduling [20], a technique that reorders instructions on the fly to keep the pipeline running, even if a previous instruction would stop the flow (stall the pipeline). Another approach aims at reducing the effect of branch instructions which disrupt the pipeline. Here, run-time statistics are used to attempt to predict the behavior of branches and feed the pipeline with the correct instruction stream (taken vs. untaken branch) [20].

Beyond this single pipeline parallelism, true parallel execution of instructions using multiple processing pipelines (superscalar execution) has also been worked on since the early 1980s [20]. However, since these parallel pipelines are also sourced from a single sequential instruction stream, even greater efforts have to be undertaken to extract separate instruction sequences that can actually be correctly executed in parallel. To increase this degree of parallelism, more advanced techniques such as speculative execution [20] come into play. This technique attempts to reduce the impact of control dependencies by executing the two (or more) possible branches of a conditional in parallel with evaluating the conditional expression itself. Once the actual value of the conditional has been determined, the results of the appropriate, speculatively executed alternative are committed for future use, while the effects of the mis-speculated alternatives are discarded.

Many of these architectural improvements rely on performing more and more complex analyses of the instruction flow at run-time. In recent microprocessors, the complexity of these analysis units dwarfs the area of the processor actually performing calculations. Consider a recent Intel Pentium IV Prescott core fabricated in 90 nm technology (Figure 1.6 [22]): The parts of the processor responsible for the actual calculations are the two Rapid Execution Engines (containing L1 data cache, ALUs and register files) and the Floating Point and Multi-Media (small vector) unit. The rest of the chip is filled with cache memory and analysis/administrative logic.

Beyond its original domain of transistor counts, Moore's Law has sometimes been extended to computing performance. However, in recent years, it has become apparent that this no longer applicable (Figure 1.7). The efficiency of microprocessors, measured million operations per second per MHz clock frequency per million transistors has been dropping. Despite of all the fabrication technology enhancements, despite of ever more complex processor architectures, the actual gains in computing power become smaller and smaller. And major breakthroughs similar to the CISC-to-RISC transition that would solve this dilemma are currently not conceivable in the processor architecture field. This becomes quite obvious when comparing the announcements for new processors by the major players: Literally all major processor manufacturers are set to release CPUs that

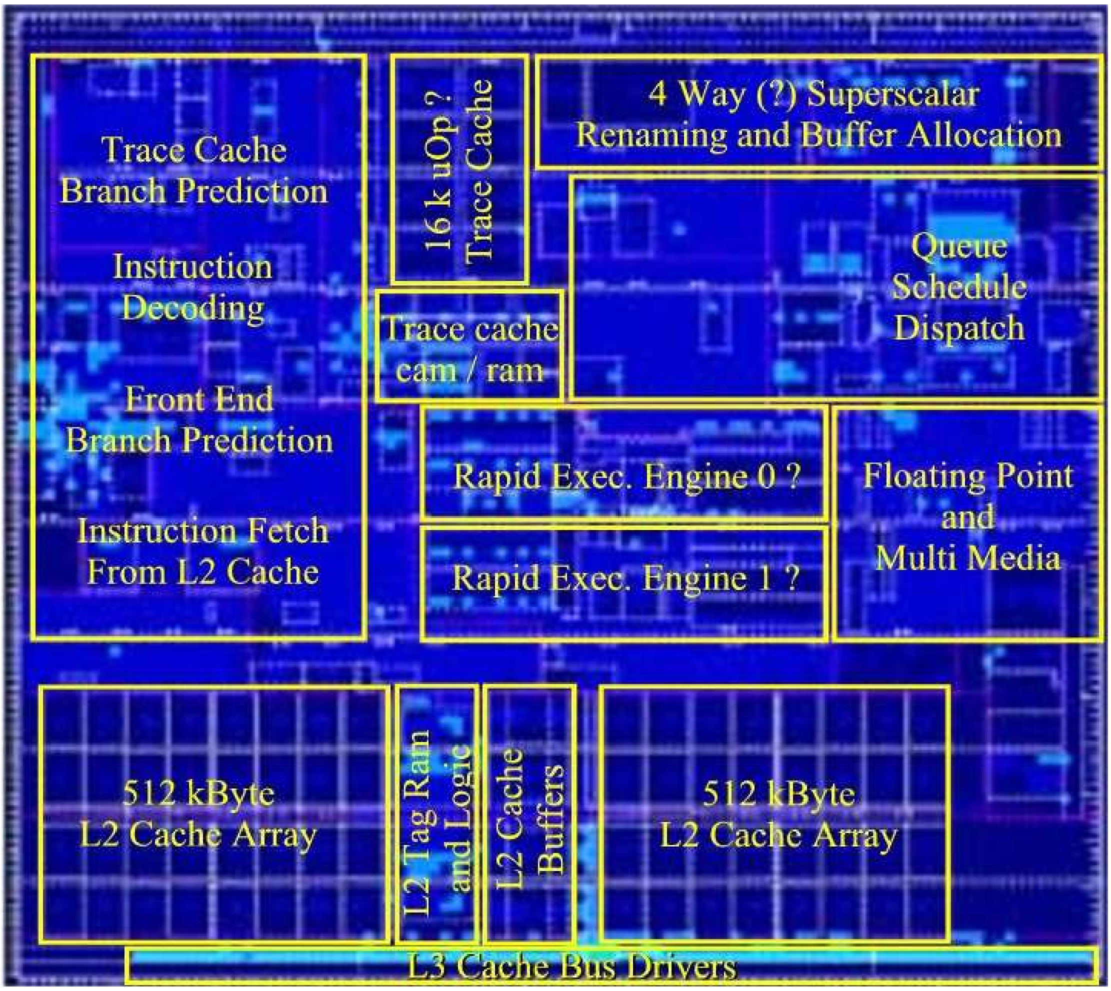


Figure 1.6: Die diagram of Intel Prescott Processor [22]

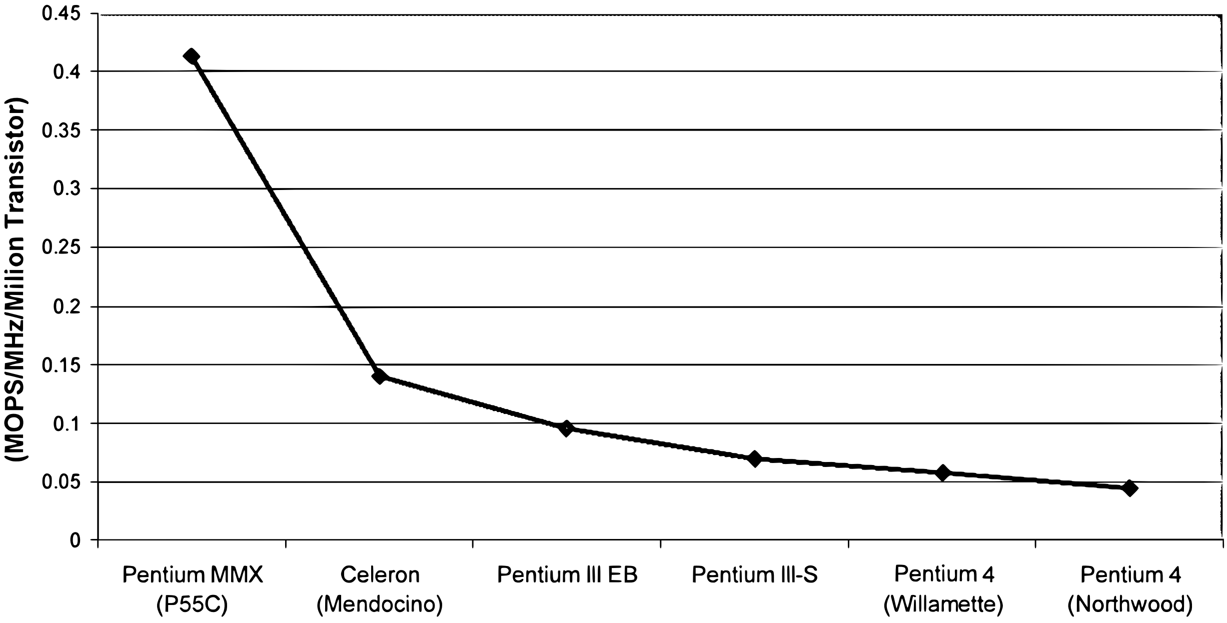


Figure 1.7: Processor efficiency [23]

have two or more separate processor cores integrated onto a single die. Note the absence of true architectural innovation in this approach: Instead, the already established practices are just replicated to fill more than 1.7 billion of transistors [24].

At this stage, we should consider the deeper implications of the issues discussed in the preceding text: Even the very latest general purpose microprocessors are still based on the architectural model first introduced by John von Neumann in 1945 [25]. This has been progressively implemented in computers based on vacuum tubes, discrete transistors, then small-to-medium scale integrated circuits up to the very densely integrated circuits available today. However, the cost model for these machines has changed significantly. The original von Neumann model is based on reusing the scarce hardware resources (ALUs, registers, memory) as much as possible. However, with the exponential growth in available on-chip area, it is no longer applicable.

Processors which no longer adhere to the von Neumann model do exist. In the area of digital signal processing, more specialized architectures have been successful for many years. As an example, a 600 MHz digital signal processor (DSP) can beat a general purpose processor running at 1.4 GHz at a suite encompassing a variety of compute tasks [26] by a factor of 1.5 [27]. The DSP has an internal architecture significantly different from a classical von Neumann machine

Figure 1.8 shows the architecture of this chip, a current TigerSHARC DSP. While the general-purpose ALU still exists, it has been augmented with special function units for address generation and data streaming. Furthermore, in addition to the single external memory bank defined in the classical model, the processor has access to six separate memories, each with their own cache-infrastructures. Four memory accesses can be pro-

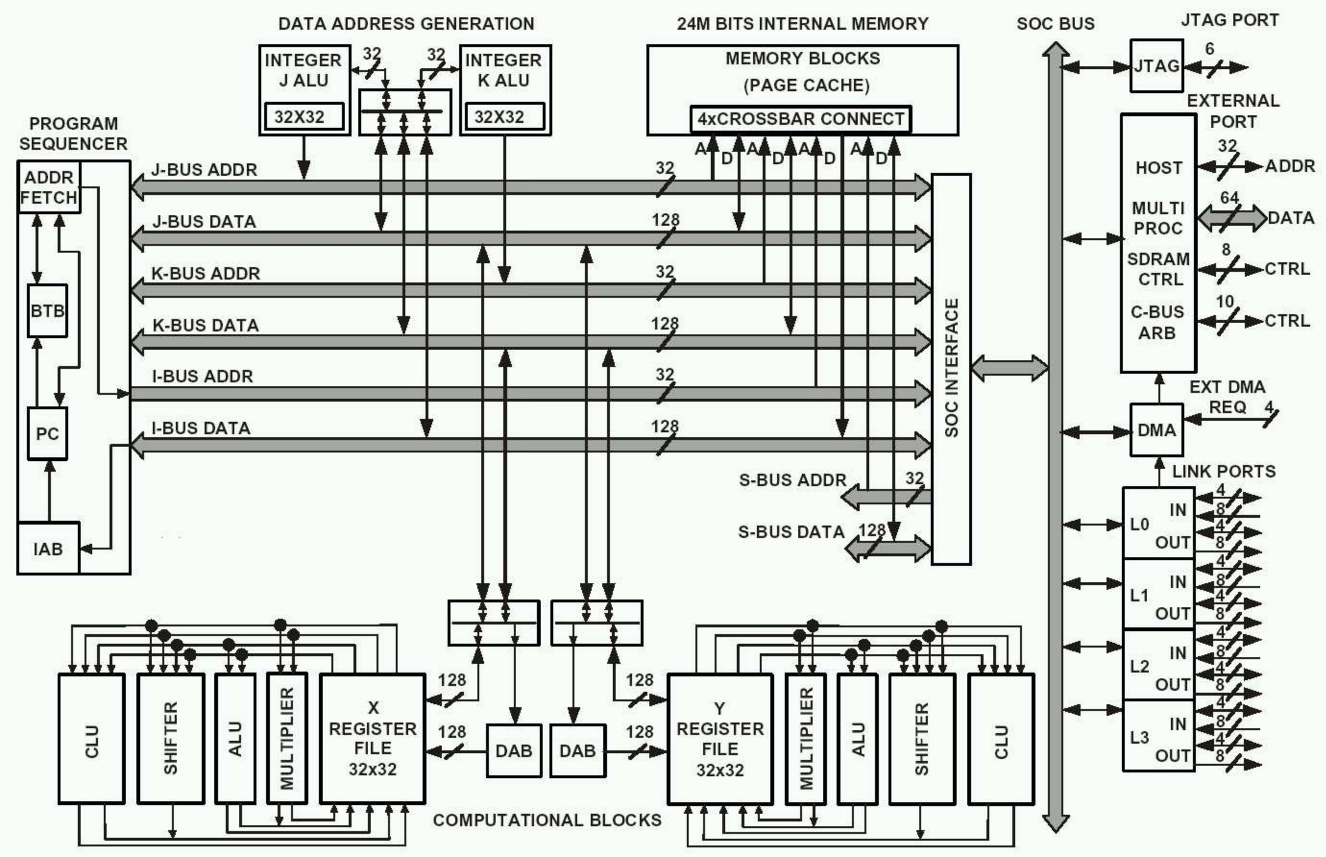


Figure 1.8: Analog Devices TigerSHARC DSP [28]

cessed in parallel by this subsystem, significantly exceeding the capabilities of even the very latest CPUs.

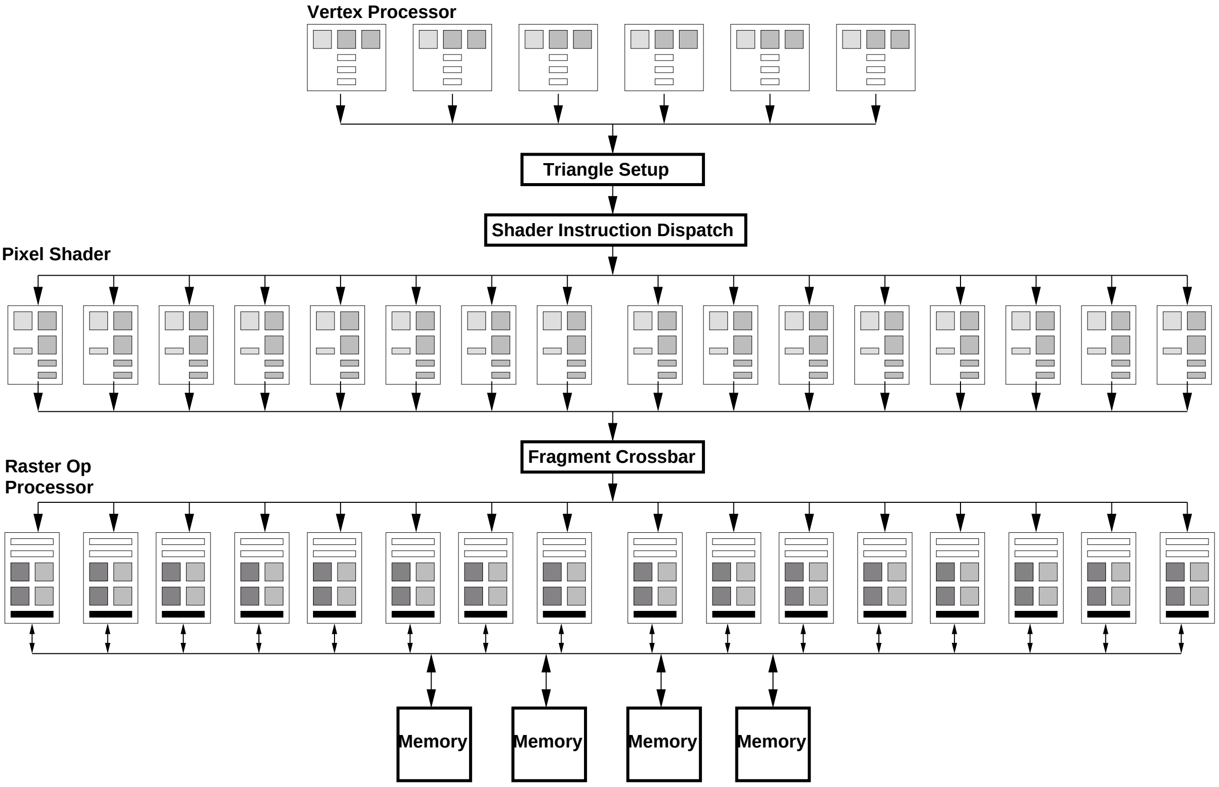


Figure 1.9: Simplified architecture of NVidia GeForce6800 GPU

In the completely different domain of 3D graphics processing, specialized graphics processing units (GPUs) have long since exceeded the performance of CPUs. While early GPUs implemented a fixed-function rendering pipelining with only limited flexibility in the choice of parameters, current models insert freely programmable compute units between the fixed pipeline stages. These vertex and pixel processing units support vector operations up to full IEEE floating-point precision. From an architecture standpoint, modern GPUs are highly parallel streaming processors optimized for vector operations, providing both MIMD (vertex) and SIMD (pixel) compute models combined with very high memory bandwidths. As an example, a current model (Figure 1.9) offers 16 parallel processing pipelines and memory bandwidth of over 35GB/s [29]. In comparison, a sample modern 64b CPU is 3-way superscalar and has a memory bandwidth of just over 5 GB/s[30]. With the freedom offered by the programmable units, work has recently begun to apply this compute power to applications outside of the graphics domain. Initial results are very promising. As an example, a parallel flow simulation using a previous-generation GPU has a speedup of over 8 compared to a CPU-based implementation [31].

The performance improvements of both the DSP and GPU examples given here can be traced to a better matching of the processor architecture to the problem to be solved. This is obviously true for the DSP, specialized to execute variations of well-known algorithms (filters, FFT, etc.). In the graphics arena, a much greater flexibility is required to accommodate a wider variety of different algorithms. Thus, the GPU processing structure, while not limited by the von-Neumann constraints, is more amenable to be used for some non-graphics applications (e.g., evaluating linear expression [32]).

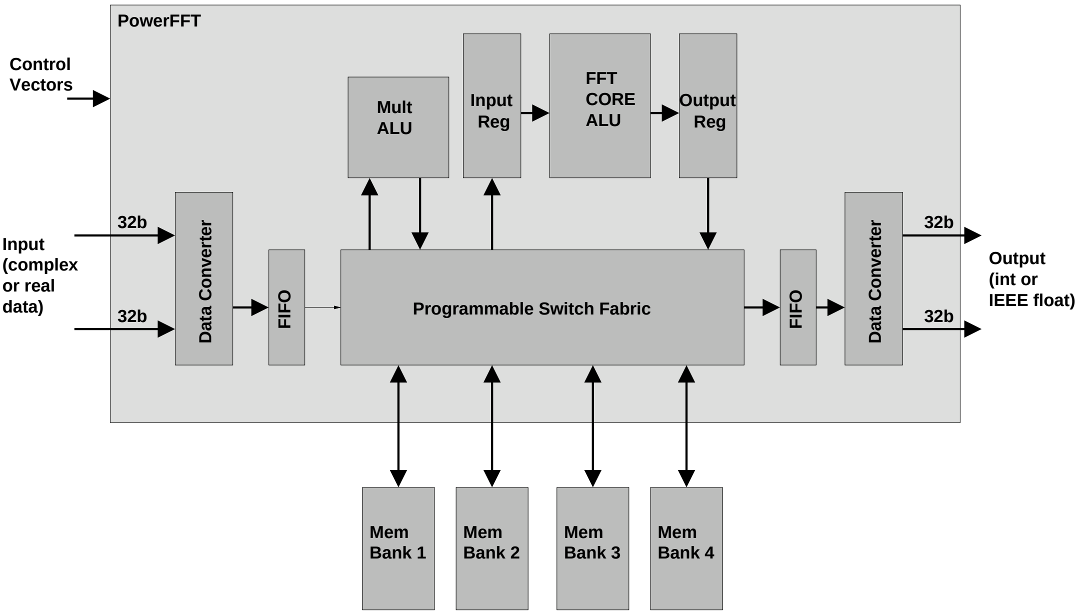


Figure 1.10: Architecture of PowerFFT FFT processor

The perfect match of processor structure to application is, of course an application-specific integrated circuit (ASIC). These devices have been the traditional vehicle of choice when tackling computational problems beyond the capabilities of general-purpose processors. For example, even before DSPs, special ASICs were fabricated to just perform FFTs in the quickest possible manner. Even today, a state-of-the-art DSP [33] running at 600 MHz requires 15.64us to perform a 1024-point complex FFT on 32b floating point numbers. A specialized ASIC (Figure 1.10) running at just 128 MHz requires only 10us for the same task [34].

In all of these cases, the gain in performance achievable by matching the processor architecture to the application is accompanied by a corresponding loss of flexibility. In the extreme case shown here, the device can *only* compute FFTs, no other operations are possible. Ideally, however, we would like to design a computer that can be matched *perfectly* to *arbitrary* algorithms.

2 Adaptive Computers

2.1 Computing Paradigms

When contrasting the von Neumann approach with recent processor architectures that actually did yield significant performance increases, it becomes apparent that the latter all have one characteristic in common: Instead of the heavily shared resources of the von Neumann paradigm, which distribute the steps of a computation *over time*, they use dedicated compute units for each step, preferably in parallel. The resulting structure realizes the entire computation distributed over an area, or more generally, distributed *in space*.

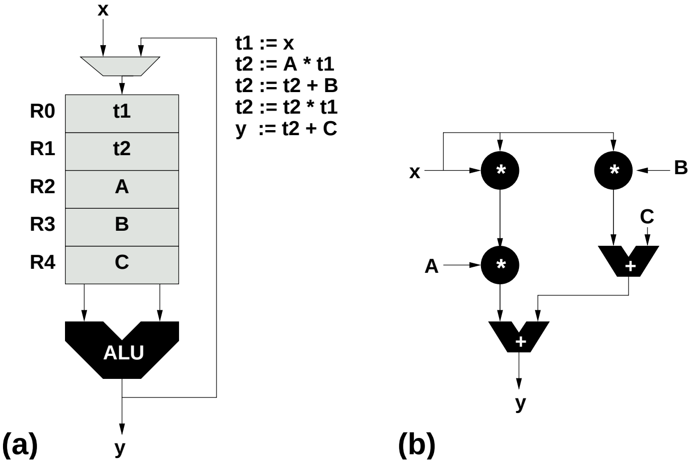


Figure 2.1: Temporal vs. spatial distribution of computation

To illustrate this difference, Figure 2.1.a shows a significantly simplified form a a conventional processor architecture. It computes the polynomial $y = Ax^2 + Bx + C$, where x is a variable and A, B, C are constants. A single universal compute unit realized as an ALU is linked to a register file that holds intermediate results. The entire computation is described by a software program that, at each time step, defines the operation currently performed by the shared single ALU as well as its operand origins and result destination. A variable software program can thus distribute arbitrary computations temporally over the fixed structure.

In contrast, Figure 2.1.b shows the same computation, now spatially distributed across dedicated operators. Input and intermediate data flows from operator to operator, allowing both the parallel computation of intermediate results within one evaluation of the polynomial (e.g., computing x^2 and Bx in parallel) as well as interleaving successive evaluations (e.g., for two input values x_1, x_2, computing x_2^2 and Bx_2 in parallel with Ax_1^2 and $Bx_1 + C$). Not shown in the figure is the coordinating instance that orchestrates the flow of the computation. This *controller* can be realized either in a distributed (local) fashion in each operator ("output a computed result once all inputs have values available") or in a centralized (global) manner ("activate the the adder Add2 one time step after Mul3 and Add1 were activated"). The combination of operators, their interconnections and control scheme then very efficiently computes the required function (the polynomial). However, this structure is limited to performing only the *single* computation it has been composed for. Different computations can only be realized using different structures.

As discussed in the previous section, it is this move toward a spatial computation model that gives specialized processors such as DSPs and GPUs an edge over standard processors. ASICs have long leaned toward the fully spatially distributed implementation, which is the natural view of a problem when designing hardware (multiple transistors always operate in parallel). Furthermore, the concept of spatial distribution is also applicable to resources that are not operators in the classical logic or arithmetic sense. Most importantly, it can also extend to memory (see Section 2.5).

Between these two extremes, a wide spectrum of design choices exists. Modern processors no longer strictly follow the temporally distributed model. For example, superscalar and vector architectures are well capable of processing more than one operation at a time. Conversely, limited available silicon area on an ASIC might preclude a fully spatially distributed implementation, and necessitate operator sharing or non-pipelined multi-cycle operators (that cannot accept a new input while a previous one is still being processed).

2.2 Flexible Computation Structures

Until recently, the practical applicability of the spatial computing paradigm was severely constrained by its inflexibility: Once a computing structure was committed to hardware, that hardware could no longer realize algorithms other than the specific one(s) implemented. At best, a limited set of parameters could be modified to alter specifics of the algorithm. For decades, general-purpose computing thus was the sole domain of software-programmable processors following the temporal distribution paradigm.

However, *reconfigurable* devices allow the implementation of different computation structures even after the device fabrication process. With roots in early solutions such as PAL/PLAs (programmable logic arrays, programmable array logic) that allowed the post-fabrication implementation of logic expressions in a sum-of-products form, a new class of chips known as Field-Programmable Gate Arrays (FPGA) was developed in the mid-1980s These devices combined blocks providing a wider variety of logic functions (e.g., all boolean functions of 4 inputs or less) with interconnection networks able to connect the blocks in a flexible fashion (meshes, trees, etc.). Early reconfigurable devices were fabricated in a 2 μm technology and offered only 64 of these configurable logic blocks (roughly equivalent to 800 gates on a fixed-function ASIC). As such, they were mainly intended to replace sets of discrete 74-series chips (in so-called large-scale integration technology), each integrating a handful of simple logic gates. The main application of these first FP-

GAs was the realization of interfaces between more complex components, a role called *glue logic*. However, in the early 1990s, the logic capacities of configurable devices had advanced into almost a thousand function blocks per chip. By this time, the first reconfigurable systems intended not for interfacing, but for actual computing purposes were built [58] [73]. This type of machine is now commonly called an *adaptive computing system* (ACS). For further study, a more detailed history of the technology can be found in [41] and [42].

2.3 Terminology

Despite the comparatively short history of the field (less than 20 years), the terminology used by different authors has become somewhat murky. This section will alleviate this situation by clearly defining each term.

2.3.1 Definitions

Configurability means the ability to *structurally* adapt a compute unit to a specific problem or problem set.

Reconfigurability indicates the capability to perform the configuration ability *more than once*. This generally implies that configuration can be performed on already-deployed hardware (post-fabrication).

Dynamic Reconfiguration describes the process of reconfiguring the device for another computation structure *during the execution of an algorithm*. This is sometimes referred to as run-time reconfiguration.

Partial Reconfiguration applies the reconfiguration process just to *part* of the device. This generally allows shorter reconfiguration times and can possibly occur with the unaffected parts of the device continuing to operate.

Specialization refers to the adaptation of the computation structure not only to the current algorithm, but also to the specific parameter values used for the *current execution*. For example, once the coefficients have been determined for a run, a specialized filter uses smaller and faster constant value multipliers instead of general-purpose variable value multipliers.

Programming is the process of varying the behavior of a device while *preserving* its computation structure. This covers the traditional software programming of CPUs as well as, e.g., changing to a different operating mode by writing a new value into a device register.

Granularity is a measure of the functionality of *individually configurable* blocks. This can range from individual transistor pairs (no longer in use) over the lookup-table (LUT) structures common to FPGAs to coarser granular structures such as multi-bit ALUs or even complete processors.

Binding Interval is the minimal time between two *changes in device function*. Practically, it can reach from infinity (a non-reconfigurable ASIC) down to a single clock cycle (a CPU that changes the currently executed function after each instruction).

2.3.2 Discussion

Current FPGAs generally support only reconfigurability, but not dynamic reconfiguration (the reconfiguration process is too slow). But hybrid approaches do exist. For example, a processor core may be *configured* pre-fabrication in terms of bus widths or special instructions. Additionally, such a device may then also contain a *reconfigurable* compute unit (RCU) to accelerate some applications [35]. ASICs with a mostly hardwired structure may allow the *reconfiguration* of a limited number of individual logic elements [36]. And software instructions may alter the operation of a device by *programming* new data values into the registers of a *reconfigurable* unit. In general, the latter is preferable to even *partial* reconfiguration since it is much faster.

In this scheme, FPGAs and CPLDs are considered to be finely-granular devices (due to their single-bit operators and individually routable wires). The more recently introduced class of coarser granularity devices is sometimes called reconfigurable (or field-programmable) function (or node) arrays (RFA/FPFA, RNA/FPNA). In this work, we will use the first term.

The *binding interval* often depends on the *granularity*: A coarsely granular device in general requires much less configuration data to describe its function than a finely granular one. For the spatially distributed computation paradigm, shorter binding intervals are preferable, since they allow a better reuse of the comparatively expensive (in terms of silicon area required) reconfigurable resources. However, the extreme case of completely reconfiguring a device each clock cycle has proven unsuccessful in practice (the immense number of transistors switching during reconfiguration causes *severe* power dissipation / cooling problems) [37].

Note that reconfigurable computing structures not implemented using FPGAs can and do exist. While FPGAs have been in use longest in this role, their fine granularity is not optimally suited to handling the word oriented applications commonly encountered. Furthermore, the glacial reconfiguration speed of most FPGAs (on the order of 100ms for larger devices) often precludes the use of dynamic reconfiguration.

Recent improvements in FPGA architecture are so-called *system FPGAs*. Their on-chip memories, multipliers and even complete processors allow a higher system integration density which, while also useful for implementing ACSs, still does not address the issue of efficient dynamic reconfiguration. Only the still-rare RFA-based compute units currently do allow the practical use of dynamic reconfiguration in real applications. For example, a CDMA2000 Rake Finger can be realized efficiently on a small RFA by reconfiguring the device 57000 times per second [38].

However, despite their many advantages, coarse device granularities may not always be desirable. For example, finely granular devices are better *specializable*, since the values of individual bits of per-run constants (e.g., filter coefficients or a crypto key) can directly be folded into the hardware realizing the computation structure (see Section 3.4.3).

For simplification, the computation structures executing on the RCU (however it may be realized) are often referred to only as “hardware”, even if they are fundamentally just a data stream for configuring the RCU fabric.

2.4 System Architecture Considerations

While an RCU could well be used as the only processor in a computing system, this approach is not efficient for general-purpose applications. In practice, many programs spend most of their computation time in relatively small sections of code. The rest consists of less performance-critical tasks such as I/O, memory management and error handling. While the RCU could also implement these operations in a spatially distributed fashion, doing so would require valuable reconfigurable space for computations that are executed rarely if ever (compared to the computation-intense *kernels*). Thus, such functions are better provided by a conventional processor. Since the application kernels are executing on the RCU, this CPU does not even have to be a high-performance processor.

An ACS is thus characterized as having both a conventional fixed-structure CPU satisfying the base demands for computing performance and an RCU for handling the peak demand. Considerable freedom exists, however, in how to implement a complete system combining these two components with memories and I/O devices. The following will briefly discuss various architectural choices. It is a continuation of the discussion begun in [39] [40], now annotated with current specifications and using the terminology introduced in [41].

2.4.1 Stand-alone RCU

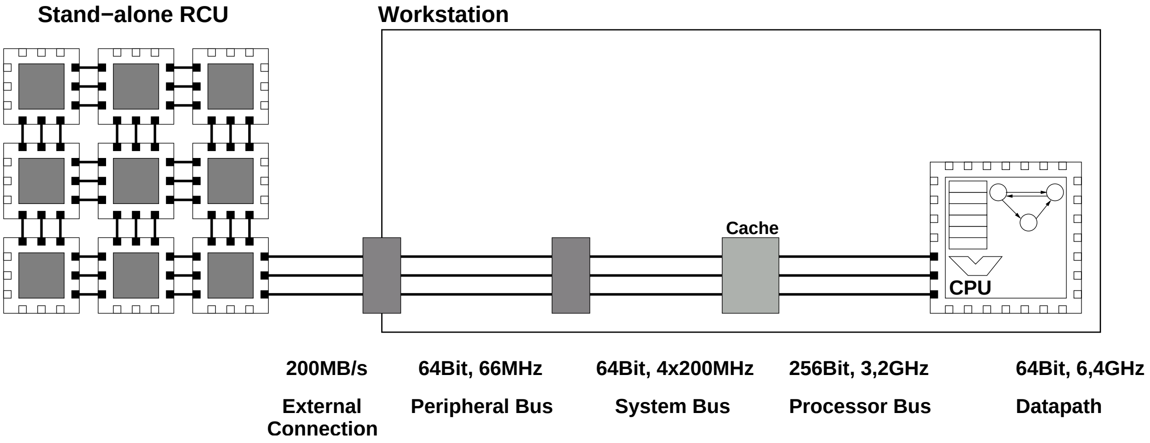


Figure 2.2: Stand-alone RCU

The *stand-alone RCU* shown in Figure 2.2 is generally not used to realize a system for computing purposes, but for *emulating* logic circuits that are to be implemented later in an ASIC, hopefully discovering all errors before the first fabrication run. For this purpose, the most important system capability is a very large reconfigurable capacity. Older emulation systems often used large arrays of stock FPGAs to hold the circuit under emulation, but more recent ones rely on custom processors specialized for ASIC emulation. Current systems, such as the Cadence Palladium series [43], have a capacity of up to 128 million gates. The external enclosure holding such an RCU, weighs 1.3t and requires almost 16 KW of three-phase power. It communicates with the conventional CPU of a host workstation using an FC-AL interface providing a transfer rate of 200 MB/s. Due to this relatively low data transfer rate and the associated communications latency of hundreds of microseconds, such a stand-alone RCU is suitable only for applications with limited data exchange with the CPU followed by long periods of intense computation one the RCU.

2.4.2 Attached RCU

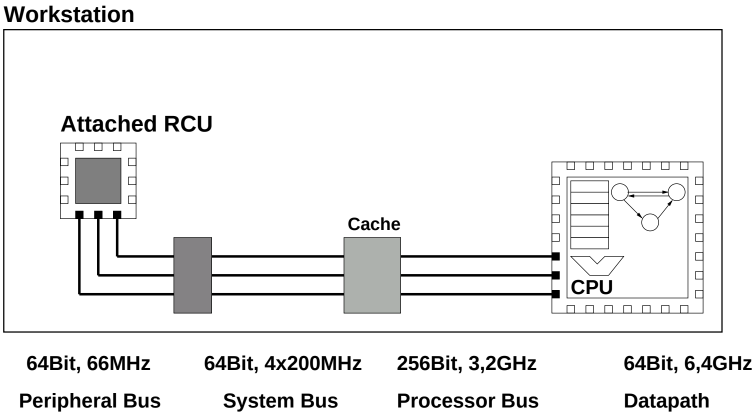


Figure 2.3: Attached RCU

Today, the most common method to realize an ACS is to attach the RCU to the peripheral bus of conventional computer (Figure 2.3. Moving away from legacy technologies such as VME and SBus, today the dominant busses are variations of the PCI standard. These have raw data transfer rates from 32b@33MHz to 64b@66MHz, with extensions specified at up to 64b@533MHz (PCI-X 2.0). However, for some applications, latency can still be a problem. Due to inefficiencies in practical PCI implementations, a memory write via PCI can still take 10 clock cycles, a read even more than 30 cycles. The same experiments also measured a peak achievable transfer rate of only 75% of the specified maximum in a real PC-architecture host. Despite these limitations, the ease of use (simple plug-in cards, wide choice of hardware) of such an *attached RCU* has proven quite persuasive in practice. Section 2.9 will discuss an example of an attached RCU in greater detail.

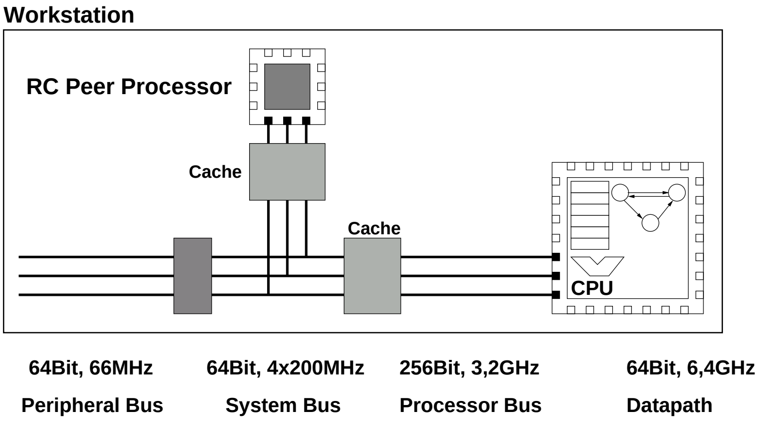


Figure 2.4: Reconfigurable Peer Processor

2.4.3 Reconfigurable Peer Processor

A novel approach to achieving integration tighter than offered by the attached RCU is the *RCU as peer processor* (Figure 2.4). Relying on commonly available SMP motherboards, one or more of the conventional processors could be replaced with an RCU. These would then be connected to the 64b@200MHz quad data-rate (four data items transferred per clock cycle) system bus for a maximum bandwidth of 6.4 GB/s [44]. In an RFA-based RCU, the interface required to communicate at these speeds would be realized as a hardwired block. However, the very latest FPGAs *are* actually capable of implementing the fast interface logic using reconfigurable blocks. The RCU peer architecture would be significantly more difficult realize than the attached RCU, both due to the electrical intricacies of such a high-speed design as well as the more complex SMP bus protocol. However, it would combine the ease-of-use of the attached model (an RCU could easily be added to a commodity dual-processor base system, turning it into an ACS) with the significantly increased communications bandwidth of the peer architecture.

The first practical realization of such an architecture is the Cray XD1 [45]. However, it eschews the commodity SMP motherboards for custom board designs that not only connect the FPGA to *two* processors on a single board with a 3.2 GB/s link, but also allow multi-board configurations with 8 GB/s links between boards. All communications are managed by a scalable distributed switching fabric.

2.4.4 Reconfigurable Co-Processor

An even tighter integration between CPU and RCU requires the integration of both units on the same chip or device (e.g., as a multi-chip module), with the RCU acting as a *co-processor* of the CPU. In this model, the RCU shares the cache with the CPU (though possibly only the L2 or L3 stages). While this sharing could lead to access conflicts between the two units, these should not occur in practice: In the ACS model, the CPU and the RCU generally do not operate in parallel. Compute kernels are ideally processed just by the RCU, with the CPU being dormant, while administrative tasks performed by the

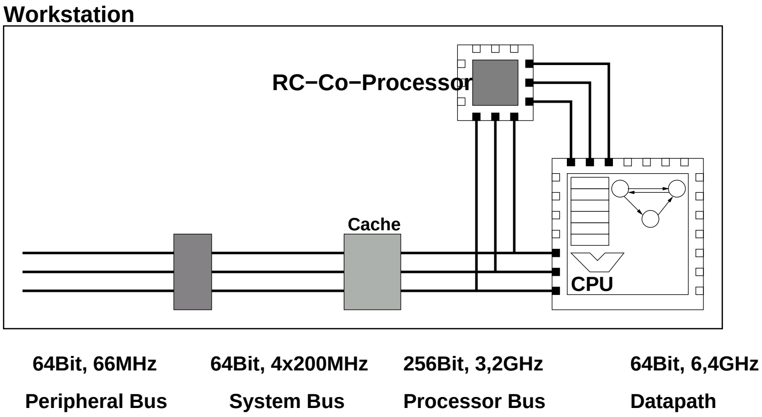


Figure 2.5: Reconfigurable Co-Processor

CPU do not require the intervention of the RCU. This quasi-exclusive access to the shared cache also eliminates the need for the possibly complex coherency protocols that must be realized, e.g., in the peer processor model. The co-processor model has been used successfully in academic research chips such as [46] and is now slowly finding its way into commercial offerings [47] [48]. The maximum available bandwidth available to the co-processor in current CPU designs is 256b@3GHz, with an L1 latency of 3-4 cycles and a L2 latency of 28 cycles [49].

However, operating at these frequencies is far beyond configurable logic today: The very latest FPGAs are designed for 500 MHz operation [50], while the fastest RFAs [47] operate at 2 GHz and access their caches at just 1 GHz. While this appears to make the RCU co-processor model unsuitable for an ACS realization, remember that an ACS does not require a high-performance CPU. Thus, a more realistic realization would combine an RCU coprocessor with a less powerful, but much smaller CPU. In [40], an implementation using the very compact ARM7TDMI core was suggested. On a 130 nm process, this would allow 133 MHz operation when combined with a cache shared between RCU and CPU [51]. More recent developments couple an ARM11 CPU with an RCU co-processor, allowing the resulting ACS to perform MPEG-2 decoding at just 200 MHz instead of the 650 MHz the CPU would require unassisted for the same task [48]. If the actual fabrication of a custom chip for the ACS is an option, the co-processor model still remains attractive especially for embedded applications.

Section 2.9 will examine an architecture for an RCU coprocessor emulated as an attached RCU.

2.4.5 Reconfigurable Function Unit

The extreme case of tight integration moves the RCU directly into the processor as a *reconfigurable function unit* (RFU), shown in Figure 2.6. This has been suggested quite early in in the history of ACS [52]. After continuous improvements [53], the first commercial implementations are also starting to appear [54].

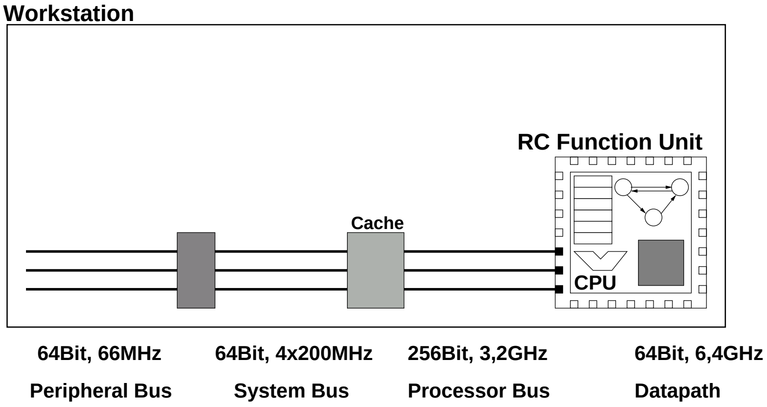


Figure 2.6: Reconfigurable Function Unit

In all cases, very low latency communication between the fixed-function units and the register file of the processor were achievable. However, the architectures and capabilities of the RFUs vary widely.

PRISC-1 [52], based on the MIPS R2000 single-issue processor architecture, added a 32b wide datapath consisting of a three-level network of 4-input LUTs. The RCU could accept two operands from the register file and produce a single output value. It was intended to run at the same clock frequency as the rest of the CPU (200 MHz). The RFU did not have access to the system memory and relied on software instructions for data delivery and retrieval. The small RFU capacity and low bandwidth resulted in only limited speed-up even over the very simple unaugmented CPU: Factors of just 1.06. . . 1.91 were achievable in parts of the SPEC92 benchmark suite.

The OneChip-98 ACS evaluated in [53], took a different approach. It coupled a 85K gate FPGA array similar to the Xilinx XC4000 architecture to a superscalar dynamically scheduled version of the DLX processor (resembling a MIPS R10000 architecture). In addition to significantly more logic capacity than the PRISC-1, this RCU also had direct access to memory via the processor pipeline's MEM stage. Dedicated logic ensured the coherency of memory viewed both from the CPU pipeline and the RFU [55]. Also, the processor pipeline and the RFU no longer executed in lockstep: The dynamic scheduling algorithm employed in the main pipeline [20] allowed its synchronization with instructions that required multiple RFU cycles. The resulting system achieved speedups of 14. . . to 2057[53]. That work also supports our experience that an ACS that allows the RCU direct access to memory does not profit significantly from allowing the fixed-function units (FU) to operate in parallel to the RCU. Thus, complex coherence and synchronization protocols can be avoided in favor of simply stopping (stalling) the fixed FU pipeline.

A recent commercial offering featuring an RFU is the Stretch S5000 processor series [54]. It integrates a reconfigurable Instruction Set Extension Fabric (ISEF) into a configurable Tensilica Xtensa V single-issue in-order processor [56]. Instead of just interfacing to the main datapath and communicating via the 32b register file, it operates entirely on 128b wide operands, using a dedicated wide register file with 32 entries. The fabric provides 4096 single-bit operators, 8192 multipliers and is able to accept three 128b input operands to compute up to two 128b results. The ISEF has direct access to memory, supported by dedicated data streaming hardware in fixed logic. The fixed parts execute at 300 MHz, while the reconfigurable logic of the ISEF uses a slower clock of just 100 MHz. Despite this discrepancy in clock speed, the RFU accelerates the execution of the EEMBC Telecomm Benchmark [57] over the unaugmented processor by a factor of 190. Even compared to the currently fastest DSP (TI C6416 at 720 MHz clock, running a hand-optimized assembler version of the benchmark), this is still 1.4 times faster.

2.5 Heterogeneous Memories

In addition to the different approaches of coupling CPU and RCU, a similar variety exists for integrating memory into an ACS. In a classical temporally distributed architecture, only a single memory shared by all computations exists. For a spatially distributed implementation, however, multiple memory banks accessible in parallel allow much higher performance. Practically, these numbers can reach from a handful of separate chip-external banks to hundreds or thousands of independent chip-internal memories.

This freedom has been exploited very early in ACS designs: The SPLASH-1 [58], one of the first ACSs, featured 32 independent 128Kx8b memories in its attached RCU. Today, the availability of higher density memory chips has reduced this number somewhat. However, ACS memory systems have become much more heterogeneous.

This can be illustrated by examining the architecture of the state-of-the-art WILDSTAR-II ACS [59]. While realized as a PCI/-X card, it acts as a complete ACS combining four on-chip PowerPC CPUs with RCUs acting in attached or co-processor modes. Both kinds of processors are supported by a plethora of independent memories:

- 12 banks of 36b wide, very fast quad-data rate static RAM each bank having a capacity 2. . . 8 MB.
- 2 banks of 32b wide double-data rate dynamic RAM, each providing up to 128MB of bulk data storage.
- 3 banks of 8b wide FLASH RAM to hold persistent data.
- 888 banks of 1b. . . 16b wide chip-internal static RAM, each having a capacity of 18Kb.
- Up to 22048 banks of 1b. . . 8b wide chip-internal static RAMs realized from logic blocks, each having a a capacity of 128b.

It should be obvious that, with this degree of spatial distribution in the memory system, the bottleneck induced by the von Neumann architecture's single memory no longer exists.

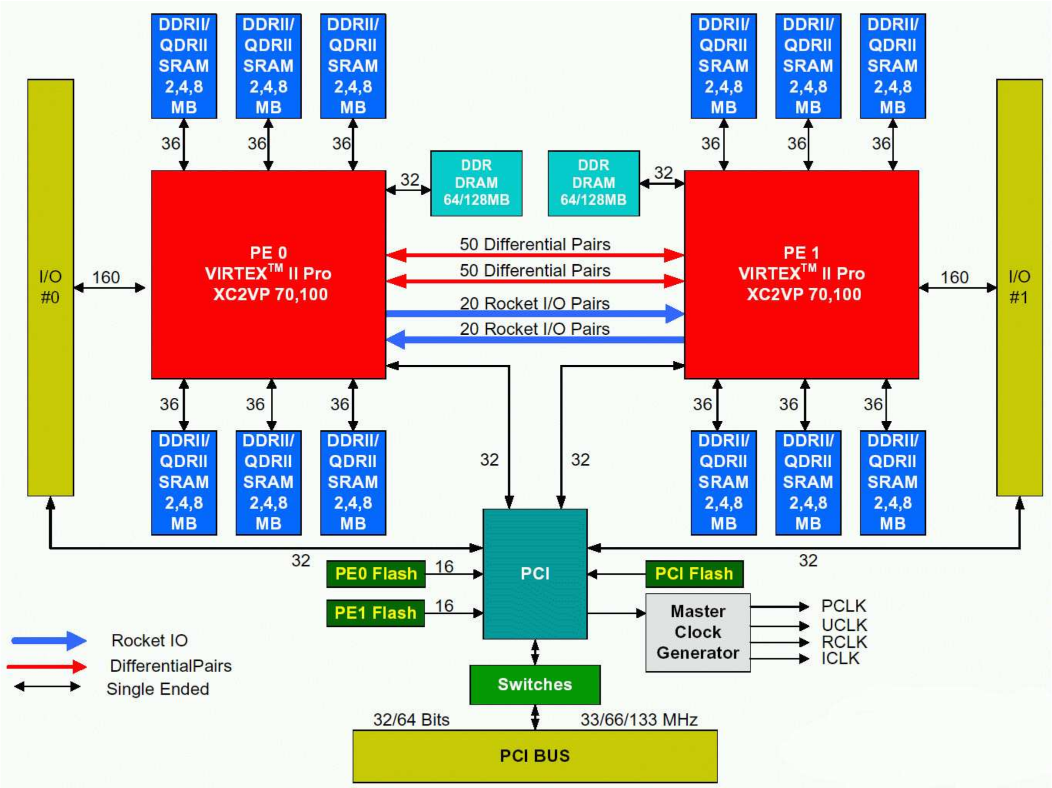


Figure 2.7: Architecture of WILDSTAR-II ACS [59]

2.6 Discussion

In general, a tighter integration between the CPU and the RCU in an ACS is preferable. The reduced latency and higher bandwidth allow a wider spectrum of applications to actually be accelerated by RCU execution. Consider, for a example, the stand-alone model, with its relatively high latency and low bandwidth. Here, only those parts of an algorithm that have comparatively long computation times (on the order of at least a few seconds) and operate only on small amounts of data before requiring interaction again with the CPU can profit from RCU execution. In other cases, the communications overhead will most likely exceed the RCU performance gain. On the other hand, in the more tightly coupled co-processor or RFU models, the acceleration of much smaller algorithms becomes profitable. The first RFUs [52] were used to accelerate straightline code sequences consisting of less than 10 instructions (RFU computation times of 5ns). More recent co-processor [46] and RFU [54] implementations support control flow (conditional execution, loops) as well as direct memory accesses and are thus able to operate independently of the CPU for longer periods of time. In this fashion, a larger fraction of the complete application can actually profit from RCU acceleration.

However, integration beyond the attached RCU becomes increasingly difficult. Even at board-level for realization of a peer RCU, the increasing clock frequencies require high-speed design and fabrication techniques. Tighter approaches require a chip-level integration and thus an ASIC design and fabrication. While the design complexity itself is somewhat reduced by the availability of IP blocks both for various RCU granularities [60] [61] [62] as well as for the CPU (many vendors, e.g., [63] [64] [65] [66]), a significant non-recurring engineering (NRE) charge covering system verification as well as the fabrication itself remains.

Furthermore, the extremely high operating frequencies used in many of todays desktop and server CPUs basically preclude the use of reconfigurable logic clocked at the same speed as the fixed FUs. A recent processor such as the 90 nm Pentium 4 "Prescott" CPU [22] has a clock speed of 3.4 GHz for the processor and *double-clocks* its two 64b ALUs to 6.8 GHz each. This speed is well outside the range achievable for even the fastest reconfigurable fabrics achievable today in standard chip process technology. Only highly experimental approaches relying on exotic SiGe materials and current-mode logic have the potential to realize a very simple reconfigurable architecture clockable in range the 5...20 GHz [67]. However, these approaches are hampered by their excessive power consumption (253 W for a 64x64 array of finely granular logic blocks).

While even in the area of desktop and server CPU design some rethinking is occurring (move towards higher efficiency over ever-increasing clock speeds, multi-core processors, greater emphasis on power management), the continued need for high-performance execution of the immense portfolio of legacy software will preclude fundamental changes in the design of these processors. When such systems require augmentation by an RCU, the attached realization will thus be the most common choice.

However, in the field of embedded computing, a completely different set of requirements offers a much better outlook for the practical use embedded ACSs. In embedded systems, the primary objective is *efficiency* in characteristics such as cost, performance, power consumption, flexibility, form factor, etc. Issues such as compatibility with legacy code and (to some extent) ease of use are only secondary. This is an environment in which ACSs can thrive, as both academic [68] and commercial [69] efforts have demonstrated the immense potential of reconfigurable computing. Instead of requiring a large, power hungry CPU, a small processor combined with one or more RCUs/RFUs can easily supply the required performance, at much lower costs in terms of area and power consumption [54]. In the ACS paradigm, complicated dynamic scheduling and synchronization mechanisms between the fixed and the reconfigurable units are not required [53]. Instead, a currently unused unit can simply be put in a sleep-mode, reducing power consumption even further. Since most embedded systems today are realized as *system-on-a-chip* (SOC), the tight integration optimal for ACS usability is achievable with relative ease. Furthermore, the clock speeds of a few hundred MHz common in the embedded area are well within the range of current reconfigurable fabrics.

2.7 Data Access Management

As mentioned in the preceding sections, one of the major differences both in realization and use of an ACS is the way the RCU or RFU obtains and delivers data from the rest of the system, specifically from the CPU and main system memory. For purposes of the following explanation, memory banks directly connected to or even realized inside of the

RCU will not be addressed here (but see Section 2.9 for some additional discussion).

In *slave mode*, all data transfers are controlled from outside of the reconfigurable unit. This applies to transactions initiated by the CPU such as reads/writes to RCU registers mapped into the CPU memory space (see Section 4.6), or using dedicated CPU instructions for communication [52] [46]. It also covers sourcing/sinking the RCU from an external DMA engine programmed by the CPU.

In *master mode*, all data transfers are controlled by the RCU. This includes direct accesses to memory shared with the CPU as well as programming external DMA engines and then processing the streamed data. Note that even an RCU capable of master-mode will generally be slave-capable in order to receive an initial set of parameters from the controlling software on the CPU.

Slave mode RCUs are limited by two factors: First, they can accept/deliver data only at the maximum rate the external master supports, which can be quite slow: For example, a software loop executing on the CPU to write/read data to/from a slave-mode RCU often leads to single bus transactions instead of longer data bursts, and might just use 10% of the available bus bandwidth. Second, when the algorithm realized on the RCU has to alter its access patterns due to data dependencies, it has to notify the external master (CPU or DMA engine) in some way to request the actual changing of the data transfers. Depending on the coupling between the units, this operation can be time-consuming (interrupt, software handler, shutting down existing transfers, reprogram engine, restart RCU processing, etc.).

In master mode, the RCU can adapt its access patterns directly to changed requirements. Furthermore, since the algorithm on the RCU is aware of its data transfer behavior, it can issue all transfers in the most efficient fashion, exploiting burst transfer modes whenever profitable. Section 4.6.3 will discuss a reconfigurable memory system supporting these capabilities. The significant advantages come at the price of increased RCU complexity: In addition to physical access to the main memory busses (requiring I/O resources such as pins/pads), the RCU must now also implement the memory access protocols. For the attached and peer RCU models, these protocols (e.g., PCI/-X, SMP bus protocols) are not trivial to realize, especially when cache coherency issues between CPU and RCU also have to be considered.

2.8 System Software

An adaptive computing *system* consists of more than hardware, of course. In general, software support for the following services is provided, either in the operating system via an appropriate device driver, a library linkable to user applications, or special CPU instructions assisted by dedicated hardware.

Initializing the RCU Common functions include the loading of the appropriate device driver and the initialization of the RCU hardware to a known state (clearing a possible leftover configuration, resetting outdated pending interrupt requests, etc.).

Configuring the RCU This service actually configures the RCU for a specific operation. Options here include a choice of the configuration source (from a file, from a constant array in the executable image) and the configuration format (compressed vs. uncompressed, full vs. partial). Since the full configuration for a large fine-grained device such as the Xilinx XC4VLX200 requires close to 6MB of data, reducing that size (even if only for storage when multiple configurations are used) becomes paramount. This is achievable by compressing the often sparse configuration bitstream [70] or just performing a partial configuration (if the device supports it) using a shorter bitstream. Some RCUs also allow the readback of a configuration to the CPU (generally for verification and debugging purposes).

Establishing Communication with the RCU As described in Section 2.7, there are multiple ways to communicate between RCU and CPU in an ACS. In some cases, special CPU instructions already perform that task [52] [46]. However, more frequently, the RCU is mapped into the memory or I/O spaces visible to the CPU (see Section 4.6). Then, appropriately addressed read/write instructions are routed to the RCU device. Conversely, for master-mode RCUs, all or part of the main system memory is made visible to the RCU. Such a mapping can be non-trivial on operating systems relying on virtual memory. Since most RCUs do not possess a memory-management unit (MMU) capable of performing the address translation between physical and virtual addresses, the mapped ranges must be contiguous and locked into physical memory (may not be paged out to disk). Some systems with slave-mode RCUs instead provide dedicated DMA channels for quickly sending/receiving streams of data to/from the RCU under control of a hardwired DMA engine (with the limitations described in Section 2.7). The handling of interrupts (e.g., registering a callback procedure for processing RCU-initiated interrupts) also falls into this area.

CPU-RCU Synchronization In some ACS architectures, the CPU and RCU may execute concurrently. The required support may either be realized in hardware [55], or, with a coarser granularity, by appropriate software routines. In the simplest case, an RCU hardware register is polled for a completion status. A more advanced approach that also supports multi-threaded execution on the hosts relies on one or more semaphores controlled by the RCU (e.g., by issuing appropriate interrupts to the CPU) that can control software threading on the CPU [70]. For ACS architectures that do not support cache coherency mechanisms in hardware, the appropriate functions must also be realized here in software.

System Management This catch-all heading covers operations such a programming the possibly variable clock frequency of the RCU (common for FPGA-based RCUs) and status inquiries, e.g., as to RCU temperature and power supply status.

2.9 ACS Architecture Development

After establishing the ACS fundamentals in the preceding sections, we can now follow the development of ACS architecture by examining two concrete ACS examples: The

SPARXIL, introduced in 1993 [71] [72], and the ACE-V, introduced in 2000 [70]. While both systems are physically realized as attached RCUs, their computing models differ drastically.

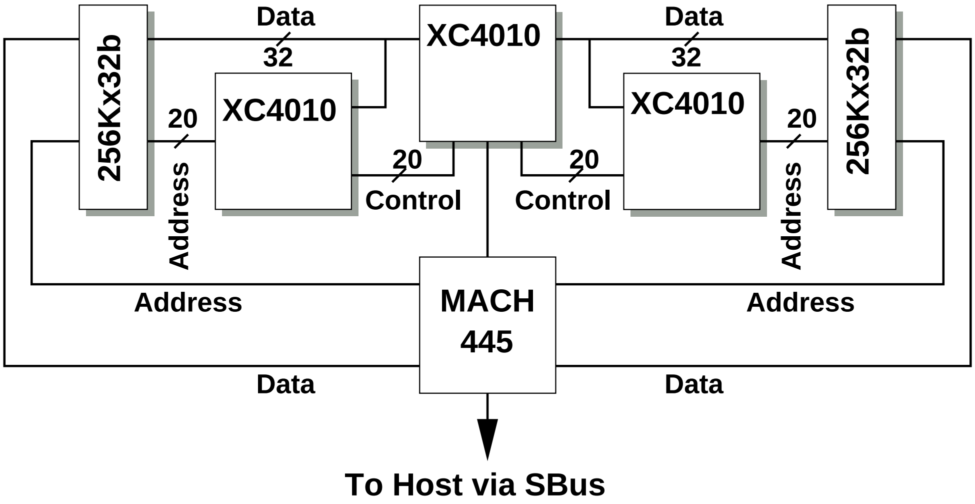


Figure 2.8: SPARXIL architecture

SPARXIL (Figure 2.8) was limited to the pure slave-mode operation typical of many early ACSs. As the SPLASH-2 [73], it attached to the Sun SBus, a 32b peripheral bus running at 20-25MHz and peaking at 50 MB/s effective bandwidth. The host CPU could read and write data to the two 256Kx32b static memories (fast for the time, 20ns cycle time). The RCU could then be signaled that the input data was set up correctly in its local memory and would take over control of the memory (literally disconnecting it from the bus using dedicated hardware) before beginning execution. At the completion of the calculation, the RCU would release its access to memory and indicate its readiness to the host by an interrupt. The host would then retrieve the results by reading the RCU memories. The RCU itself consisted of 3 Xilinx XC4010 FPGAs combined with a AMD MACH445 CPLD acting as bus interface unit (BIU) to the SBus and providing additional support functions (status and command registers, clock frequency programming, interrupt control).

In contrast, the ACE-V was designed to *emulate* the capabilities of an ACS following the co-processor model until suitable integrated chips would become available [46] [74] (which sadly never happened). It combines a microSPARC-IIep RISC processor with an RCU realized by a Xilinx XCV1000 FPGA. As in SPARXIL, the RCU is also supported by dedicated memories: Four 256Kx36b synchronous static SRAMs are directly connected to the FPGA. A PLX PCI9080 chip acts as BIU to the 32b@33MHz PCI bus to the CPU, while support functions similar to those used in SPARXIL are provided by a Xilinx XC95144XL CPLD (not shown in the block diagram).

However, here the similarities between the two systems end. In contrast to SPARXIL, which just provided RCU services to the host workstation, ACE-V is a complete ACS with its own CPU and operating system, a customized port of the RTEMS [75] real-time OS (RTOS). While the entire ACE-V is also implemented on a PCI card and housed in a conventional workstation, it relies on the host only for I/O (console and disk access),

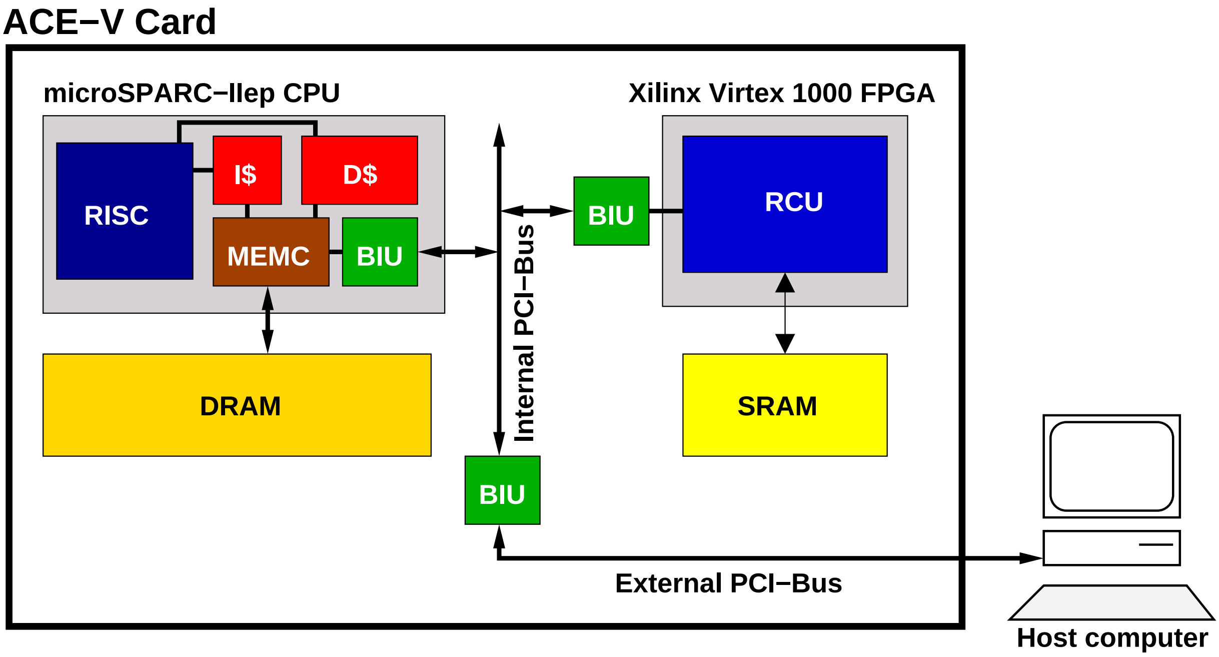


Figure 2.9: ACE-V architecture

but not for computation. Furthermore, the RCU fully supports master-mode access to the CPU-connected main memory via the PCI bus. Since the microSPARC-IIep is not intended for multi-processor applications, cache coherency mechanisms are implemented in system software. In this fashion, both software and hardware allow the RCU to act in a co-processor-like manner. The real-world performance of the system is of course limited by the deficiencies of the PCI bus (high latency and low bandwidth compared to a real co-processor model).

Beyond these fundamentals, the development of ACS realizations over almost a decade is also reflected in the design choices at the implementation level. The largest FPGAs available for SPARXIL could not simultaneously support the required 32b data and 20b address busses in a regular fashion, namely running horizontally across the FPGA matrix. Thus, the address and data busses were attached to *separate* FPGAs. Figure 2.10 shows the details of the left half of this arrangement, the right half is symmetrical.

The central FPGA (UFPGA, called the *user* or *data* FPGA) housed the main datapaths and was connected to the data ports of both the left- and right-hand memory banks by ULDATA and URDATA. The address port of each memory bank was connected via L_ADR and R_ADR to LSFPGA and RSFPGA, so-called *service* or *address* FPGAs. These devices were intended to act as dedicated address generation units (AGU). The service FPGAs could be controlled by the user FPGA using 16b control busses LCTRL and RCTRL. Furthermore, they could also receive data by listening on the data bus connected to their corresponding memory bank.

While this arrangement has proven suitable for generating fixed address sequences triggered by the user FPGA, the issue of data-dependent address patterns is more difficult. In the worst case, the address value has to be transferred over the data bus from the user

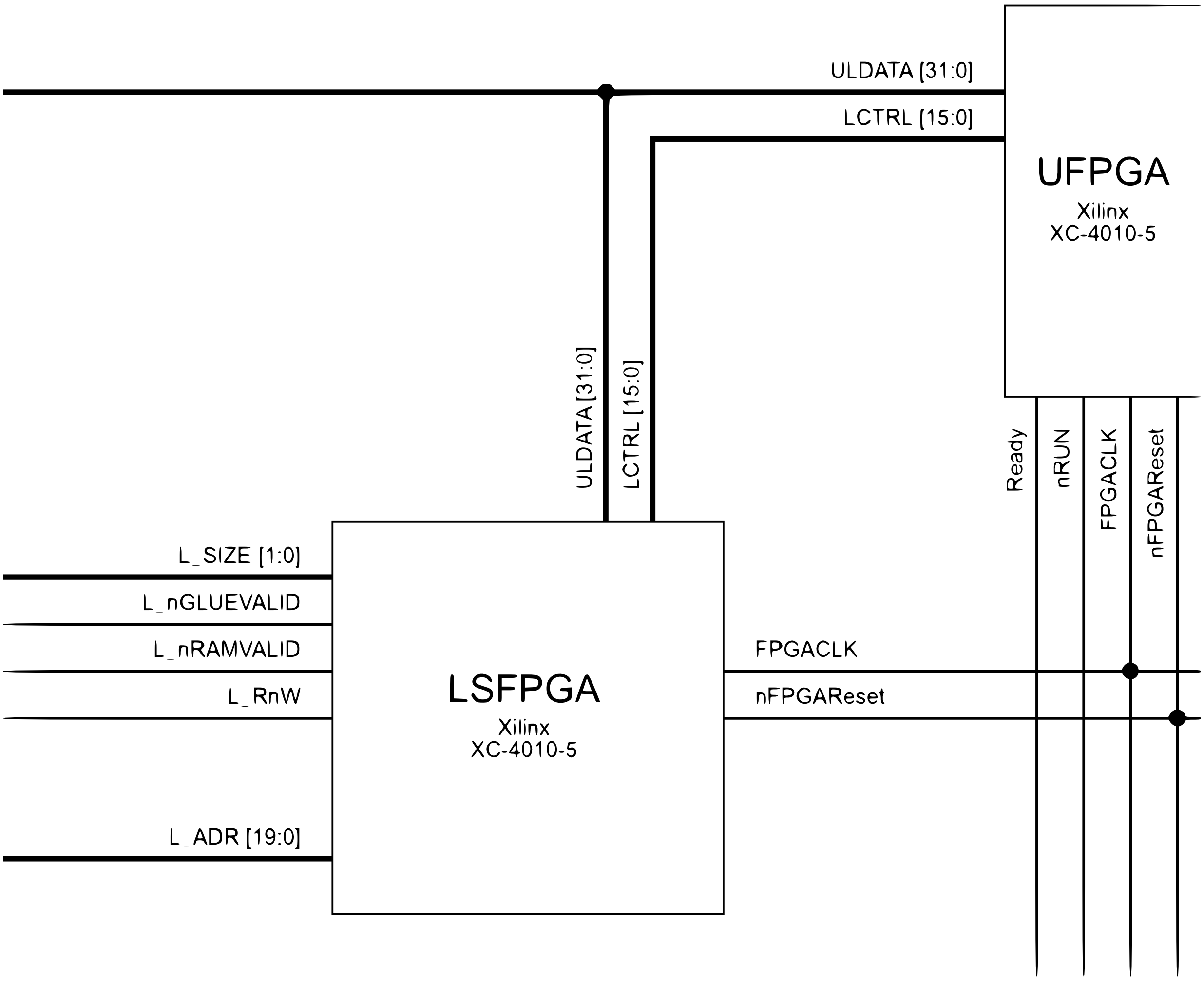


Figure 2.10: Dedicated user and service FPGAs in SPARXIL

FPGA to the service FPGA, which then forwards it to the RAM address lines. This significantly reduces the memory bandwidth to (at best) one half of the original capability. In limited cases (such as the image labeling application of Section 3.2), this overhead can be avoided when a narrow data width allows the dedication of the unused upper data bus lines to inter-FPGA communication.

The separation between data and address processing is only one area where the limited reconfigurable capacities imposed certain design choices. Another attempt to reduce the demand on the reconfigurable area is the realization of support functions in hardwired logic. Examples include the implementation of a byte-steering network (responsible, e.g., for always making byte-aligned data available on bits 7 to 0 of the data bus) and multiplexing access to the two RAM banks between the SBus and the FPGAs. Both functions are provided by discretely realized networks of tri-state buffers, controlled by the CPLD.

On the ACE-V, the FPGA has sufficient capacity to directly implement all of these operations on-chip. While requiring more configurable area (see also Section 4.6.3), this approach allows a greater flexibility in matching the RCU to the needs of the application. Data-dependent address sequences are efficiently realizable, for example, by directly connecting the datapath to the memory address lines.

SPARXIL featured a more powerful configuration management scheme, though: A dedicated bank of slower RAM held four complete system configurations (all three FPGAs plus clock programming data) on-board. Reconfigurations could be initiated both by the host or by hardware on the user FPGA itself (self-reconfiguration). The new bitstreams could then be loaded without taxing the SBus for a long period of time (at a slow maximum configuration speed of 1 MB/s). This was intended to allow the RCU a greater independence from the host CPU when performing a computation too complex to spatially implement in the available reconfigurable space. However, due to the slow reconfiguration speeds of the FPGA chips and the other limitations of the SPARXIL platform discussed above, the feature was never employed in a real application.

In summary, implementing very general functions, such as byte steering and memory access multiplexing, in hardwired logic on SPARXIL was successful in preserving reconfigurable area without limiting the flexibility of the system. However, the decision to split data and address calculations in the realized manner allowed an ideal spatial distribution of only a limited set of applications (fixed sequence AGUs). More complex computations with data-dependent behaviors were adversely affected by the implementation. The lessons learned here are still relevant to the construction of SoCs with reconfigurable components, where careful trade-offs between the reconfigurable and hardwired implementation have to be made for each feature.

3 Applications

Over the years, ACSs have been used to implement and accelerate applications from a wide spectrum fields. One of the first major successes was the comparison of gene sequences. In 1993, the SPLASH-2 ACS beat the MasPar MP-1, one the fastest commercially available computers then, with a speed-up factor of 1300 [76]. Recent revisits to this domain [77] have increased that performance by yet another factor of 1200. Cryptography has also been fertile field for ACS usage. In 1993, an ACS-based RSA decryptor [78] had 10x the performance of the fastest conventional processor. These successes continue today, with the fastest manually optimized assembly software implementations of the Advanced Encryption Standard (AES) on a 3.2 GHz Pentium 4 processor achieving roughly 1.5 Gb/s [79], while FPGA-based solutions reach 17.8 Gb/s at a clock frequency of just 140 MHz [80]. The latter result is of great practical interest with the increasing need for secure data communications at multi-gigabit speeds. In addition to performance gains, ACS-based solutions can also excel in other areas. For an LMS correlator (a common DSP application in modern wireless communications), a single ACS-on-a-chip successfully replaced a realization consisting of 36 TMS320C54 DSPs [81]. [68] describes a voice compressor implementing the VCELP algorithm on an ACS. With a power consumption of only 1.78 mJ/s, it uses only a fourth of the power required even by a 1V low-power DSP.

In our own studies of ACS applications, we have also examined a mix from widely disparate areas. Three of the major ones considered will be briefly described here.

3.1 Wavelet Image Compression

3.1.1 Application

Wavelet image compression is a lossy operation that aims to reduce the storage size of images while preserving their visual quality [82]. It relies on multiple hierarchical 1-D filtering passes, shown in Figure 3.1, alternately performed in the horizontal and vertical directions, to separate the low-frequency components (large, obvious parts) from the high-frequency parts (minor details).

Figure 3.2 shows the complete process: The resulting wavelet coefficients are then quantized to data words with a narrower width, deemphasizing the higher frequency components. The computed data stream is then further compressed by first performing a variant of run-length encoding (replacing longer sequences of zeroes by a single counter value) and a final step using static-table Huffmann coding to replace common data values with shorter (in terms of bit width) codes.

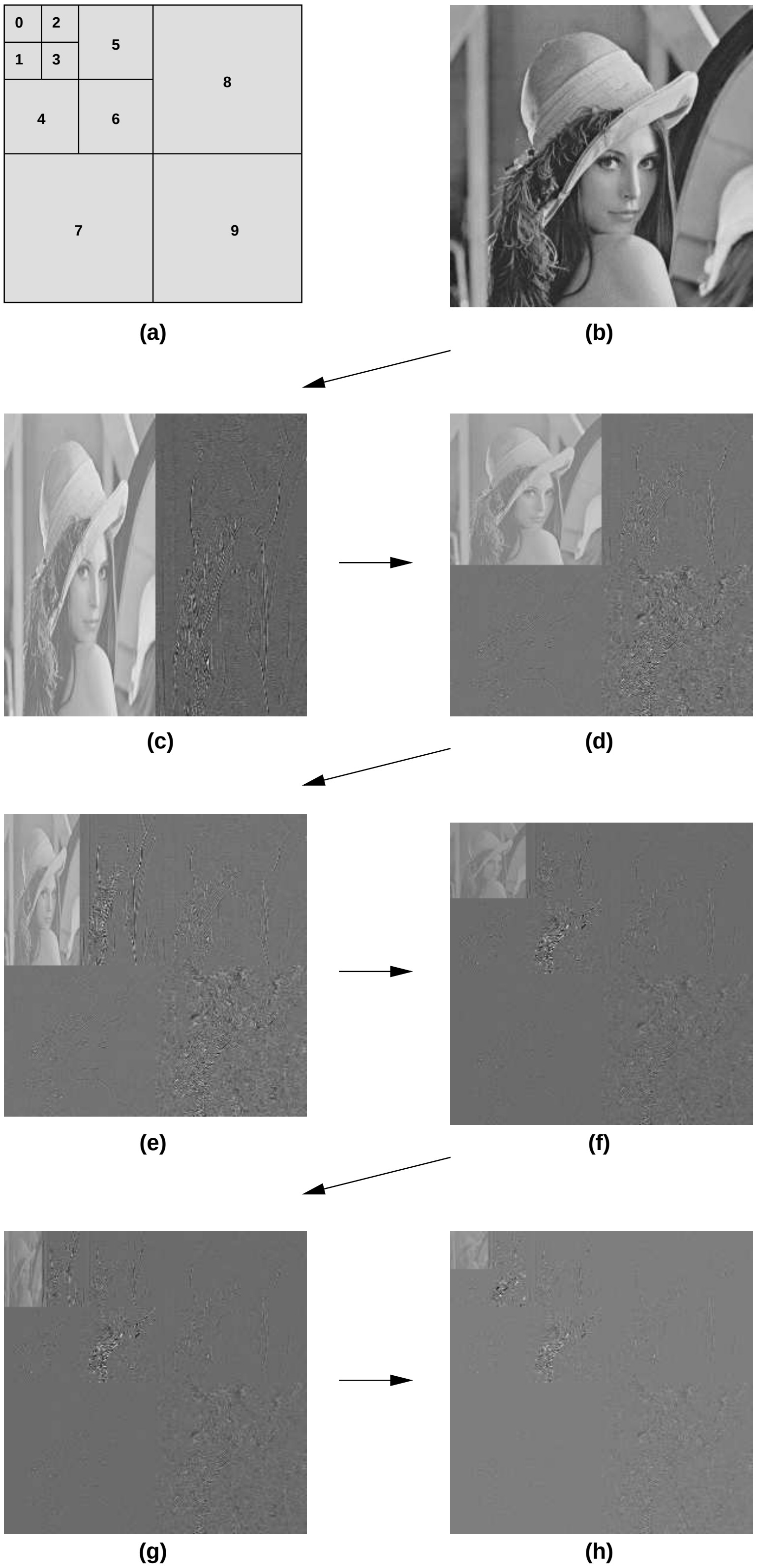


Figure 3.1: Wavelet transform applied to image

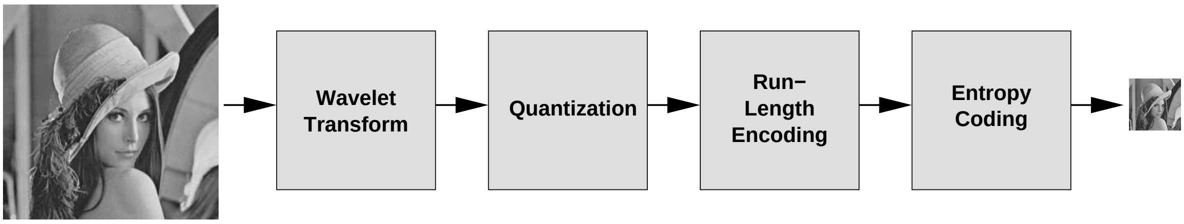


Figure 3.2: Wavelet image compression steps

The application fundamentally has a data-flow structure (with variable data rates after the run-length encoder, though), but little conditional execution (apart from the flow-control between different processing stages). It is thus representative of many digital signal processing algorithms in use today.

3.1.2 Realization

Our implementation [83] on the ACE-V ACS [70] realizes the entire processing pipeline on the RCU, only setup (configuration, memory management) and disk I/O is executed on the CPU. Due to data dependencies, the execution has to proceed in two stages: The Wavelet processing and the Quantization / Run-length / Huffman (QZH) phases execute sequentially. Each phase, however, is heavily pipelined internally. The memory ranges for holding intermediate and final data were carefully distributed across the available external RAM banks to both parallelize (three simultaneous data streams) and to allow contiguous accesses (to avoid restarting streams). In addition, constant data, such as the static Huffman table, is stored in chip-internal memories that are also accessed in parallel with the external RAMs. All data streams are implemented using MARC (Section 4.6.3). The RCU heavily exploits subword parallelism, fetching and processing up to four 8b pixels in parallel. The degree of sub-word parallelism is limited by the 32b width of the external busses and could easily be increased on a different target ACS.

Dynamic reconfiguration could have been used to implement the sequentially executing Wavelet and QZH pipelines in separate configurations. This would have lead to area savings of circa 25%. However, this low reduction (due to the need to realize MARC in both configurations) combined with the very long reconfiguration time of the XCV1000 FPGA, made its use unattractive in practice.

3.1.3 Results

Figure 3.3 compares the performance of different implementations of the algorithm. Despite the limitations of the ACE-V (1998's vintage FPGA as RCU, slow RCU-memory access via PCI), the application could be realized with a 33 MHz RCU clock frequency with performance matching or exceeding that of conventional CPUs running at clock frequencies up to 50x higher (especially for larger images of 512x512 pixels). The 1.1W power consumption of the RCU was only a fraction of that required by some CPUs, which could

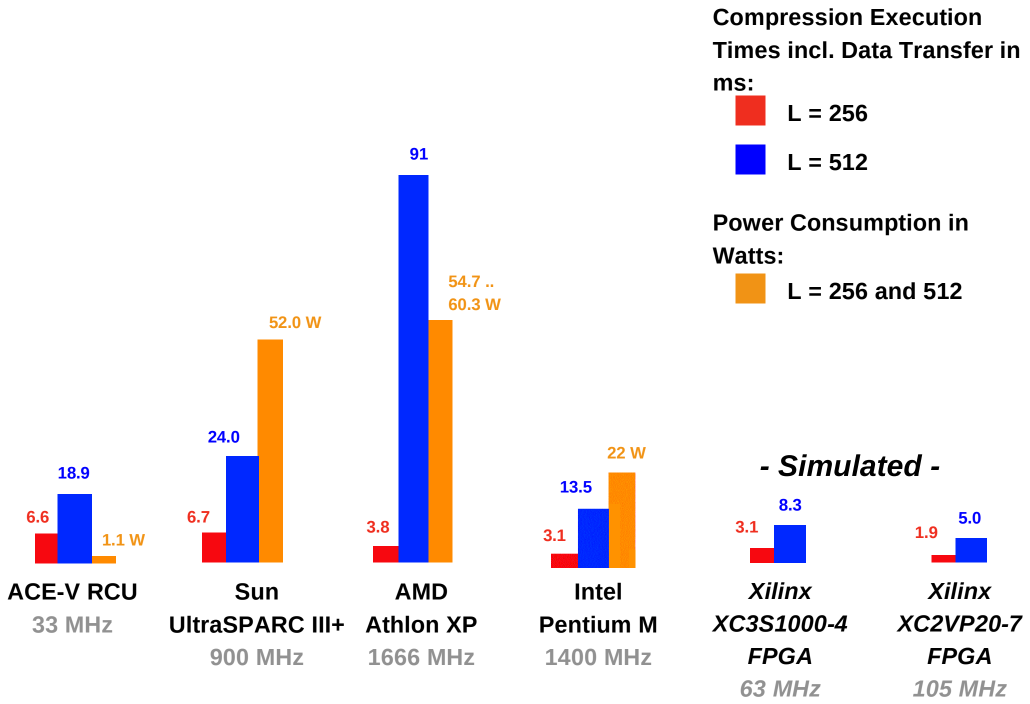


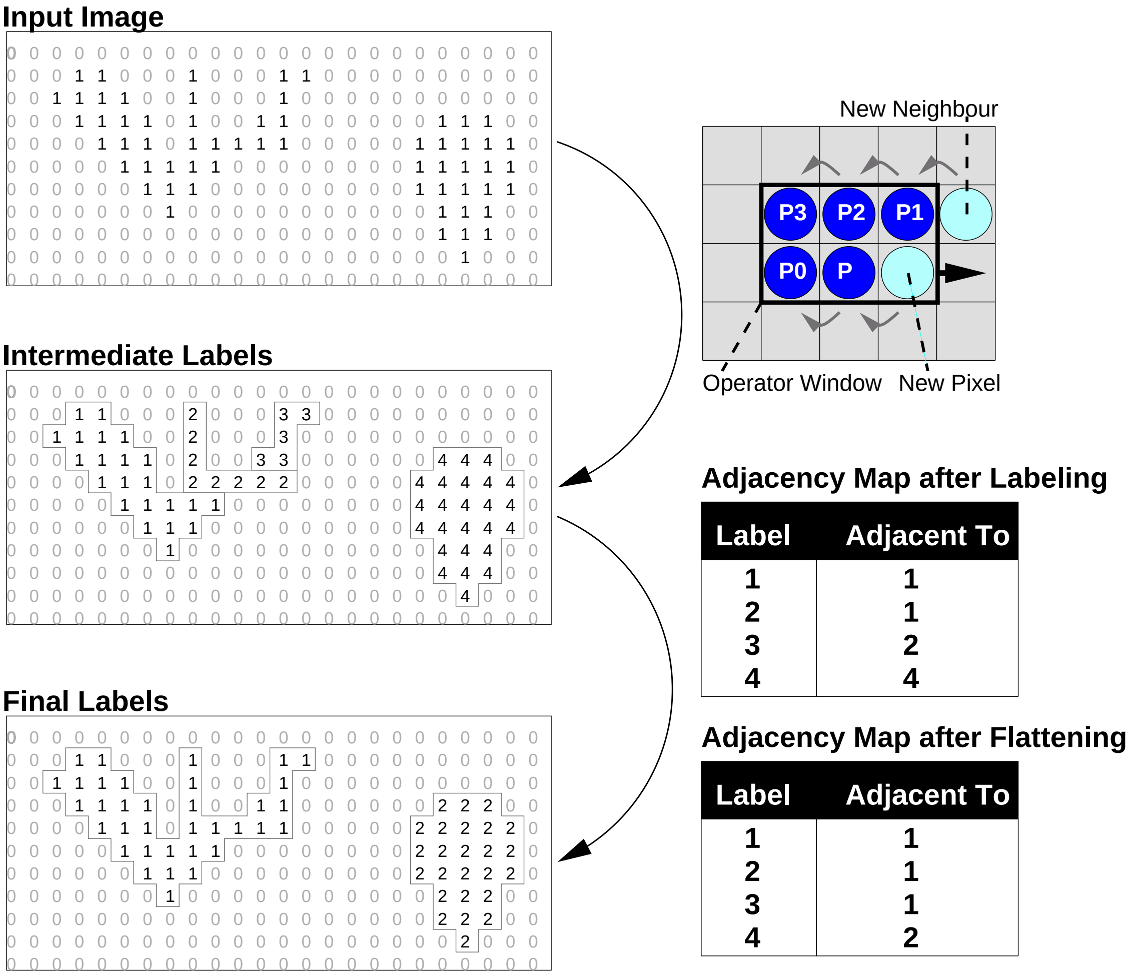
Figure 3.3: Wavelet image compression performance comparison

easily reach more than 60W. For comparison, we also simulated the performance of the implementation on faster RCUs in the same system architecture. To this end, the design was re-targeted both to a commodity device (Xilinx XC3S1000, with a cost of less than USD 12 in volume) and a high-end chip (Xilinx XC2VP20, cost of several hundred USD). For larger images, even the low-cost device easily beats the performance of a modern Pentium M CPU. For the high-end FPGA, even greater speed-ups would be possible when using the on-chip CPU and a 64b path to the main memory.

3.2 Image Segmentation

3.2.1 Application

This application accepts as input a black-and-white image consisting of set and unset bits. As output, it produces an array with dimensions identical to those of the input image, but now holding 16b label values. All pixels belonging to the same object (defined as a contiguous region of set pixels) have the same unique label. The computation proceeds in three phases (Figure 3.4). First, a sliding operator window is horizontally scanned across the rows of the input image to determine adjacency relationships between pixels and stores these in an adjacency map. Second, the map is processed to merge formerly undiscovered transitive adjacencies. In a last phase, the optimized map is then used to actually generate the final label array from the input image.



Input Image

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0
0 0 1 1 1 1 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 0 1 0 0 1 1 0 0 0 0 0 0 1 1 1 0 0
0 0 0 0 1 1 1 0 1 1 1 1 1 0 0 0 0 0 1 1 1 1 1 0
0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 0
0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Intermediate Labels

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 0 0 2 0 0 0 3 3 0 0 0 0 0 0 0 0 0 0
0 0 1 1 1 1 0 0 2 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 0 2 0 0 3 3 0 0 0 0 0 0 4 4 4 0 0
0 0 0 0 1 1 1 0 2 2 2 2 2 0 0 0 0 0 4 4 4 4 4 0
0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 4 4 4 4 4 0
0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 4 4 4 4 4 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 4 4 4 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 4 4 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Final Labels

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0
0 0 1 1 1 1 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 0 1 0 0 1 1 0 0 0 0 0 0 2 2 2 0 0
0 0 0 0 1 1 1 0 1 1 1 1 1 0 0 0 0 0 2 2 2 2 2 0
0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 2 2 2 2 2 0
0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 2 2 2 2 2 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 2 2 2 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Adjacency Map after Labeling

Label	Adjacent To
1	1
2	1
3	2
4	4

Adjacency Map after Flattening

Label	Adjacent To
1	1
2	1
3	1
4	2

Figure 3.4: Image segmentation algorithm

The application combines dataflow processing (the sliding image window and final labeling phases) and a more control-intense part (the analysis and processing of the adjacency map).

3.2.2 Realization

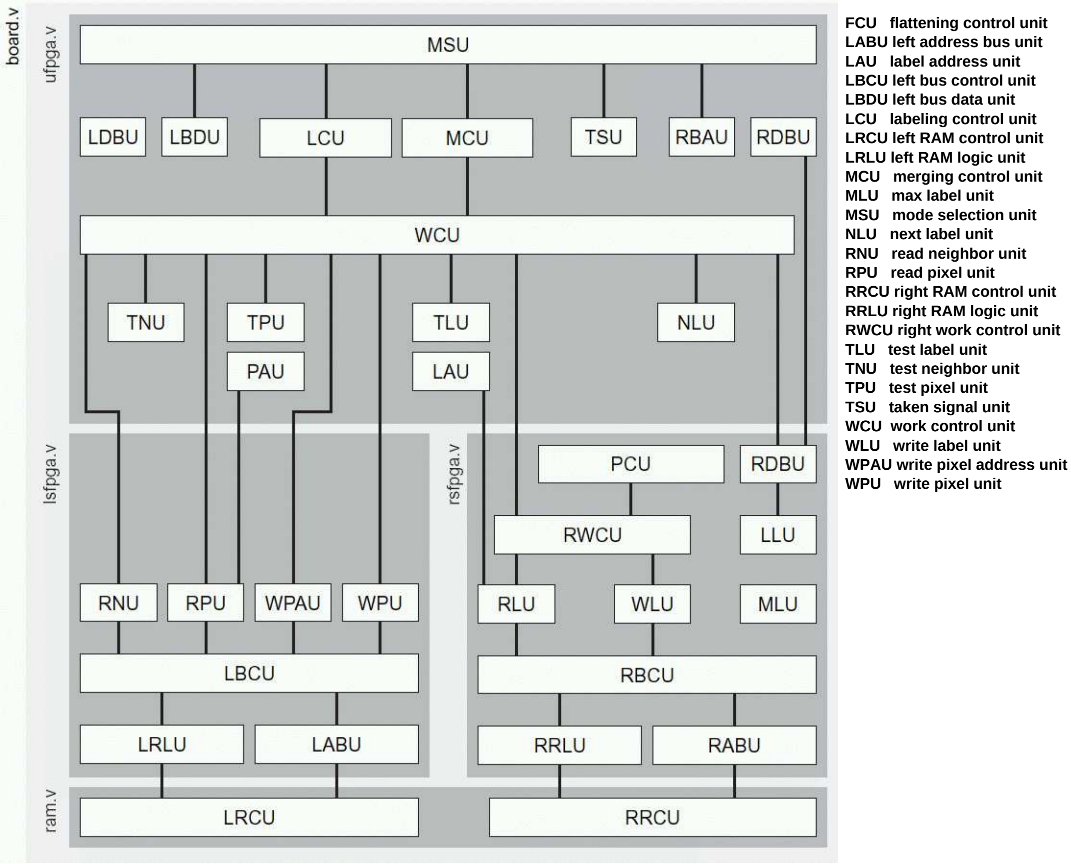


Figure 3.5: RCU Architecture on SPARXIL

This application is particularly interesting, since it was realized both on the SPARXIL [72] as well as on the ACE-V ACSs [84]. On the latter platform, the entire design could be realized on the single-chip RCU instead of being split across the separate data and address FPGAs on SPARXIL (Section 2.9). Furthermore, the increased number of resources (logic capacities and memory banks) allowed a wider spatial distribution on the ACE-V. The different hardware structures are shown in Figures 3.5 and 3.6.

The increased number of memories on the ACE-V also simplified the design, since the locking and arbitration logic necessary to coordinate memory access on SPARXIL is no longer necessary. Instead, on the ACE-V, each data structure is housed in its own dedicated memory (Figure 3.7).

For better performance comparison, both realizations operate strictly as attached RCUs in slave-mode, the master-mode capabilities of the ACE-V were not used here. The areas

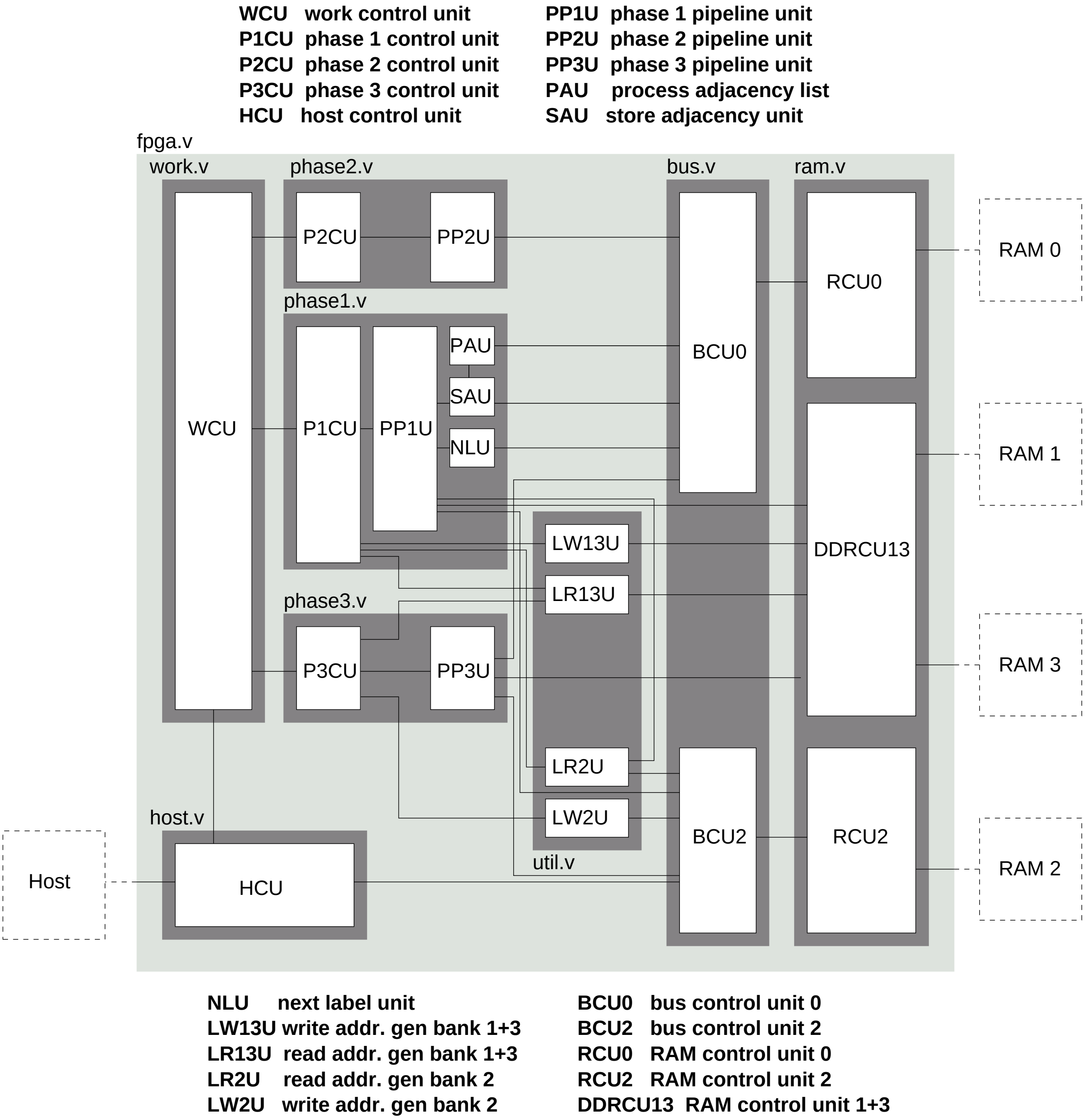


Figure 3.6: RCU Architecture on ACE-V

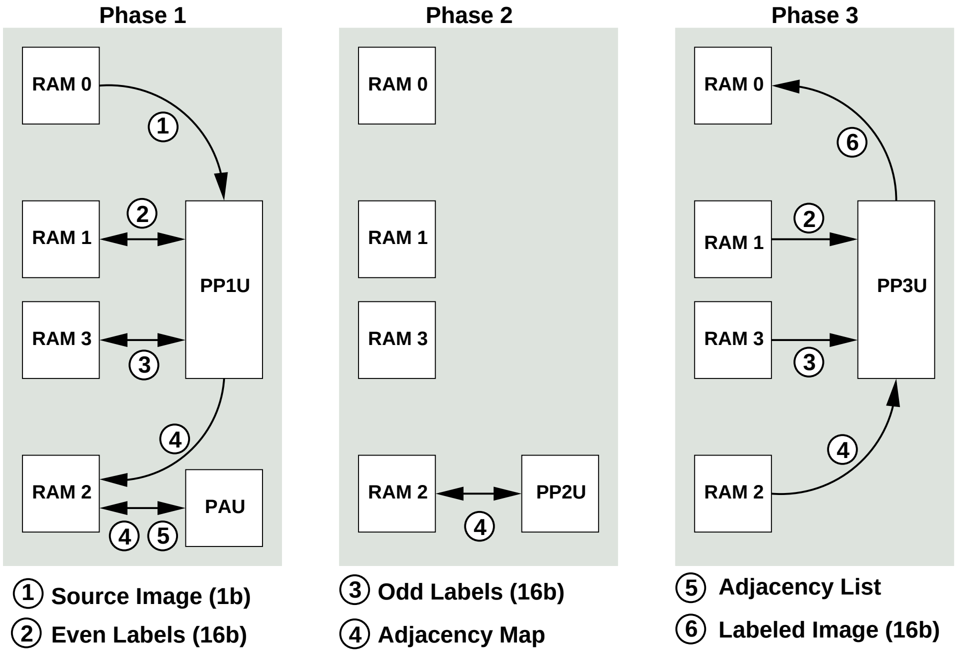


Figure 3.7: Memory organization and dataflow on ACE-V

required for the two realizations, are not directly comparable, however, since the ACE-V has to realize the CPU-to-memory interface (four 52b wide datapaths) on the RCU. On SPARXIL, this functionality was implemented externally in the MACH445-based BIU.

Since all three phases of the algorithm execute sequentially, dynamic reconfiguration could have been used to reduce the required reconfigurable capacity. Due to the pure slave-mode operation of the RCU, no RCU-local memory controller (as in Section 3.1) was required. Thus, the possible area gains would have been significantly better than for the Wavelet example. However, again, the glacial FPGA configuration speeds precluded the actual use of this technique.

3.2.3 Results

In 1997, the SPARXIL implementation running at 25.2 MHz was able to outperform very fast RISC processors running at 296 MHz by a speedup factor of 4.7. The ACE-V-based realization shown in 2003 ran at just 36 MHz, but with its architectural improvements exceeded the performance of a then-current RISC CPU clocked at 900 MHz.

The impact of these architectural changes is shown in Figures 3.8 and 3.9 which compare the average number of clocks required for processing a single pixel in varying image densities (blank to completely filled). In addition to the general reduction in cycle counts, it is also apparent that the per-pixel processing time of the improved implementation on the ACE-V is less data-dependent than the SPARXIL version.

Despite the clock speed-up of just 1.4x, the improved architecture of the ACE-V implementation allowed a performance gain of 2.2x over SPARXIL. Furthermore, it turns out

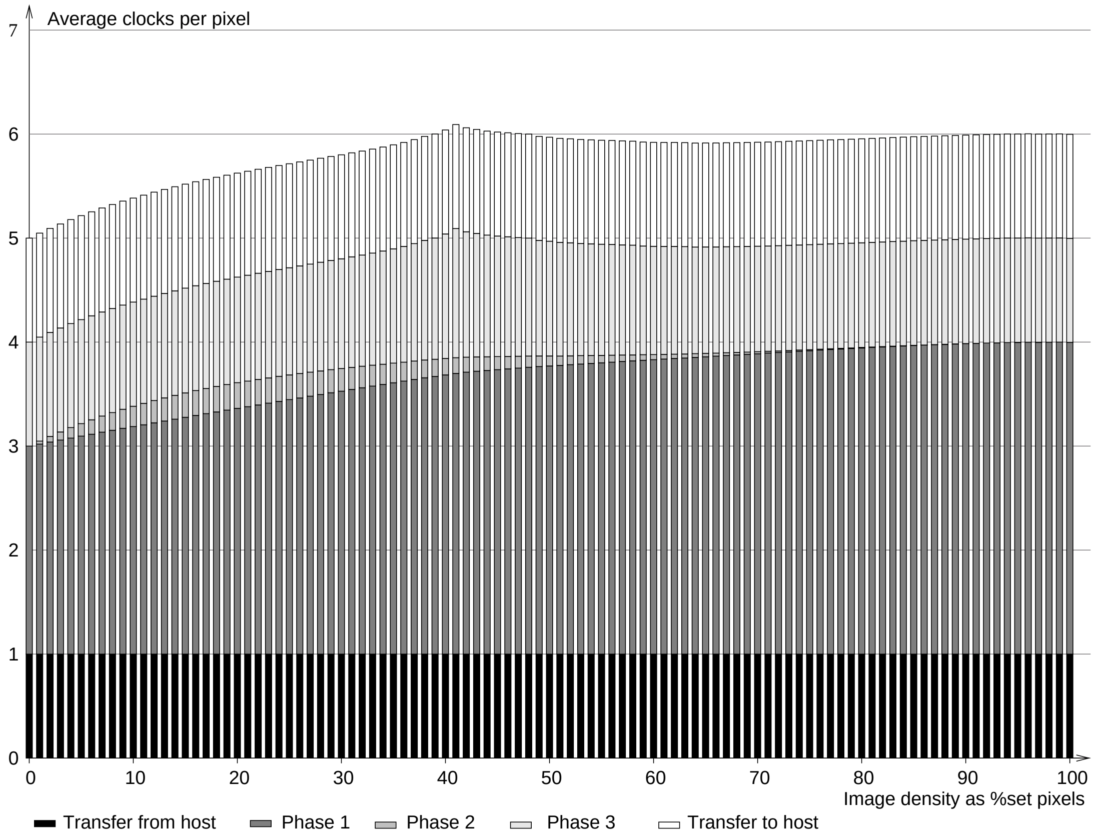


Figure 3.8: Average clocks per pixel vs. image density on SPARXIL

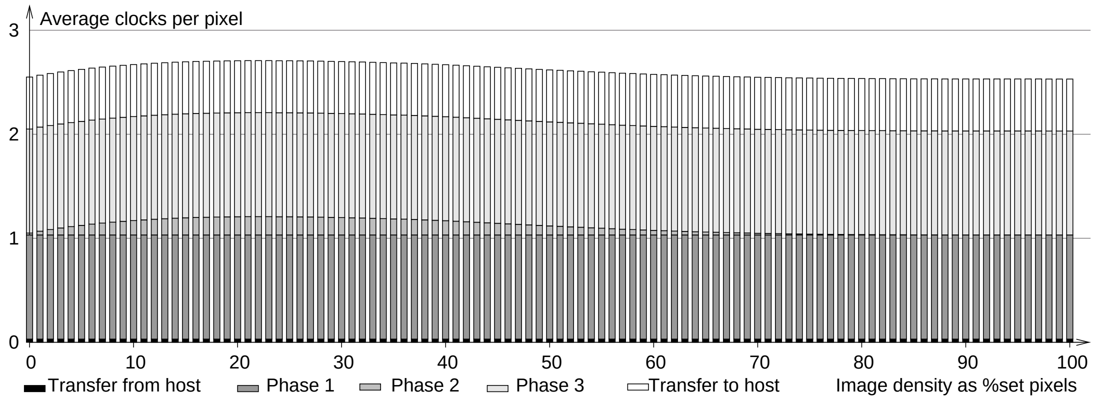


Figure 3.9: Average clocks per pixel vs. image density on ACE-V

that the maximum clock speed of the ACE-V is not limited by the RCU, but by the clock limit of the external PCI9080 BIU. The internal pixel-processing pipelines can run considerably faster: If the same spatial computation would be mapped to an RCU realized using a more recent FPGA (Xilinx Virtex II, speedgrade -6), the total execution time would drop from the current value of 15ms to just 6.5ms, thus outperforming even state-of-the-art processors with multi-gigahertz clock speeds.

3.3 Artificial Life Simulation

3.3.1 Application

Computer simulation of “digital organisms” can be employed to experimentally study natural evolutionary processes that would take millions of years in biological systems. TIERRA [85] is one of several artificial life simulation systems that have been developed in the past. It provides a software model of a virtual multiple-instruction, multiple data (MIMD) shared memory computer and operating system. The organisms are modeled as programs (using the instructions in Table 3.1) that are executed in parallel on this processor. In contrast to conventional CPUs, TIERRA is designed to allow the evolution of the organisms by mutation (changing the program code on the fly) and recombination (exchanging code segments between programs). Organisms that still remain functional (contain valid instructions) after such modifications continue to live (run), while those that are damaged (contain invalid instructions) die (are marked for removal from execution and memory).

In contrast to the previous two examples, TIERRA is a heavily control-dominated application. While the hardware structure and instruction set of the processor appear rather simplistic at first glance (four 16b registers, 16 entry stack, 32 operations), the execution of programs proceeds significantly different from conventional processors:

- Mutation randomly alters instruction semantics such as source/target registers, introduces +/- 1 inaccuracies in arithmetic instructions, and can cause errors in memory-to-memory copies.
- TIERRA does not use jump addresses but marks relevant locations in memory using patterns of special NOP instructions. A jump instruction thus has to perform a bidirectional search through memory for the NOP marker sequence.
- Two of the “instructions” are actually complex services such as memory allocation with garbage collection and spawning a new parallel thread of control (both used for cell division).

Additionally the TIERRA processor also tracks statistics such as the number of illegal operation attempted by each thread of control (organism). Too many of these lead to removal of the thread (death of the organism).

Code	Description
nop0	no-op, code 0x00
nop1	no-op, code 0x01
pushA	push AX on stack
pushB	push BX on stack
pushC	push CX on stack
pushD	push DX on stack
popA	pop AX from stack
popB	pop BX from stack
popC	pop CX from stack
popD	pop DX from stack
movDC	DX ← CX
movBA	BX ← AX
movii	[AX] ← [BX]
subCAB	CX ← AX − BX
subAAC	AX ← AX − CX
incA	AX ← AX +1
incB	BX ← BX +1
incC	BX ← BX +1
decC	CX ← CX −1
not0	CX ← CX *xor* 1
zero	CX ← 0
shl	CX ← 2·CX
ifz	skip next instr if CX $\neq$ 0
jmpo	jump to nearest label
jmpb	jump backwards to label
call	call to nearest label
ret	return from call
adro	AX ← find nearest label
adrf	AX ← forward find label
adrb	AX ← backward find label
mal	allocate memory
div	create new thread (cell division)

Table 3.1: TIERRA Instruction Set 0

3.3.2 Realization

When the project was started in 1996, we intended to fully apply the spatial computation paradigm to the hardware implementation of the normally software-simulated TIERRA processor model. As in RISC processors, a pipeline of execution stages would implement the instruction set and the TIERRA peculiarities described above. Dedicated units would realize the fast label searching (supported by caches of pre-determined label locations) and the threading mechanisms.

However, it quickly became apparent that the space requirements of such a solution were far beyond the capabilities not only of the SPARXIL architecture, but also the largest FPGA-based RCUs available at the time. Thus, we altered our implementation strategy to arrive at a design that could be realized on a hypothetical platform superficially similar to SPARXIL (slave-mode only, two memory banks), but replacing the triple-FPGA structure with a single higher density device (Xilinx XC4062XL or XC4085XL) [86].

Even with the increased reconfigurable capacity (double to triple that of the original SPARXIL), the application could be realized only by a deeply microcoded architecture. Each TIERRA instruction is processed by three levels of microcode.

1. The 32 instructions are mapped to 25 micro-routines, each consisting of multiple micro-instructions. A NOP, for example, is processed in just two micro-instructions, while a memory allocation instruction requires 25 micro-instructions (also containing conditionals and loops).
2. The 25 micro-routines are composed from 129 nano-code routines, each consisting of one to six nano-instructions.
3. Each nano-instruction is realized as 16b control word that contains both constant bits as well as variable bits determined by literal values etc. Each word requires three clock cycles for execution (two memory accesses plus a cycle of bus turnaround).

Even in this micro-coded implementation (shown in Figure 3.10), an area-efficient design style following the paradigm of temporal distribution had to be used. For example, each of the nano-routines is re-used an average of 2.6 times for implementing micro-instructions. The upper bound for re-use reached as high as 21 times.

However, our design was able to exploit parallelism in its memory layout (Figure 3.11). By separating the state of the organisms (in the left RAM of SPARXIL) and the administrative data (in the right RAM), administrative tasks running in the background could occur simultaneously with the actual execution of an organism (called a *cell*).

Since there are no separate processing phases or modes, our TIERRA design would not profit from classical dynamic reconfiguration. However, in a fully spatially distributed design, selective partial reconfiguration could be employed to implement the mutation effects by actually altering the corresponding hardware operators. For performance reasons, the static reconfiguration controller orchestrating these modifications would also be realized on-chip, leading to *self-reconfiguration* using on-chip access to the configuration network. These capabilities are present in modern FPGAs (Xilinx Virtex II Pro and Virtex 4 series) and could be exploited for a completely different approach to the application.

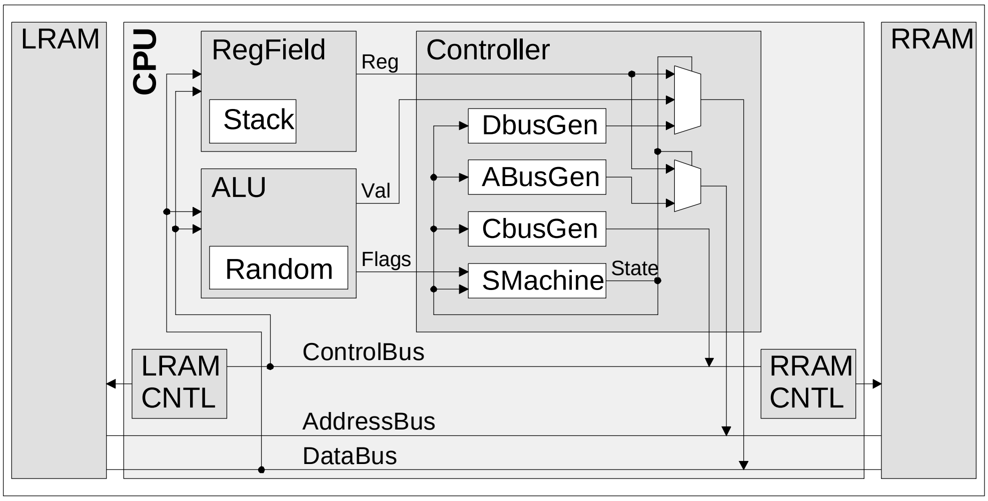
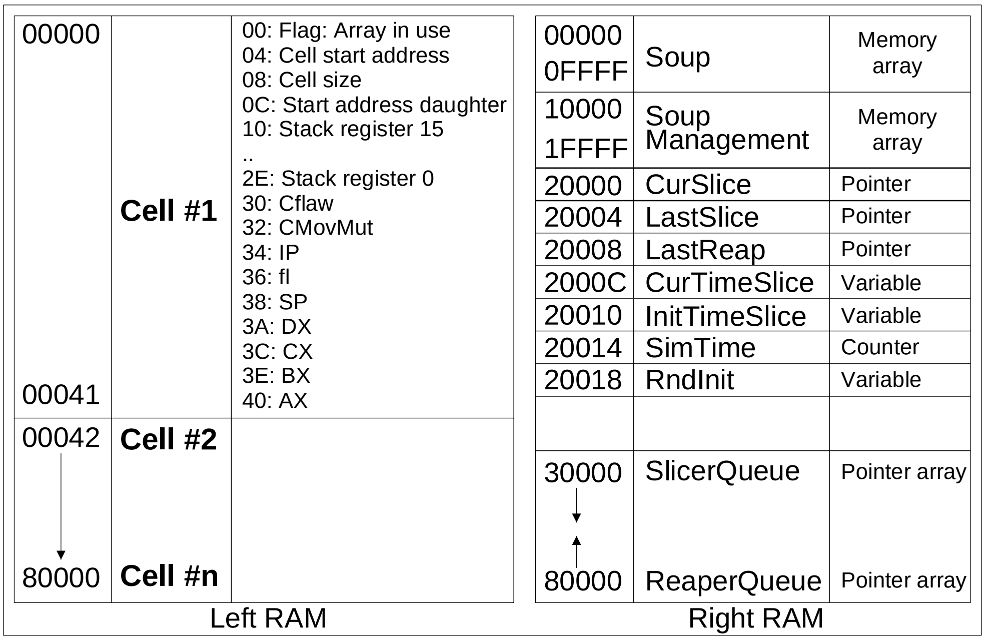


Figure 3.10: Architecture of implemented TIERRA processor



00000 00041	**Cell #1**	00: Flag: Array in use 04: Cell start address 08: Cell size 0C: Start address daughter 10: Stack register 15 .. 2E: Stack register 0 30: Cflaw 32: CMovMut 34: IP 36: fl 38: SP 3A: DX 3C: CX 3E: BX 40: AX
00042 ↓ 80000	**Cell #2** **Cell #n**	

Left RAM

00000 0FFFF	Soup	Memory array
10000 1FFFF	Soup Management	Memory array
20000	CurSlice	Pointer
20004	LastSlice	Pointer
20008	LastReap	Pointer
2000C	CurTimeSlice	Variable
20010	InitTimeSlice	Variable
20014	SimTime	Counter
20018	RndInit	Variable
30000 ↓ ↑ 80000	SlicerQueue ReaperQueue	Pointer array Pointer array

Right RAM

Figure 3.11: Memory organization of the TIERRA processor

3.3.3 Results

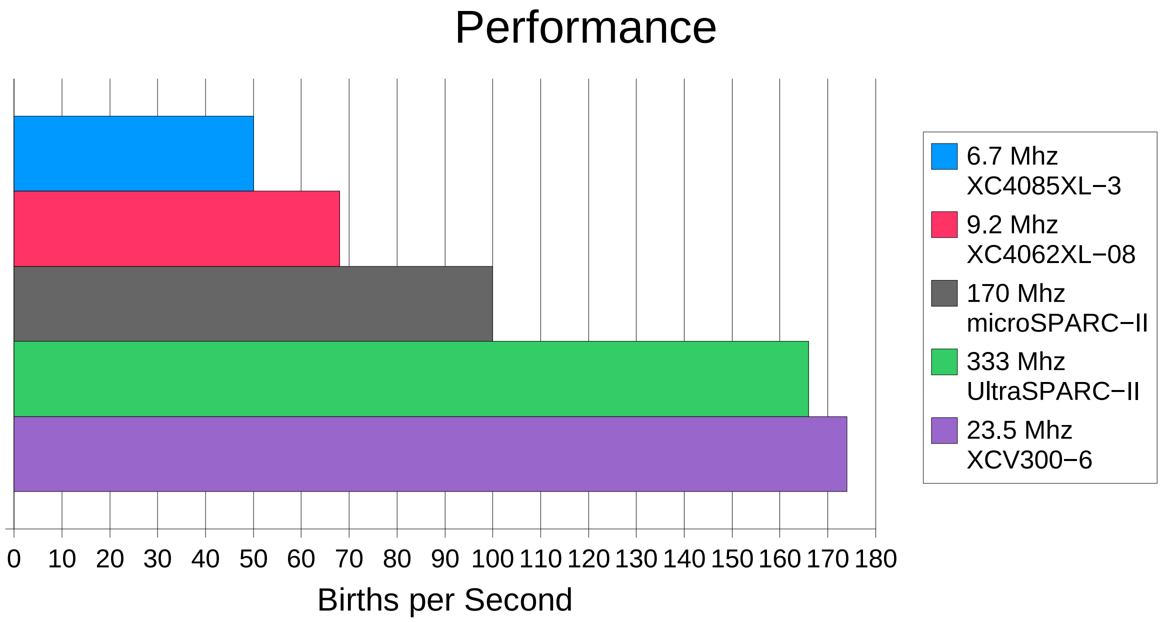


Figure 3.12: Performance comparison of TIERRA implementations

While control-intensive tasks are often considered unsuitable for RCU implementation, our TIERRA realization performed surprisingly well. As shown in Figure 3.12, the fastest FPGA considered during the implementation phase of the project, the Xilinx XC4062XL-08, achieved only a clock frequency of 9.3 MHz and barely reached 70% of the compute performance of a 170 MHz RISC processor (measured in cell divisions per second). However, when mapping the final design at the end of the effort to a then-current mid-size device (Xilinx XCV300-6), our TIERRA processor supported a clock frequency of 23.5 MHz, allowing it to exceed the performance of a 333 MHz super-scalar RISC.

The main lesson learned from the TIERRA processor project was, that reconfigurable computing can improve performance even in situations where a fully spatial distribution of the application is not possible. Just by being able to match the temporally shared computation structure to the needs of the algorithm, high performance at relatively slow clock rates can be realized. Furthermore, the TIERRA application profited from the fine-grained architecture of the FPGA-based RCU. The very dense, single-bit-oriented controller making up the bulk of the design would have been very inefficient when mapped to a coarse-grained RFA.

3.4 ACS Implementation Techniques

Before realizing an application on an ACS, it is worthwhile to consider some of the techniques feasible on the target technology. This also applies strongly to the development of automated software tools for mapping or assisting in the mapping of algorithms to an ACS (see Chapter 4).

3.4.1 Hardware-oriented Algorithms

Even though an ACS does contain a software-programmable CPU, only by examining the application from a *hardware-centric* viewpoint can the paradigm of spatially distributed computations be fully exploited. At this stage, it is interesting to consider the history of high-speed processing: Digital signal processing started in the late 1950s not on software-programmable processors, but by implementing all functionality in custom hardware. To this end, a host of specialized algorithms has been developed that were relegated to dusty tomes with the emergence of software-programmable computers. These hardware-centric algorithms are still extremely useful for implementation on an RCU.

As an example, consider the magnitude calculation for a vector (a, b). Mathematically correct, it would be computed as

$$|(a,b)| = \sqrt{a^2 + b^2}.$$

For its calculation, computer arithmetic techniques employ approximations often realized using iterative algorithms [87]. However, assuming a maximum inaccuracy of 10% is acceptable, the approximation

$$|(a,b)|' = \max\{a,b\} + 1/2 \min\{a,b\}$$

is much more efficient, in terms of both area and delay.

The Coordinate Rotation Digital Computer (CORDIC) technique invented in 1959 [88] falls in the same category. It allows the calculation of trigonometric and other transcendental functions using just shifts and adds. Since shifts are essentially free in hardware (they reduce to permutations of wires), this is a very efficient approach for realizing these otherwise expensive functions in hardware. The actual Wavelet transform in Section 3.1 was implemented in the so-called lifting scheme [89], which realizes both high- and low-pass filters using just shifts, additions, and registers. The resulting structure can be mapped to efficient hardware with only minimal effort. Of course, given the abundance of on-chip memories on many reconfigurable devices, direct lookup-tables of precomputed results are the fastest realization for many functions with small domains.

3.4.2 Custom Number Representations

In the simplest case, the bit width of RCU operators can be matched to the numeric range of the data as finely as the fabric granularity allows. For example, the Quantization stage of the Wavelet compression has to process a maximum integer value of 380. Thus, the required comparators have been realized as 9b wide operators, saving both area and delay (due to shorter internal carry-chains).

As the next step in adaptation, standard number representations can be slightly altered to better fit the algorithm. Very common is the use of fixed-point formats. For example, the format called 1.15 partitions a 16b word to have 1b in front of the binary point and 15b to its right. This allows the representation of values in the range 0 to 1 32767/32768 with a precision of 1/32768. Addition and subtraction operators on these representations are identical to their integer forms. Multiplication and division do require scaling afterward, but this can often be realized by simple truncation (dropping of excess bits to arrive at intended width). Similarly, standard floating point formats can be modified to achieve either a increased precision with a reduced dynamic range (or vice versa). While both of these adaptations can lead to smaller areas and delays, they have to be performed carefully. Reducing the precision of a floating point representation, or even moving to a fixed-point one, introduces additional noise into the system (the difference between the actual value and that representable in the current number format). These design decisions have to be made carefully, possibly supported by software tools such as MATLAB/Simulink [90], to ensure that the minimum quality-of-results requirements of the algorithm are not violated.

Beyond these modifications of standard formats, finely-granular RCUs support the implementation and manipulation of even more exotic representations. Formats such as 1's-Complement and Diminished-1 allow the efficient realization of so-called Number-Theoretic Transforms (e.g., Mersenne, Fermat, etc.) that can significantly outperform more established algorithms such as FFT [91].

3.4.3 Partial Evaluation and Late Binding

Both of these techniques exploit the presence of known values to optimize the computation structure implemented on the RCU. This is most efficient for finely-granular fabrics, since even constant single bits can be used for logic optimization there.

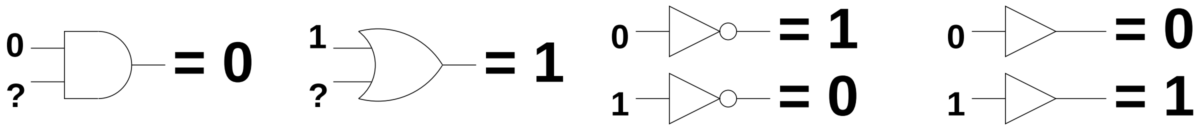


Figure 3.13: Partial evaluation replaces gates with constants

Consider the examples shown in Figure 3.13. In all of these cases, the presence of a known value on even one of the gate inputs allows the optimization of the entire gate, replacing its output with a constant value. By propagating constants in this manner, entire circuits may be reduced in area and delay. Practical applications include constant multiplications in DSP algorithms and cryptographical hardware specialized for the current encryption/decryption key. Since partial evaluation significantly affects the circuit structure, it is generally performed early in the circuit generation process (HDL synthesis, module generation). Partial evaluation figured heavily in the controller of the TIERRA processor (Section 3.3). Since the lowest-level control words consist of a mix of constant and variable bits, they could not be implemented statically using a lookup-table as ROM. Instead, they were described explicitly by logic equations. Partial evaluation was then applied in the Verilog logic synthesis step to maximally exploit the constant bits.

In cases where some variations, e.g., in multiplicand values has to be allowed, a partial reconfiguration approach can be used on suitable RCUs. Late binding is a technique that changes the circuit function, but retains the structure of the mapped circuit (including

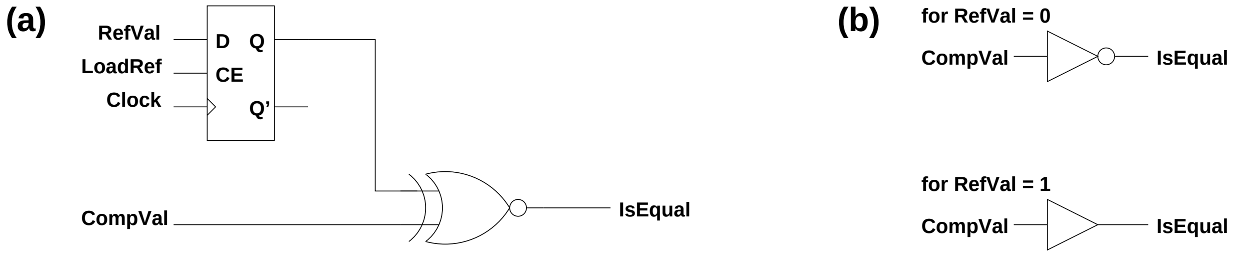


Figure 3.14: Variable comparator: (a) programmable, (b) using late-binding

placement and routing). A simple example is shown in Figure 3.14: (a) depicts a typical realization of a reprogrammable comparator. A register writable, e.g., by the CPU using LoadRef, holds the reference value RefVal of the comparison. It is compared to the input value CompVal using an XNOR gate, which outputs a '1' on IsEqual when the values match. The same behavior can be realized using the much smaller structure shown in (b). Depending on the reference value, the reconfigurable cell is changed to either a buffer (for a value of '1') or an inverter (for a value of '0'). The behavior of the resultant circuit is identical to the former one.

Apart from being profitable mainly on fine-grained RCUs, late binding is only applicable to scenarios where the changes occur relatively infrequently: Reconfiguration (even partial) is generally slower than reprogramming. A limited form of late binding is used in the distributed arithmetic approach [92] to implement efficient multipliers having a rarely-changing multiplicand. Here, the partial reconfiguration is limited to changing the contents of small memories holding pre-computed partial products.

3.4.4 Parallelism

Parallelism can occur at multiple levels in an ACS implementation. The most obvious one, parallel execution between CPU and RCU, is generally not used in our computation model. Even though the ACE-V fully supports such a mode (both in hardware as well as in OS services), none of our applications to date would have profited from it. It is possible, however, to conceive scenarios where the CPU performs external I/O (e.g., disk, network) in a double-buffered fashion with the RCU already processing the next block of data. However, a more intelligent master-mode RCU could dedicate some of its configurable resources for performing the same task. In some cases, such as high-speed networking (multi-gigabit per second), the CPU might not even be capable of sustaining the required I/O rates.

Thus, our use of parallelism in an ACS occurs at lower levels. In our experiments, we have isolated the following areas of parallel execution in an ACS:

Threads

Thread-level parallelism occurs at a relatively coarse granularity within an algorithm. Here, complete functions or even chains of functions are executed in parallel.

An example of thread-level parallelism is used in the image segmentation application of Section 3.2: In Phase 1, one thread unpacks the bit-compressed image and assigns preliminary label values, while another inserts newly discovered transitive adjacencies into a list. The second thread just monitors the stream of preliminary label values generated by the first one, it executes completely independently otherwise.

Pipeline

Pipeline parallelism splits one thread of execution into separate phases that can then execute in parallel, each processing a separate data item. It is a traditional means of implementing complex sequential flows in hardware. Thus, it also figures heavily in the computation structures implemented on RCUs.

Both of the image segmentation and compression applications (Section 3) use pipelining in all of their processing phases. When data-dependencies prevent the forming of a single pipeline, the computation can often be realized by splitting it into sub-pipelines, which are then executed sequentially when the data-dependencies have been satisfied. This has been used in the image compression implementation, splitting the complete processing flow into separate Wavelet transformation and Quantization/Run-length/Huffman processing pipelines that execute sequentially.

Subword

Another name for this kind of parallelism is small-vector parallelism, which traces its origin to the SIMD computation model implemented in vector computers. However, instead of the large vectors common for such systems (e.g., 256x64b words on the NEC SX-6 [93]), small vector units as implemented on modern processors and ACSs such as the Stretch S5000 [54] have total vector widths of up to 128b. These can then be partitioned as needed for 16x8b, 8x16b, 4x32b and 2x64b vectors. For primarily data flow-oriented computations, which are limited by the bandwidth of external data sources and sinks (peripheral bus for attached RCU, width of RCU-local memories), the effective total vector length is determined by the width of the external bus (often 32b or 64b).

When the Wavelet image compression (Section 3.1) was implemented on the ACE-V platform, the RCU processed in parallel the four 8b pixels that could be transferred at once over the 32b PCI bus. In later stages, the width of intermediate data expanded to 16b, allowing a parallel processing of only two elements at once (this time limited by the 32b wide RCU-local memories).

Memory

Parallel memory accesses are one of the major differences between an ACS and a von Neumann-CPU. When programming the latter, the memory space appears homogeneous. At best, high-performance applications consider a matching of access locality to the characteristics of the cache subsystem. On an ACS, the true structure of the memory system is exposed to and exploited by the RCU. This applies not only to the organization (e.g., the four independent RCU local memories on the ACE-V), but also to bandwidth and latency issues (e.g., the slow main memory access via PCI on the ACE-V). Optimal parallel use of these heterogeneous resources can require the insertion of buffering/prefetching support (see Section 4.6.3).

All of the applications considered in Section 3 distribute their data across multiple memory banks: The image compression uses them to store the wavelet coefficients at different hierarchy levels and performs three accesses in parallel. Furthermore, it also assigns two RCU-internal memory banks of 512 B each as FIFO buffers for faster main memory access via PCI (allowing the issue of PCI burst instead of single transfers). The image segmentation implementation actually performs four concurrent memory accesses in two of its three processing phases (Figure 3.7). And even the TIERRA processor, despite its otherwise limited parallelism, makes use of SPARXIL's two memory banks: The organism state (registers, stacks) and jump label cache are accessed in parallel to the memory holding the organisms themselves (the "soup") and the thread management data (Figure 3.11).

From the preceding sections, it should be obvious that the highly specific exploitation of parallelism in its many forms is the key to the advantages of ACSs over conventional CPUs. While the latter also attempt to extract parallelism from a sequential program, techniques such as branch prediction, speculative execution and super-scalar execution carry a large overhead in silicon area and do not yield linear speedups. For example, many of the calculations performed speculatively will be discarded once they are discovered to have been superfluous. Before that, however, they required the use of processing units (silicon area) that did consume energy for their operation. In a well-designed ACS application, such waste will not occur.

4 Tools and Infrastructure

In order to exploit the ACS capabilities using the techniques discussed in the previous chapter, a non-trivial degree of expertise in hardware design is required. While today this generally stops at the level of register-transfer logic (RTL), and does not descend to the increasingly analogue depths of real deep sub-micron ASIC design, it still poses a significant hindrance to the hosts of programmers familiar with the software programming of conventional processors.

Thus, support programs in the form of software tools have long been used to assist in the programming of an ACS. Splash-1 could be programmed by algorithmically describing hardware structures in an extension of the Lisp language [58]. This technique, generally called *module generation*, allows the creation of highly optimized hardware blocks and is still in use today (Section 4.7.1). However, it still requires the ability from the programmer to think in and express hardware structures. Splash-2 was programmed in the VHDL hardware-description language, which was then *synthesized* to actual RCU configurations [73]. While this did lift the abstraction level to the register-transfer form still common today, software programmers were still left out.

In order to close this gap, research on design flows (Figure 4.1) raising the abstraction level even higher can draw from previous work in high-level synthesis of hardware circuits [94] [95] as well as work on parallelizing compilers [96] and compilers for computers with very-long instruction word (VLIW) processors [97] [98]. However, while all of these efforts have resulted in significant advances in their respective fields, the problem of efficiently programming the hybrid CPU/RCU architecture of an ACS has only recently been tackled [41].

Furthermore, the computation structure generated by the compiler (often in the form of a datapath and controller [99]) has to be actually mapped to the RCU. Depending on the RCU architecture, this so-called *back-end* can be quite involved. Especially for finely-granular RCUs (e.g., realized by an FPGA), placement and routing procedures similar to those required for ASIC layout need to be employed. For best results, they should be assisted by floorplanning and module generation functions that specifically optimize the datapath.

To actually exploit the full potential of the ACS approach, all parts of the design flow, from the compiler down to mapping tools, need to be considered and matched to each other as well as the RCU architecture. An overview over these issues was given in [39] and [100].

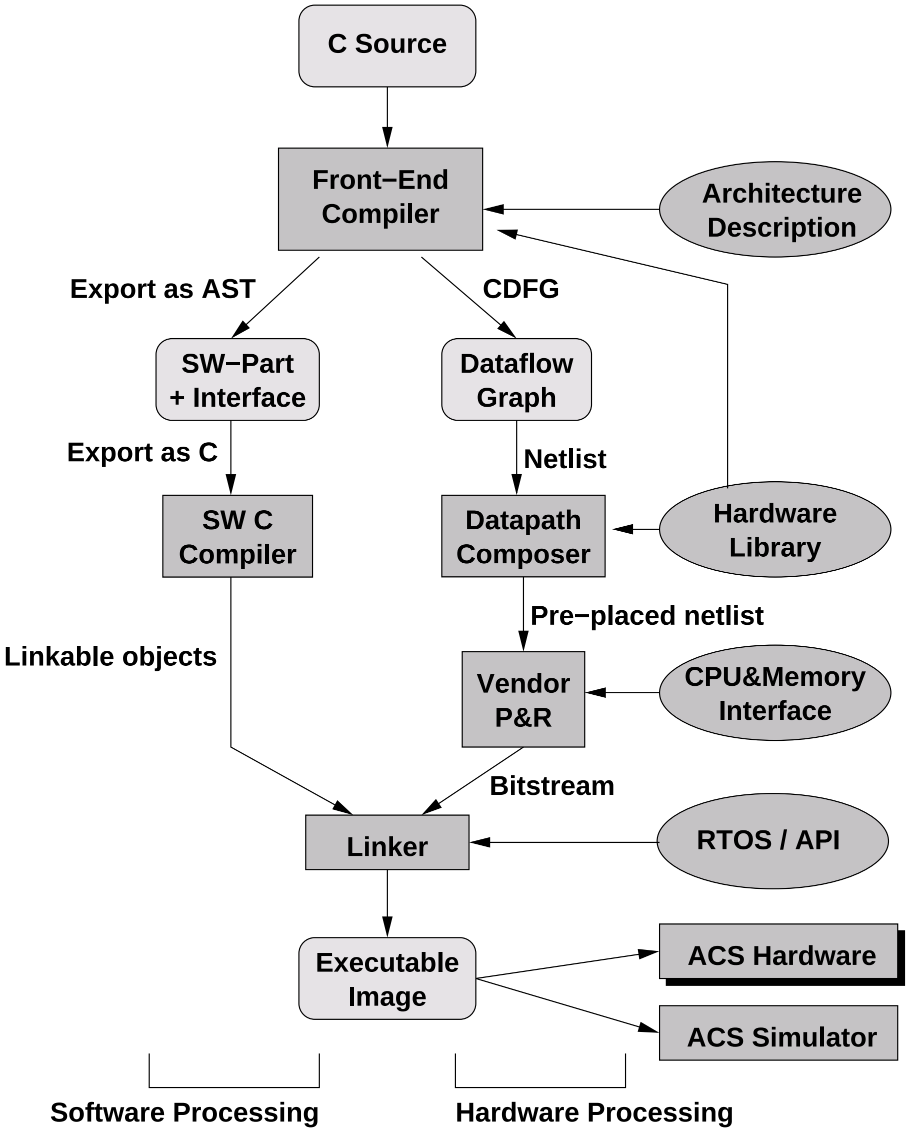


Figure 4.1: A sample compile flow for ACSs

4.1 Compilers for Adaptive Computers

Due to the complexity of solving the complete problem, most research in ACS compilers tackles only a subset. Table 4.1 gives an overview over some existing systems and their capabilities and limitations.

All of the systems shown here accept a high-level language as input that originates in the software domain. In general, however, only a subset of that language is actually compilable by the system. While less advanced systems can only process simple sequences of instructions, others allow the crucial compilation of loops. Some systems have the constraint that only loops with fixed bounds can be compiled. Other limitations apply to function calls and recursion. An important class of restrictions deals with the data types that can be handled. While simple systems are limited to just scalar variables, more advanced ones can handle arrays (but sometimes restricted to a single dimension). Even if array accesses are actually supported, some systems can handle only affine indices (of the form $ax + b$, with x being the iteration variable). Only very few systems support arbitrary memory accesses using pointers.

Also, some compilers require the user to manually annotate the original HLL source with directives to aid or even enable the compilation. The authoring of these directives generally requires experience in hardware design. The compilers also differ in their capabilities, e.g., with regard to loop transformations [96] and whether they are able to automatically partition the application between CPU and RCU execution.

While some back-ends can generate configurations ready for loading onto the RCU, others require external hardware synthesis steps (often RTL, but sometimes even at the behavioral level). As previously mentioned, the generation of efficient datapaths can profit from specialized placement directives provided by the compiler. However, not all systems can compute these. Finally, the memory architecture supported by the different compilers varies. While very simple systems do not support memory accesses at all, more advanced ones might support fast data streams for regular address patterns and even exploit caches for accelerating irregular address patterns.

4.2 High-Level Language Compilation for ACSs

Fig 4.2 shows the anatomy of a flow for compiling a high-level language to an ACS target, with some example for operations in each of the compilation phases. All of the steps in the front-end (parsing the input language, building an abstract syntax tree etc.) and many target-independent optimizations (e.g., common subexpression elimination, constant folding, dead-code elimination, etc.) are handled identically to compilers for conventional processors [109] [110]. This also includes steps for improving the locality of data memory accesses, since cache hierarchies are also used on an ACS target.

In order to exploit the different kinds of parallelism achievable on an ACS (see Section 3.4.4), much of the lore on building compilers for parallel computers is still applicable [111] [96] [98]. Specifically, all transformations suitable for systems with low-latency com-

	Chimaera	DEFACTO	Pipelined Vectorization	C-to-HDL	STREAMS-C	GARP-CC
	[101]	[102] [103]	[104] [105]	[106] [108]	[107]	[46]
Input Language	C	C, MATLAB	C	C	C	C
Compilable Constructs						
Instruction seqs.	✓	✓	✓	✓	✓	✓
Loops		✓	✓	✓	✓	✓
Variable bounds					✓	✓
Function calls						✓
Recursion				✓		
Arrays		n-D	1-D	1-D	1-D	n-D
Non-affine Indices				✓		✓
Pointers						✓
Compilation does not require manual annotation of source code	✓	✓	✓			✓
Loop transformations	✓	✓	✓			
Automatic CPU/RCU partitioning	✓	✓	✓			✓
Compiler output	Configuration	VHDL	Netlist	AHDL	VHDL	Configuration
Requires external synthesis		Behavioral+RTL		RTL	RTL	
Integrated datapath placement	✓		✓			✓
Memory interface						
Caches						✓
Streams		✓	✓		✓	✓

Table 4.1: Overview of ACS Compilers

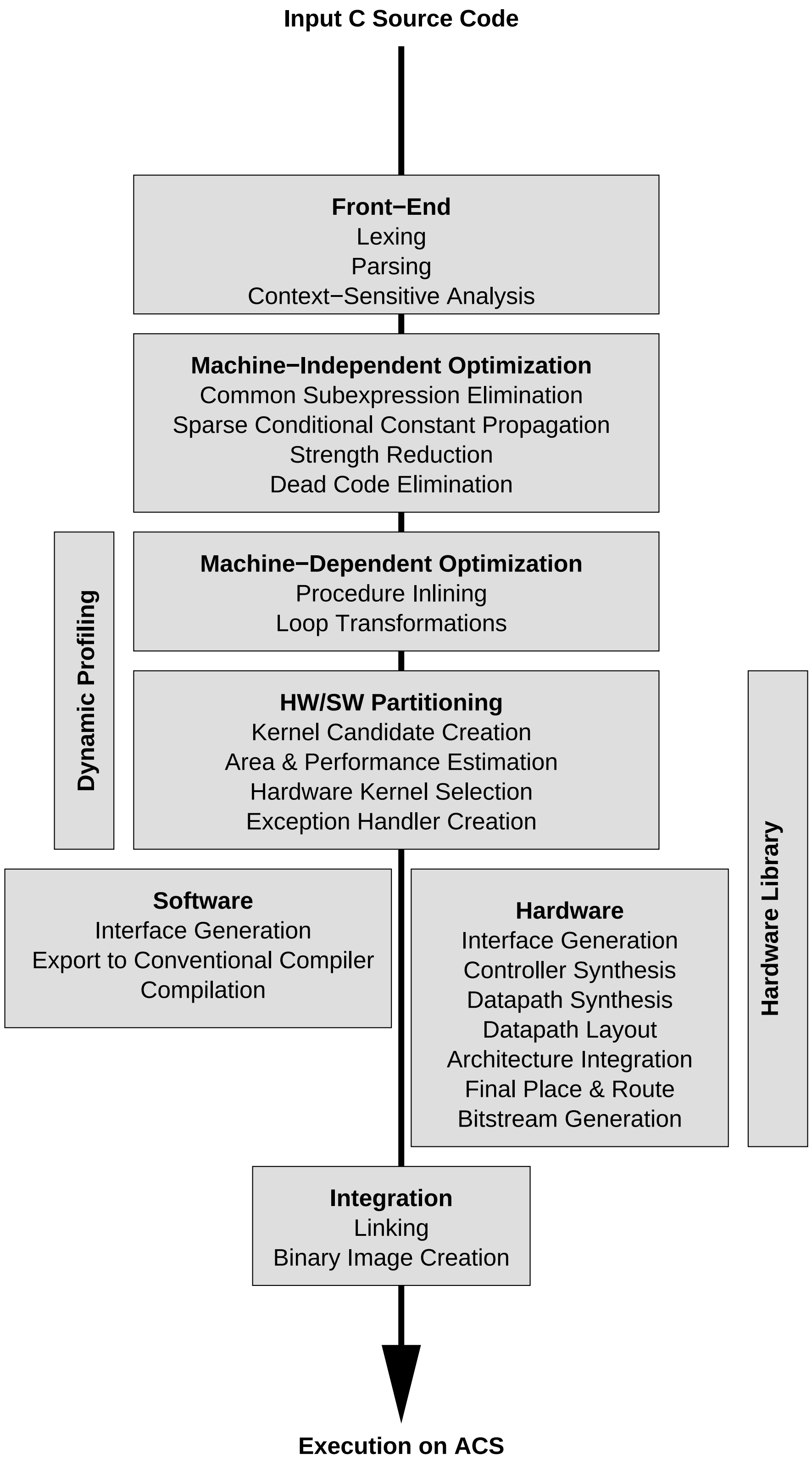


Figure 4.2: Anatomy of an ACS Compiler

munication between processing elements (e.g., vector or VLIW processors) can also be exploited for an ACS.

In contrast to an conventional software compiler, however, an ACS has to deal with issues of hardware generation and interfacing. This is a significant complication: While the conventional compiler has to target only a single architecture (however complex), an ACS compiler must generate specialized hardware architectures optimized for the needs of the individual applications. For the actual hardware generation, many techniques from the field of high-level synthesis can be applied [94] [95] [99].

However, even when building on prior knowledge both in the compiler and in the hardware synthesis fields, a crucial part of a high-level language compiler for an ACS has to deal with their boundary. Major questions include how to orchestrate the interaction between CPU and RCU and how to partition an algorithm across both units considering the reconfiguration ability of the RCU. Additional areas requiring further work are the communication protocols (also involving software components) and how to make hardware specifics visible to the compiler. The latter can be extended to also cover the IP-based design methods currently prevalent in the pure VLSI design field.

These ACS-compiler specific issues will be discussed in the next few sections. Specifically, we will address them in context of two compilers aiming to process the *full* C programming language. While this high level of abstraction lowers the entrance barrier for software programmers considerably, it also has significant influence on the design of both the compiler and the run-time hardware/software architecture.

The first system was created in a cooperation of our team with the research group Berkeley Reconfigurable Architectures, Systems, and Software (BRASS) at UC Berkeley (CA, USA), the Nimble Research Group at Synopsys, Inc. (Mountain View, CA, USA) and Lockheed-Martin Advanced Technology Laboratories (Cherry Hill, NJ, USA). It resulted in the realization of the Nimble Compiler for Agile Hardware [100] [118], which in turn was based on the GarpCC system developed previously in BRASS [46].

Based on the lessons learned from Nimble, we developed the next-generation system COMRADE [112] [113] [114] [115] [116] [117] from scratch. While it propagates some of the mechanisms that have proven practical in Nimble, in others it diverges dramatically to alleviate some of its shortcomings.

4.3 Effects of the ACS Architecture

The architecture of the ACS target architecture has a major influence on the operation of the compiler. Even if it is isolated from low-level hardware details (see Section 4.7.1), the compiler has to take a number of characteristics into account.

The degree of coupling between the CPU and the RCU determines the complexity of operations that can be efficiently executed on the RCU. Figure 4.3 shows two extreme cases: In a tightly coupled system using an RFU (Section 2.4.5), for example, even individual statements could be executed on the RCU (e.g., complex bit-manipulations that the RCU can perform in a single cycle). Generally, however, only larger kernels holding one or

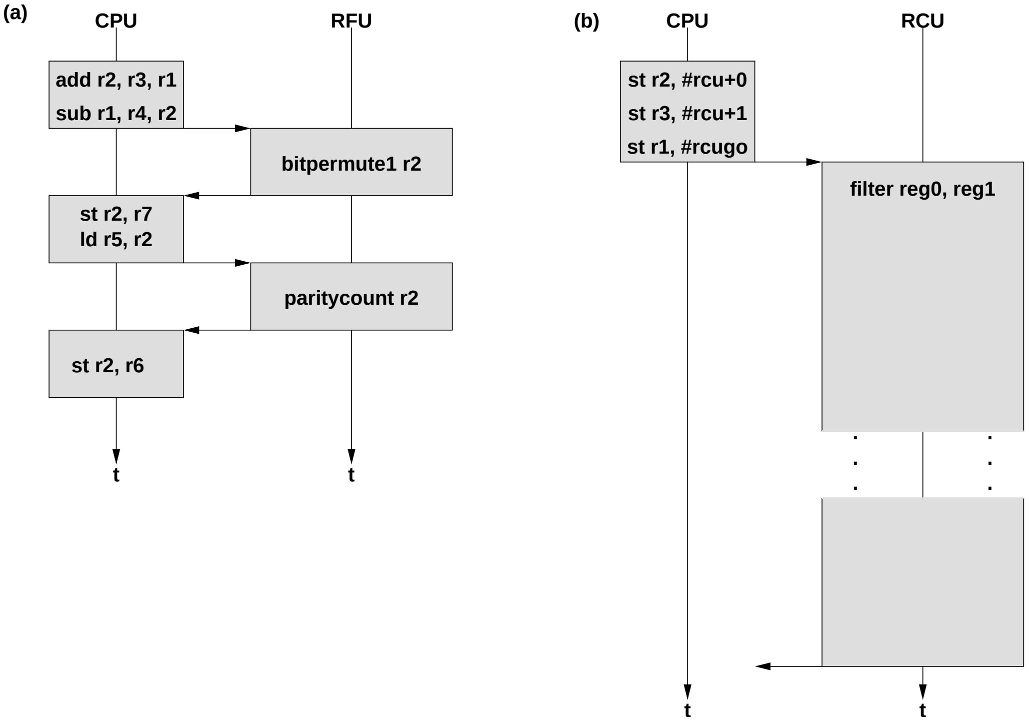


Figure 4.3: Kernel granularity: (a) tightly coupled, (b) loosely coupled

more possibly nested loops are mapped to the RCU (e.g., the complete filter operation sketched in Figure 4.3.b). This reduces the communications overhead, since the RCU can now execute for longer periods of time without requiring CPU intervention. The latter is the approach used by both Nimble and COMRADE.

The reconfiguration speed of the RCU determines how many different kernels (Section 2.4) can be executed as hardware versions during the runtime of the program (Figure 4.4). If the RCU can reconfigure quickly (b), all kernels can be executed in hardware, reconfiguring in between, and realizing significant speedups over the original all software version. For FPGA-based RCUs with their glacial configuration times, it is not profitable to execute all kernels in hardware: The intervening reconfiguration times completely obliterate any performance gains achievable by RCU execution (c). For these devices, at best, only a single kernel can be selected to achieve any speedups at all (d).

The capacity of the RCU and the number and organization of independent memory banks influences the degree of parallel processing on the RCU (Figure 4.5). With multiple memory banks and larger space for realizing multiple instances of the operators, many algorithms can be partitioned to execute on different parts of their input data in parallel. Compiler algorithms for recognizing this structure are well known [96] [111], but require information on these characteristics for actually making the trade-off decisions.

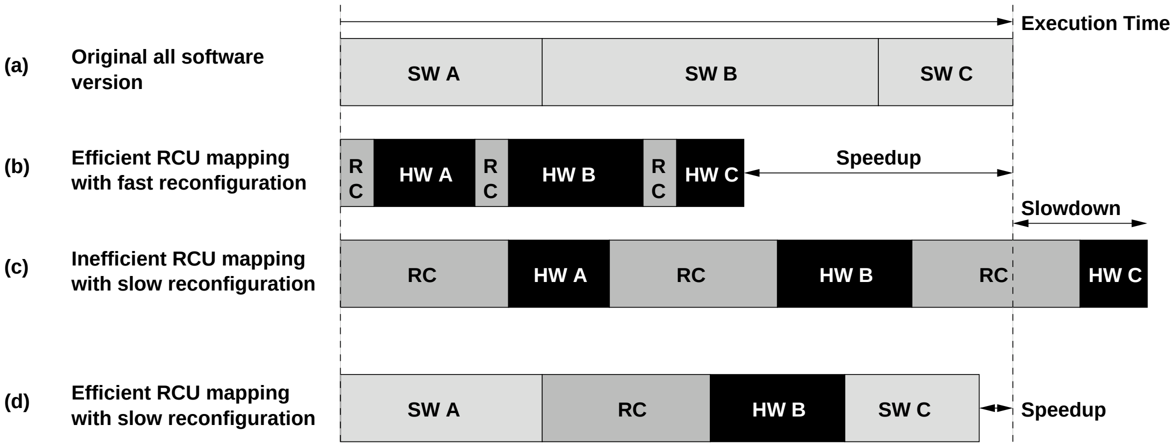

Execution Time
(a)
Original all software version
SW A
SW B
SW C
(b)
Efficient RCU mapping with fast reconfiguration
R C
HW A
R C
HW B
R C
HW C
Speedup
Slowdown
(c)
Inefficient RCU mapping with slow reconfiguration
RC
HW A
RC
HW B
RC
HW C
(d)
Efficient RCU mapping with slow reconfiguration
SW A
RC
HW B
SW C
Speedup

Figure 4.4: Effect of reconfiguration time on RCU usage

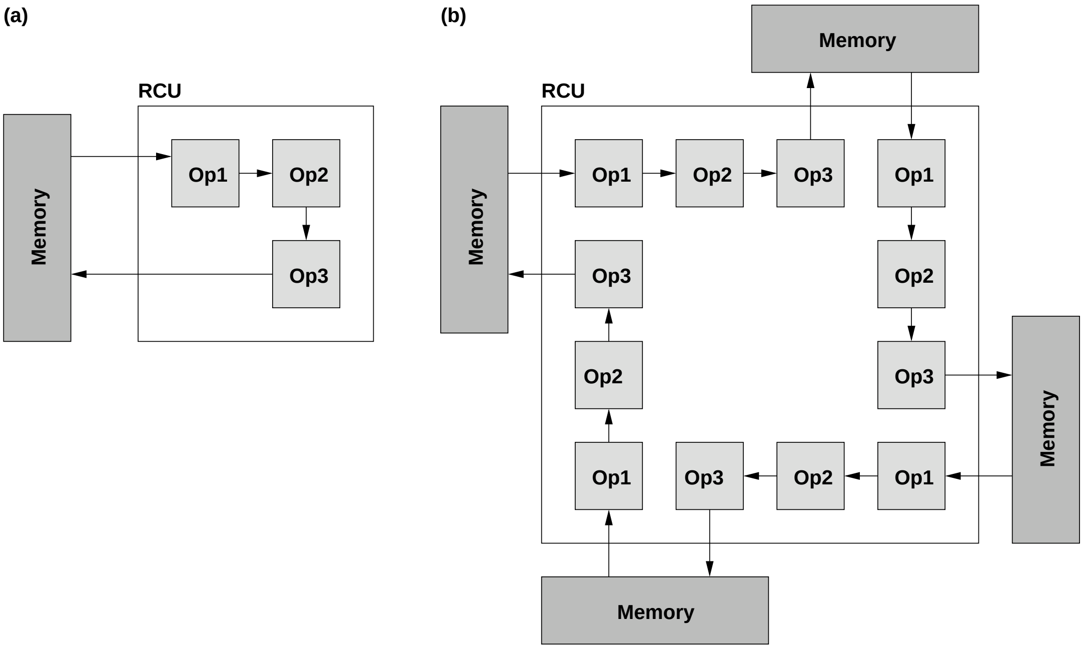


Figure 4.5: Effect of available RCU area and memory banks on parallelism

4.4 Models of Computation

Even with the increased flexibility afforded by the RCU, an automatic compiler still needs some fixed scheme how it should actually use the ACS components. Before we describe this *model of computation*, we will compare and contrast the usage of an ACS with conventional computers and actual hardware (chip) design techniques.

When attempting to translate C, the ACS target environment differs both from that of a classical software compiler as well as from that of a high-level hardware synthesis system. The first one can translate all of the input language to execute on the fixed-architecture CPU, but has no concept of hardware specifics. The second one generally accepts only a highly constrained subset of the language (e.g., no function calls, no pointers etc.) for which it can generate a hardware implementation. If constructs outside of this subset are used, the translation is usually aborted.

Our aim is the translation of the *full* C language to an ACS target. To this end, it is mandatory, that all valid C programs can be *completely* translated using the ACS compiler and that the binaries execute *functionally identical* on the ACS and the conventional computer. Additionally, suitable parts of the program *should* execute in a spatially distributed implementation on the RCU as 'hardware', resulting in shorter total application run times.

4.4.1 Integration of the RCU in the Computation Flow

One of the major aspects of the model of computation is the manner of orchestrating the cooperation between CPU and RCU. For the purposes of the following discussion, a variable defined in the C input program may be implemented as a software variable (held in CPU accessible memory location or a CPU register) or as a hardware register (held in the RCU datapath for the kernel). Only one of these implementations is valid at a time. However, a variable may migrate between the two, e.g. by writing a CPU register value into the corresponding RCU register, or by writing the value of an RCU register into the memory allocated by the CPU for the variable.

Figure 4.6 shows a simple case of the partitioning and interaction between CPU and RCU, resulting from the compilation of the given source code fragment.

A kernel mapped to the RCU is wrapped with an interface for exchanging data with the rest of the program executing in software on the CPU. This interface consists of software instructions, synthesized during compilation, that migrate data between software variables and hardware registers. On the hardware side, logic for accessing the required datapath registers from the CPU must be generated. Furthermore, facilities on the the hardware and software sides exist to start the RCU and indicate its completion to the CPU.

On entry to the kernel, all software variables used by its operations are transferred into hardware registers (namely `u`, `v`, and `n` into registers 0, 1, and 2. On exit from the kernel, a subset of the variables held in hardware registers are transferred back into their corresponding software variables. This applies only to those variables that were modified during the RCU execution and will be read in the rest of the program. In the example,

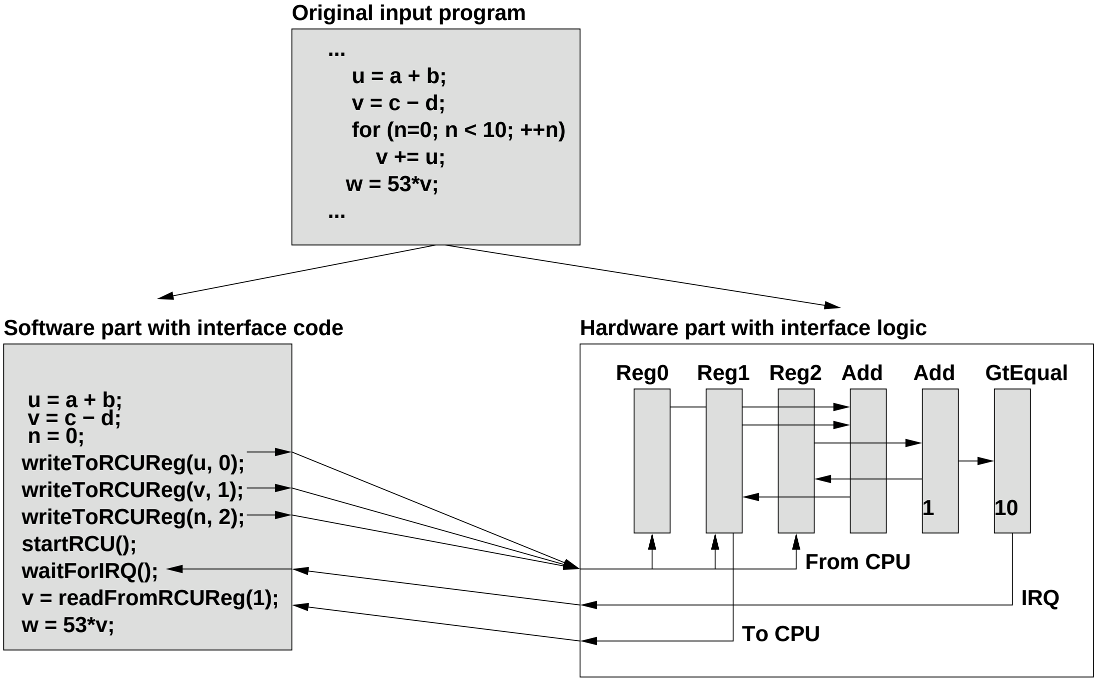


Figure 4.6: Simple case of HW/SW partitioning and interfacing

this is just variable v (we assume n is not read outside of the kernel). In classical compiler terminology, only *live* variables will be exchanged during the hardware/software transitions.

For kernels having multiple exits (e.g., a break statement in a loop and the normal loop exit), the wrapper also indicates to the software program which of the exits was taken. This mechanism, the hardware side of which is described in Section 4.6, is also used to signal exceptional conditions (see below).

4.4.2 Restrictions on RCU Usage

On the ACS, we can fall back to the CPU for the parts of the input program that contain constructs that can either not be mapped to RCU hardware at all or at least not efficiently. For example, the printf() output function of the C standard library is not very amenable to RCU implementation (highly control dominated, many special cases that would require large RCU areas, generally not performance critical, needs to access output subsystem in operating system). Instead of aborting the translation process here, this part of the input program is translated by the ACS compiler for execution on the CPU. For many of today's RCUs, standard floating-point operations fall into a similar category: While they could be mapped to a spatially distributed implementation on the RCU, they are still much more efficiently realized using the CPU's dedicated floating-point unit. For purposes of this discussion, all of these operations not suitable for the RCU will be called *infeasible*.

```
...
u = a + b;
v = c - d;
for (n=0; n<10; ++n) {
  v += u;
if (v > 1000)
  printf("warning: v too large %d", v);
}
w = 53 * v;
...
```

Figure 4.7: Input program with RCU-infeasible statement

4.4.3 Handling RCU-infeasible Operations

Given the kernel-level scope of mapping parts of the input program to the RCU, the presence of even a single infeasible operation inside of a kernel would completely prevent its mapping to the RCU. Consider the example shown in Figure 4.7: In the traditional approach, the compilation would be aborted here, and the programmer would have to rewrite the offending printf() statement manually.

Instead of this static view of failing just because of the presence of an infeasible operation in the input program, a better analysis would consider how *often* the infeasible operation is executed (if at all). In many cases, C code not specifically written for high-performance computing contains error reporting statements and similar rarely used code paths (e.g., detecting a division by zero or a possible overflow as in the last example).

Thus, using dynamic profiling techniques, the compiler can determine the execution frequency of infeasible statements inside of kernels. If they occur sufficiently rarely relative to the total execution time of the kernel, the kernel can still be considered for RCU execution. However, since the actual function of the program when executing on an ACS *must* be identical to the semantics of the C code, the exceptional cases must be handled when they occur (however rarely).

To this end, the model of computation introduced in GarpCC and refined for Nimble allows the seamless switching of hardware execution on the RCU back to software execution on the CPU. This capability allows the processing of RCU-infeasible operations by an appropriate software handler running on the CPU.

In Nimble [100], this software exception handler is created automatically by always maintaining two versions of a kernel: One complete version executable on the CPU, and another one containing only feasible statements mapped to the RCU.

Figure 4.8 shows this approach on the example. For clarity, it does not depict an actual datapath for the hardware kernel, but shows its equivalent function as statements. When entering the kernel, hardware execution on the RCU is attempted first. If no exceptional conditions occur (only paths actually realized on the RCU are taken), the RCU executes the complete kernel and then passes control and data back to the CPU (via exitToSW(0) in the wrapper).

If, however, a path leading to a hardware infeasible operation was indeed taken (the

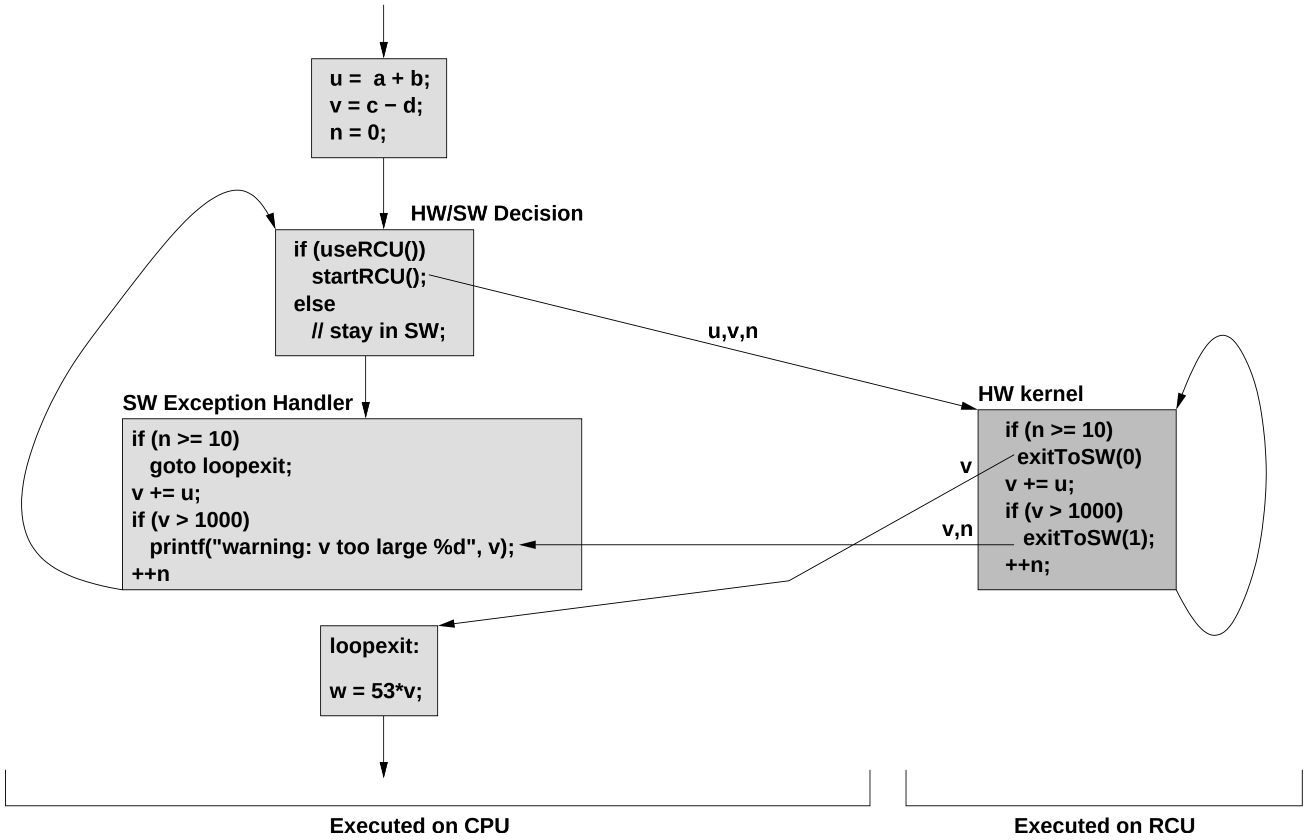


Figure 4.8: Handling RCU-infeasible statements in software in Nimble

overflow of v in the example) , the RCU execution of the kernel is stopped by taking an exceptional exit. This fact is communicated to the CPU using the normal exitToSW() mechanism for handling multiple exits from a kernel. The wrapper instructions then evaluate the reason for the exceptional hardware exit (now 1 instead of the normal 0), and restart the CPU at the address holding the software version of the RCU-infeasible operation (here beginning with the printf() and the incrementing of the index variable n).

Since the wrapper also copies the live variables from the RCU back to the CPU (as usual for an RCU-CPU execution switch), the software version of the kernel restarts with the correct context of variable values (here, just v and n were copied back to the CPU, u was not modified in the hardware kernel). These mechanisms continue to work in the presence of multiple infeasible operations in a hardware kernel: They are just treated as multiple exits, each passing a different argument to exitToSW(). The wrapper restores the correct variable context and jumps to the appropriate address of the software version for each exit.

The rest of the loop iteration is then executed in the software version. Before the next iteration is started, a decision is made using useRCU() whether to execute it in on the CPU or on the RCU. Nimble always attempts to use the RCU first. However, a more intelligent decision making mechanism (e.g., similar to the branch history tracking on CPUs) could be integrated here for further refinement.

4.4.4 Generalized Model of Computation

COMRADE [116] generalizes the model of computation introduced in Nimble in two areas. First, Nimble was limited to processing stand-alone loops, but COMRADE can process nested loops. It proceeds with greater subtlety than just duplicating the entire loop nest (which could lead to a datapath exceeding the RCU capacity). Instead, it creates an entire spectrum of hardware candidates by generating all combinations of sub-loop nests from the inside out. In this manner, all combinations, ranging from just the innermost loop(s) to the entire loop nest can be examined for hardware suitability (e.g., finding the largest nest that still fits on the RCU).

This is shown in Figure 4.9: In (a), the all-software version of the program is shown. In (b), on the far-right side, a version has been created that executes both loops on the RCU. In the middle, a version that just executes the inner loop on the RCU is located. And on the far left-side, a version exists that executes both loops in software. In addition to the useRCU() nodes that split execution between HW and SW, Common nodes mark points where hardware and software execution flows join again. At these nodes, the appropriate interfaces composed of variable transfer statements are created (see Figure 4.6).

Second, COMRADE extends the CPU-RCU interaction in the model. In Nimble, the CPU acted just as a fallback processor to handle RCU infeasible operations. The existence of these operations was considered an exceptional case and could lead to significantly reduced performance (since the rest of the loop iteration, however long, was always executed in software). In COMRADE, CPU and RCU are viewed as a pair of mutually assisting compute providers, each handling the part of the computation it is best suited

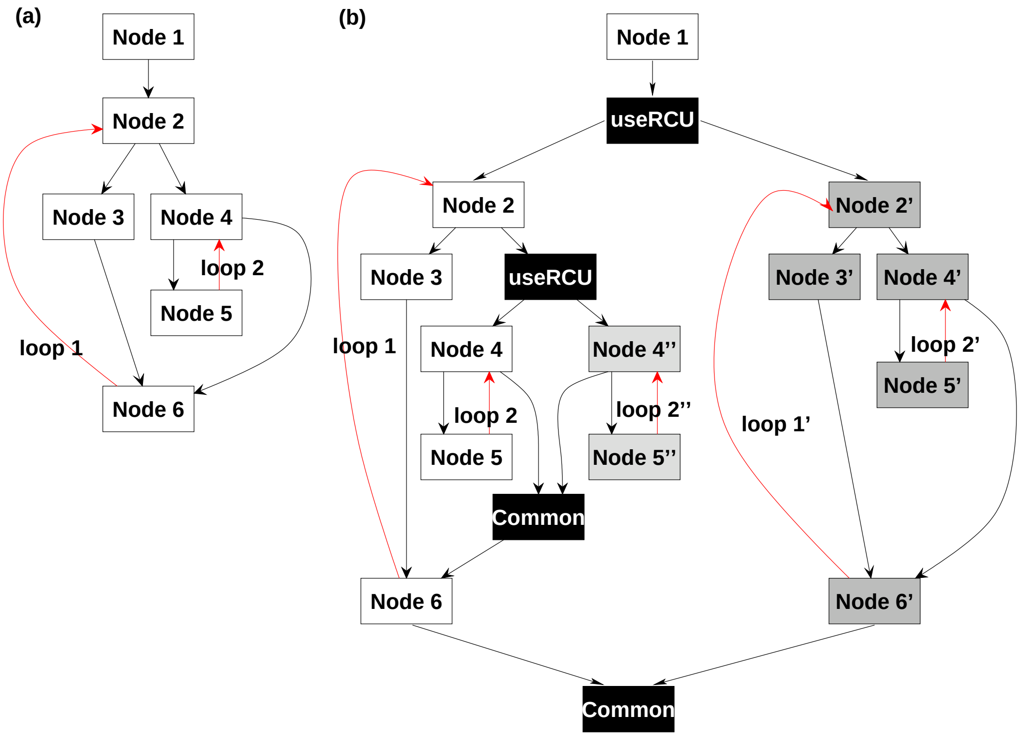


Figure 4.9: Nested hierarchical loop duplication in COMRADE

to. As soon as one processor has completed its part of the work, control is passed back to the other one: The CPU offloads the execution of kernels to the RCU for increased performance. If the RCU encounters a hardware-infeasible operation in this process, it passes control (and the live variable context) back to the CPU. However, as soon as the operation has been completed on the CPU, control passes back to the RCU (again, transferring the live variable context) and high-speed execution resumes.

This approach is sketched in Figure 4.10: During the HW/SW partitioning step, RCU-infeasible operations are discovered (a). If they occur sufficiently infrequently, just these operations are moved into dedicated software handlers, with the bulk of the computation running in hardware (b).

Figure 4.11 shows the example of Figure 4.7 in the COMRADE model: Now, the exception handler contains just the infeasible statements, the rest of the loop body continues to be executed in hardware. Note that this approach has also improved the communications overhead compared to the Nimble model: Here, n does not have to be exchanged between the software exception handler and the hardware kernel, it remains on the RCU.

The increased flexibility of this approach comes at the price of possibly increased communication costs (more frequent switches between CPU and RCU). However, some experiments have shown that this overhead is manageable in practical applications [83]. The hardware selection step has since been improved so that only for a single of the examined applications, namely capacity, an overhead occurs: 12 32b variables are exchanged between software and hardware, for a total of 1600 reads and writes. Even in this case, both the amount of data to be exchanged as well as the number of exchanges should not

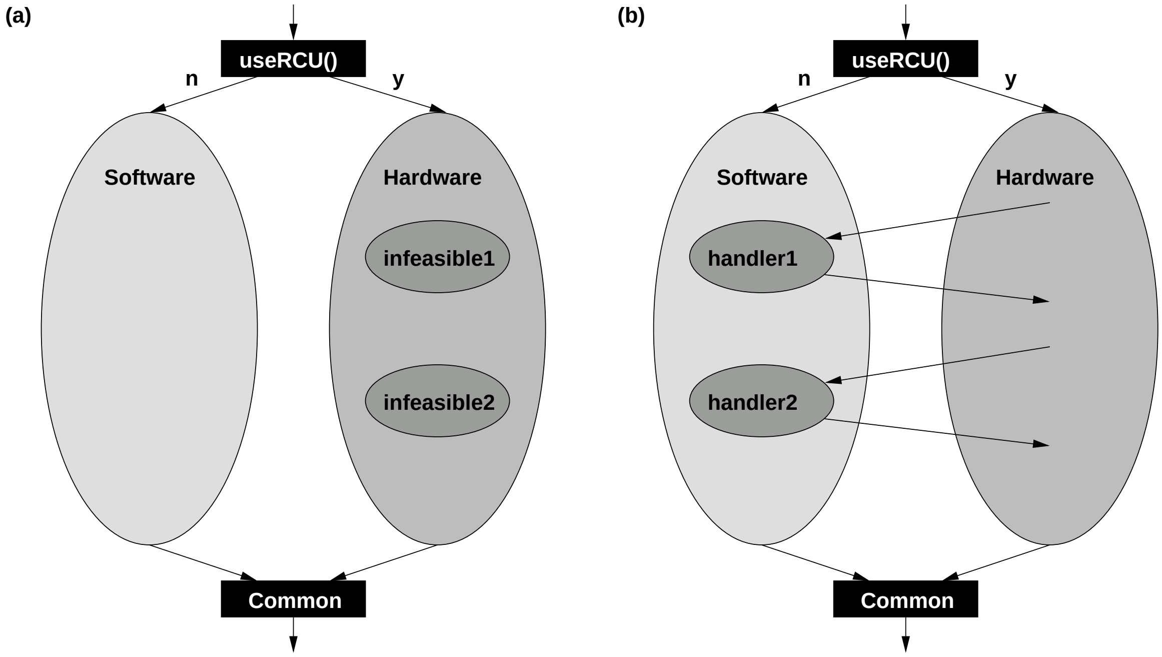


Figure 4.10: RCU-CPU interaction in the COMRADE model

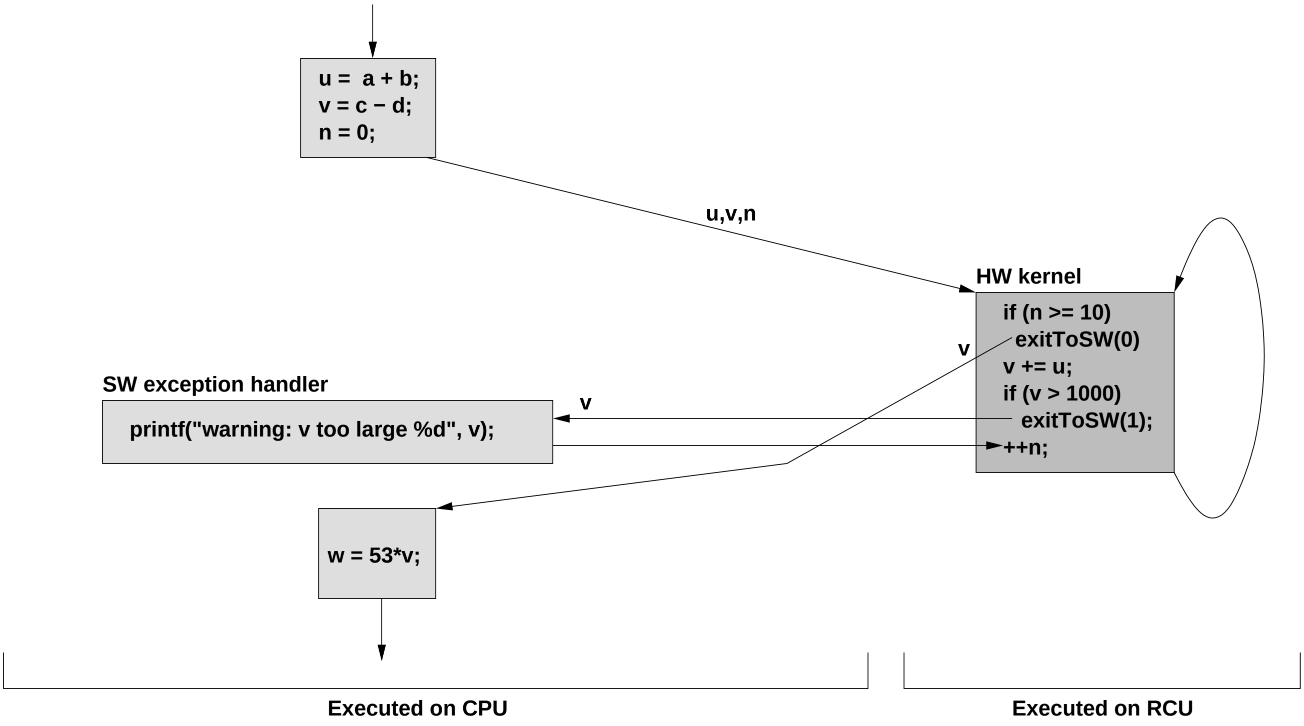


Figure 4.11: Example using the COMRADE exception handling model

lead to excessive overhead on modern peripheral or on-chip busses (low latency, high bandwidth).

4.5 Kernel Selection

As described in the last section, considerable flexibility exists in how to actually partition a program between CPU and RCU execution. It is here that the ACS compiler has to trade-off possible performance gains versus various costs. Some of these costs might include:

- The area required on the RCU for implementation
- Reconfiguration time for switching the RCU to the new hardware
- Communications overhead between RCU, CPU, and memory

The RCU area required by each of the candidates can be quickly determined with good precision. Nimble uses tables of associating each basic operator in the AST with a fixed area, while COMRADE accesses a library of parameterized modules to handle a wider variety of operators (e.g., a 24b adder, see Section 4.7.1). With this data, the compiler can then eliminate all alternatives that exceed the RCU capacity. Since the next partitioning steps in the heuristic [116] are somewhat more involved, shrinking the search space is a necessity.

In both Nimble and COMRADE, the partitioning of the program between CPU and RCU execution is mainly guided by dynamic profiling. Nimble cross-compiles an instrumented version of the program and executes it on the actual ACS CPU, gathering execution time data. While very precise, this approach actually requires a fully operational ACS connected to the computer hosting the compiler. Especially in the embedded area, this might not be practical.

COMRADE trades precision for portability: Instead of measuring execution timings on the actual target CPU, it collects per-statement execution counts [116]. Since this is independent of the characteristics of the target environment, the profiling runs can be performed on any computer. These execution counts, combined with the target CPU-specific number of clock cycles per operation, then form the basis of estimated execution times. While a linear correction factor attempts to model the effect of pipelining and cycle-per-instruction values of less than 1, the estimate is a very coarse one. However, it can still serve to identify the most interesting candidates for kernel extraction.

COMRADE also tracks the CPU-RCU communication overhead during the dynamic profiling, something disregarded in Nimble. However, this characteristic is currently dwarfed by the noise in the execution-count based run-time estimates. A better solution would be the interfacing of the profiling mechanism to a software simulator of the ACS CPU (e.g., ARMulator for ARM CPUs, SIS for SPARC CPUs or PSIM for PowerPCs). This would combine the precision of the time-based profiling in Nimble with the hardware independence achieved in COMRADE.

Application	Kernel Size in CLBs		
	Min	Average	Max
versatility	473	1398	3064
adpcm	1395	1395	1395
capacity	33	184	561
g721	97	910	2041
pegwit	186	823	2116

Table 4.2: Largest kernels extractable using COMRADE

4.5.1 Reconfiguration Scheduling

While the actual reconfigurability is one of the greatest advantages of an ACS, the most widely available and largest capacity reconfigurable devices are fine-grained FPGAs, which have not been designed with rapid reconfiguration in mind.

Nimble put considerable effort [118] into the static scheduling of reconfigurations at compile time. This scheduling was aimed at performing full reconfigurations, since the RCU on the target ACS had only limited tool support for partial reconfiguration. However, despite promising theoretical results, not a single application examined was able to compensate the huge practical reconfiguration times with an increased performance.

Learning from this experience, COMRADE aims to to select the single most promising kernel from the application to avoid multiple reconfigurations. Since it can handle nested loops and increase the kernel size (and potentially the parallelism) by profile-directed inlining, it trades a larger number of smaller kernels (and their associated reconfigurations) for a single larger kernel that potentially has a higher degree of parallel execution.

However, even with our attempts [116] to grow the kernel sizes, we do not fully exploit the complete area available on the mid-size Xilinx Virtex 1000 FPGA, 6144 configurable logic blocks (CLBs), we have been targeting thus far. Table 4.2 shows the largest kernels extractable using a current version of COMRADE. At best, the largest single kernel fills just half of the device, the average kernel size is often considerably lower.

To this end, we are currently investigating another method to increase performance when implementing multiple kernels, shown in Figure 4.12.

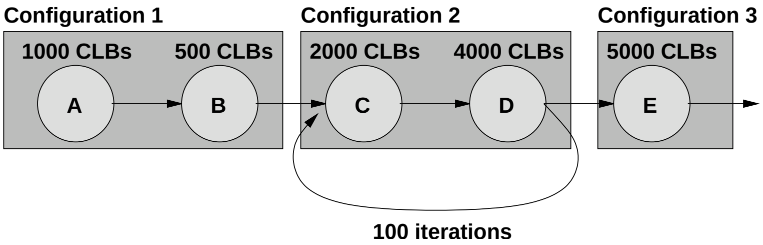


Figure 4.12: Profile directed fusing of kernels A. . . E into configurations 1. . . 3

Instead of putting each kernel into a separate configuration, our new approach relies on fusing smaller kernels into a single configuration, also guided by the dynamic execution profile. The latter is important: Assuming an RCU device capacity of 6144 CLBs, a simple greedy packing of kernels A, B, and C into a single configuration would fit into the device, but would require excessive reconfigurations due to the loop spanning kernels C and D. This inefficiency can be avoided by basing the selection on the dynamic profiling data.

In this fashion, we can still avoid the intricacies of managing partial reconfiguration as well as the excessive configuration overhead. Switches between individual kernels are quickly performed in a single clock cycle by reprogramming on-chip multiplexers with a single 32b write. Tools for automating method (kernel fusion, floorplanning and interfacing) are currently under development.

4.6 RCU Architecture

One-chip ACSs such as the GARP [46] and coarse-grained RFAs such as the PACT XPP [62] already implement various services such as CPU-RCU communication and memory accesses in fixed logic. When targeting an FPGA, flows such as Nimble and COMRADE have to generate that complete infrastructure during the compilation process. However, this also allows a greater flexibility to match even these parts of the computation structure to the needs of the application.

4.6.1 Nimble-generated Architecture

Fig 4.13 shows the RCU architecture implemented by Nimble when targeting a Virtex-based RCU: The computation is realized as a datapath of operators with widths matching the C primitive data types (8b,16b,32b).

The operation of this datapath is orchestrated by a controller (Figure 4.14) that, with its Petri-net like structure, allows the parallel execution of the datapath operators. However, it is limited in that it cannot handle variable latency operators (note the fixed delay between states 2 and 3). Thus, calculations cannot profit from shortcut evaluation computable much faster than the general case (such as multiplying by one or a power of two etc.). All variable-operand operators are always scheduled with their maximum latency.

In cases where variable latencies cannot be avoided, such as the cached master-mode interface to external memory (as in state 5), the *entire* controller is halted until the memory access has been completed. These accesses, which are crucial for implementing the C language in its array- and pointer-heavy glory, are performed via a memory system with 16KB of direct mapped write-through cache, organized as 32 lines of 32 32b words each. The tags are held in on-chip memory, while the cache lines are stored in an external SRAM bank. The entire unit takes three clock cycles to deliver data on a cache hit. The datapath accesses memory by connecting registers to the memory system. In Figure 4.13, Operators 6 and 9 can output a pointer address to memory. On the data side, Operator 7 can accept data read from memory, while Operator 10 can write data to memory.

The CPU communicates with the RCU by reading and writing memory-mapped I/O reg-

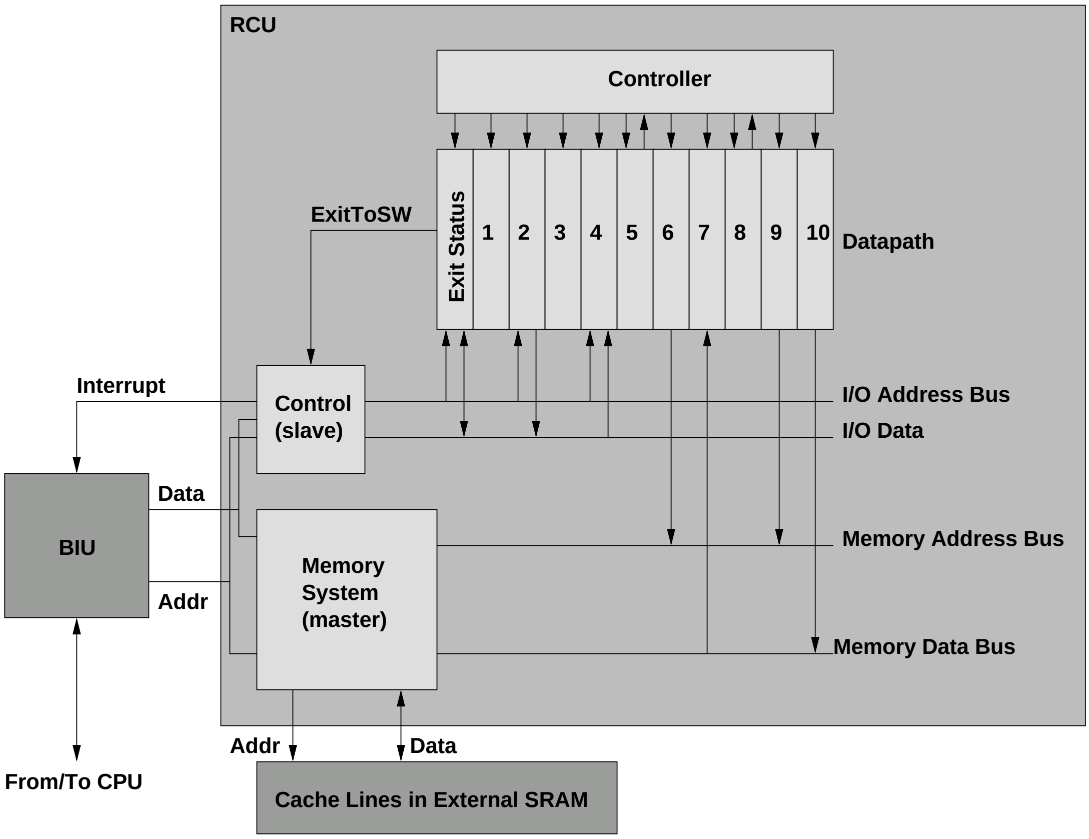


Figure 4.13: A sample RCU architecture generated by Nimble

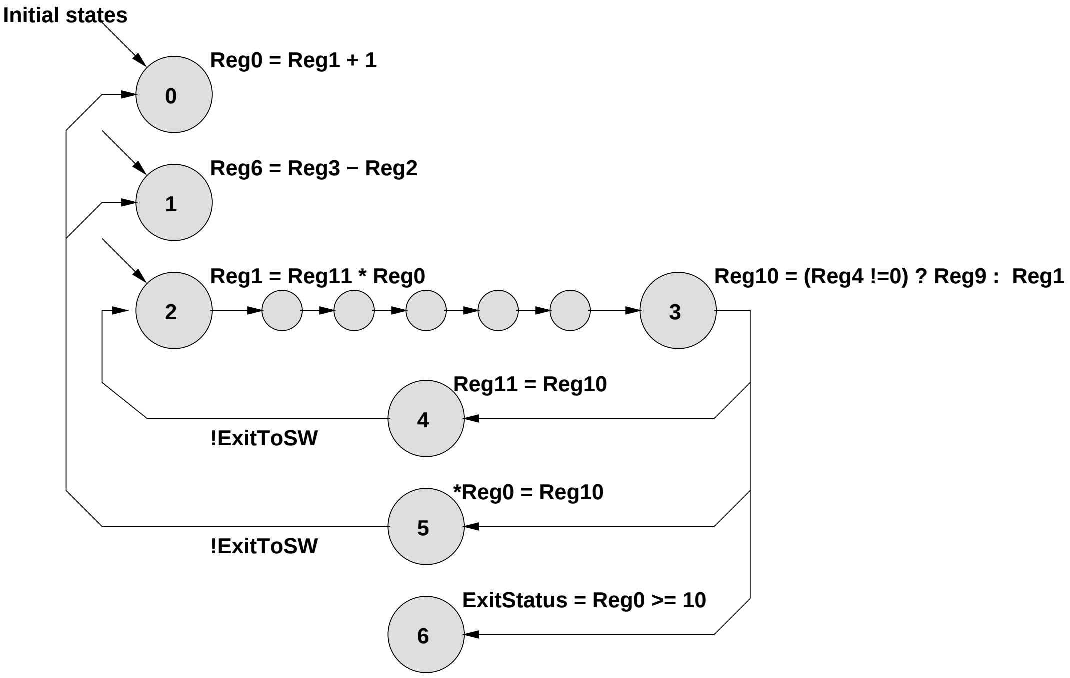


Figure 4.14: A sample Nimble-generated controller

isters within the datapath. These registers are connected not only inside of the datapath, but also to the chip-wide I/O address bus and (selectively) to the chip-wide external data input (for writable registers) and output busses (for readable registers). In Figure 4.13, Operator 2 holds a readable, Operator 4 a writable communication register.

As an extension to the I/O register functionality, a dedicated ExitStatus register sends an interrupt to the CPU when written to with a non-zero value from within the datapath. This register can then be read from the CPU to determine the cause of the interrupt (e.g., the specific hardware-software transition to execute, Section 4.4.1).

4.6.2 COMRADE Extensions

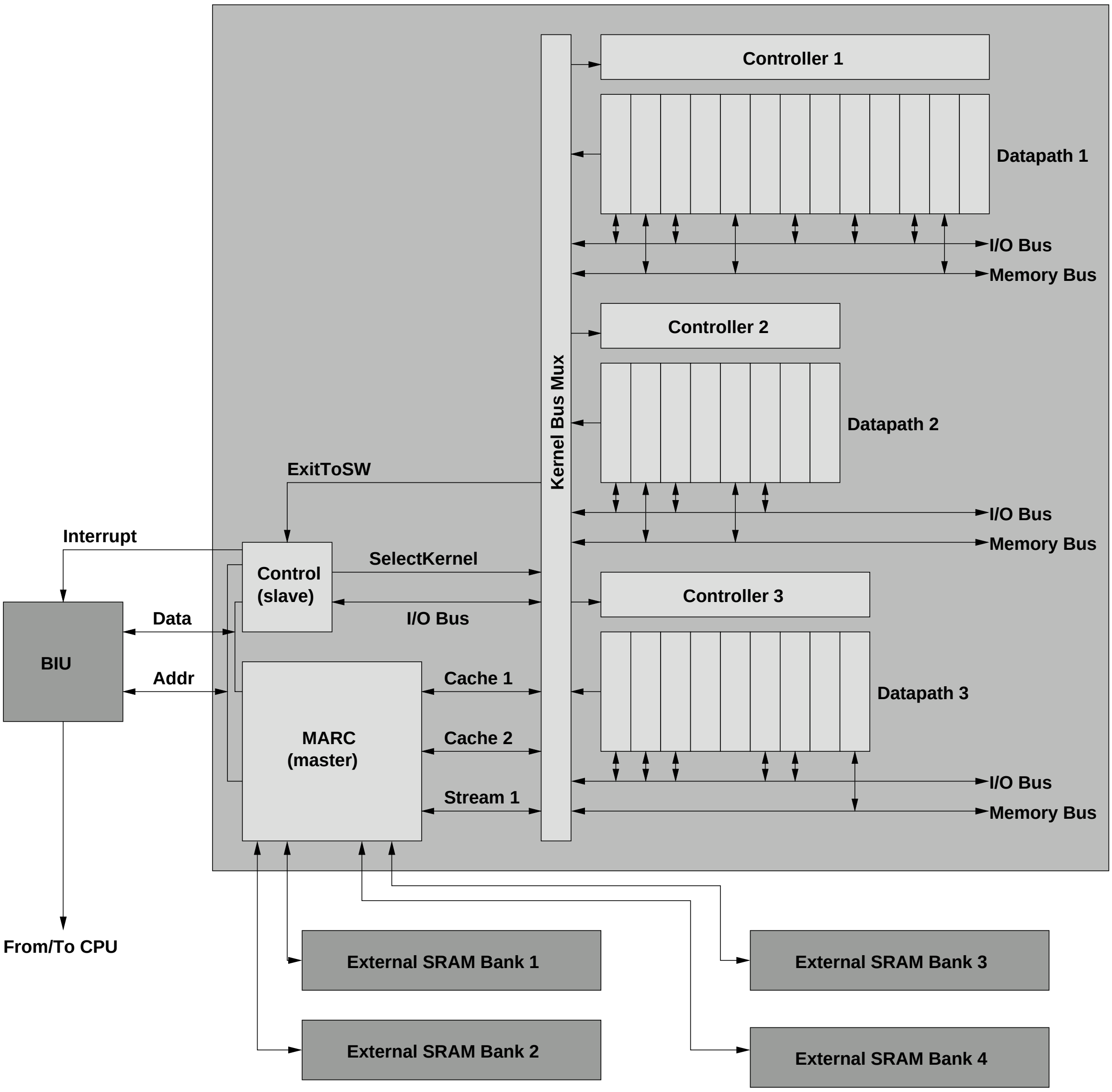


Figure 4.15: A sample RCU architecture generated by COMRADE

For COMRADE, this base architecture was considerably extended: First, the datapath is no longer realized as a single stripe across the RCU fabric. This placement scheme was due to the floorplanning tools used in Nimble, which in turn were derived from GarpCC (the Garp RCU consists only of a single stripe of 32b operators). In the new scheme, a single datapath may be realized as multiple rows on the target FPGA. New floorplanning tools are currently under development to fully automate this process. They take care to maintain the datapath regularity as well as to fully exploit device specifics (such as the direction of fast carry chains) for high-performance layouts. Independent kernels fused into a single configuration (see Section 4.5.1) are handled using the same approach.

The CPU - RCU communication is handled as in the Nimble architecture: The RCU acts as slave, accepting read and write requests from and to datapath registers from the CPU.

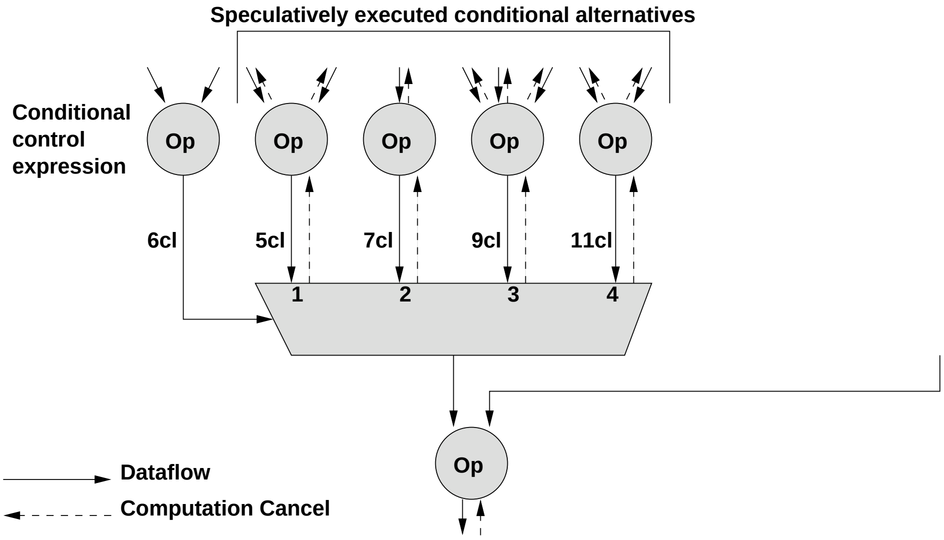

Speculatively executed conditional alternatives
Conditional control expression
Op
Op
Op
Op
Op
6cl
5cl
7cl
9cl
11cl
1
2
3
4
Op
Dataflow
Computation Cancel

Figure 4.16: Controller model used by COMRADE

The controller architecture has also been extended considerably in COMRADE. Figure 4.16 shows the computation structure supported by the new controller. Now, each operator (regardless of its actual hardware realization) is treated as a node in a data flow graph. Variable latency operators inside of the datapath and in the form of the memory interface are now fully supported. They are handled by generating dedicated logic for dynamic scheduling in the controller, keeping up the appearance of token-passing between dataflow operators. Other threads of control independent of the output of the variable latency operator are no longer stopped when such a calculation is in progress.

Additionally, the generated control logic now supports shortcut evaluation of conditionals: After the control expression has been evaluated, the total latency is only determined by the input corresponding to the control expression, other inputs are disregarded and do not delay the conditional. In the example, the value of the conditional control expression is known after 6 clock cycles. Assuming the value of 2 for the control expression, the output of the entire conditional block is known after 7 clock cycles, the slower operators on inputs 3 and 4 no longer delay the calculation. These shortcut evaluations are fully exploited by the dynamic scheduling in the controller.

For even higher efficiency, the controller can also cancel the calculations on the untaken branches, enabling those operators to accept the next input datum. In the previous example, the operator chains on inputs 3 and 4 could be advised that the result currently under computation is no longer required, and to proceed with processing the next input datum. However, since most operators in the hardware library (see Section 4.7.1) have relatively short latencies, the applicability of this capability is still limited. This could change when more complex multi-cycle operators such as predefined IP blocks are integrated into the datapath (see Section 4.7.2).

```
int f(int x) {
  int a, b, c, d, e;

  a = x & 0x3;
  b = (x & 0xf0) >> 4;
  c = (x & 0x80000000) != 0;
  d = (c << 6) | (a << 4) | b;
  e = d + 1;
...
}
```

Nimble		COMRADE	
Width [b]	Area [Slices]	Width [b]	Area [Slices]
32	16	2	0
32	16	4	0
32	16	1	0
32	16	7	0
32	16	8	4
Total	80		4

Figure 4.17: Bit-wise expression optimization in COMRADE

While an FPGA-based RCU has many practical disadvantages, in allows some optimizations that would not be possible on a device with coarser granularity: By matching the bit-width of operators precisely to the environment and allowing the exploitation of constants on a bit-wise basis, gains can be realized in multiple areas: The resulting operators can become both smaller and/or faster, and routing requirements can be reduced since constant values can be generated directly at the sink and do not have to routed between operators. Figure 4.17 shows an example how a bit-width reduction pass in COMRADE can both shrink operator sizes (as for the adder yielding `e`) as well as completely replace operators by wiring (as for `a` to `d`).

The bit-width reduction pass in COMRADE can actually perform even more complicated analyses: It can not only track the width of each expression, but also the constancy (0 or 1) of individual bits within that width. When combined with an appropriately powerful hardware library (an effort currently underway), this knowledge can be combined with partial evaluation (Section 3.4.3) to generate even smaller/faster operators. In its current state, the hardware library can already exploit non-standard operator widths (e.g., a 14b adder).

To illustrate the potential of fully exploiting per-bit analysis, Table 4.3 shows the results when applying the pass to a number of sample applications. In general, only 56-76% of the bits of the C variables (8b, 16b, 32b wide) are actually changing. The rest is either constant or not required in the rest of the program at all (dead).

Application	% live	% const	% dead
Mediabench adpcm	67.43	23.93	8.63
Honeywell decompress	76.85	20.65	2.49
Honeywell capacity	71.70	24.87	3.42
ec_field	56.22	26.06	17.71
Huffman	57.46	26.30	16.22

Table 4.3: Results of per-bit expression analysis

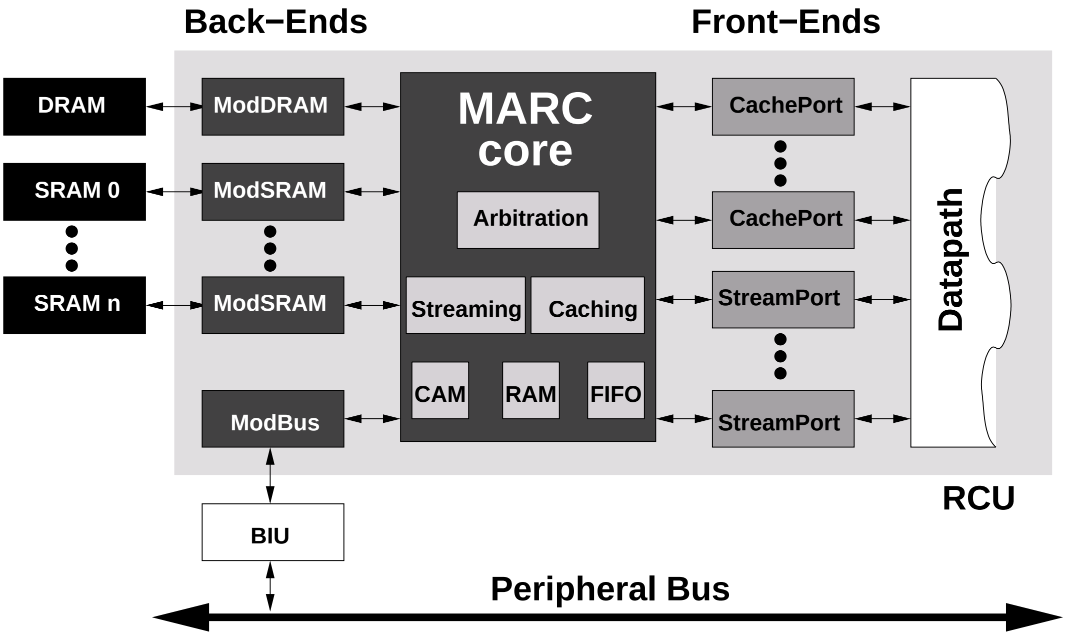


Figure 4.18: MARC architecture

4.6.3 Memory Interface

For access to all RCU-external memories, COMRADE targets the Memory Architecture for Reconfigurable Computers (MARC) [119], which has been developed from scratch. MARC, shown in Figure 4.18, is a parameterizable multi-port memory system that supports independent accesses in both streaming and random patterns. The two access mechanisms are internally backed by FIFO-based buffers and fully associative caches. For both, many characteristics such as cache organization, FIFO lengths and memory allocation are configurable and can be matched to the needs of the application.

At present, COMRADE only uses a limited set of MARC's capabilities: Since an induction variable analysis is currently not implemented, the compiler cannot exploit the efficient streaming support. Similarly, better support for loop parallelization that would allow full use of the multi-port parallel memory accesses in COMRADE is just being developed [120]. Even without these advanced features, COMRADE-generated datapaths still profit from MARC's increased performance (single cycle latency on cache hit) over the original Nimble memory interface.

4.7 Hardware Generation

Compiling for an ACS at some point requires dealing with real hardware (either on the RCU chip or at the system level), the compiler thus has to be provided with a suitably abstracted view of these aspects. The previous section described the RCU architectures generated by both the Nimble and COMRADE flows. In this section, the level of individual hardware operators is considered.

4.7.1 Module Generator Interface

As described in the previous section, for fine-grained RCUs based on FPGAs, the flexibility of matching the application's needs extends below the level of the RCU down to the realization of the individual operators. Ideally, a circuit perfectly fitted to current requirements such as operator widths, signedness, constant values on inputs (even on a per-bit basis) can be generated. This is achievable by using algorithmic descriptions of the circuit structure, so-called *module generators*. Both the Nimble and COMRADE flows formulate their hardware operators in Java using the JHDL library [121]. The resulting flexibility flexibility, however, complicates the the integration with the rest of the flow. To make informed trade-off decisions, e.g. during the CPU-RCU partitioning phase (Section 4.5), tools early in the compile flow need abstract, yet sufficiently precise measurements for quantities such as operator function, control interface, RCU area required and execution times. Given the flexibility of the generators, this information cannot be represented as a static table enumerating all possibilities (as is often done for ASIC cells in classical VLSI design flows). Furthermore, the same generators are later used to create actual circuits in later phases of the compile flow (ideally in an already optimally placed layout). These later design implementation tools have different data requirements: Now, logical aspects such as the behavior and interfaces of an operator are no longer relevant. Instead, physical characteristics such as layout topology, port placement and netlist formats become important.

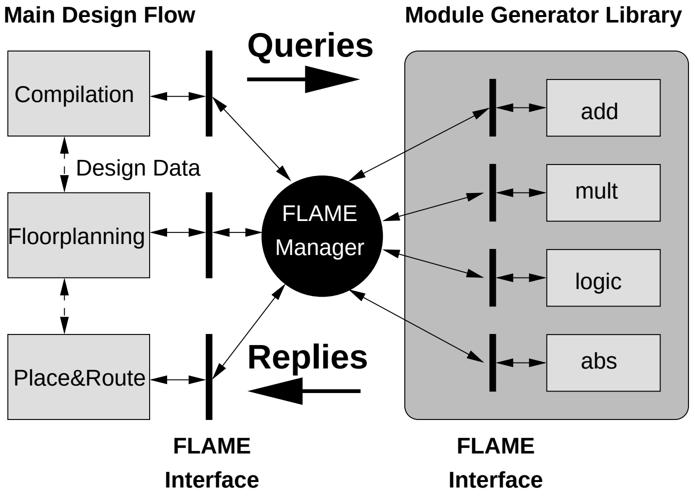


Figure 4.19: FLAME-based tool flow

As an encompassing solution, we developed the Flexible API for Module-based Environments (FLAME) [122] [123] [124] [125] [126] [127], shown in Figure 4.19. FLAME provides an active API that lets different data requesters (called *clients*) retrieve from the hardware library exactly the data they require in the form of specific *views*. These include, for example, a behavioral view that describes the function of an operator and a topology view that specifies layout characteristics. Together, all of these views form a comprehensive design data model that is currently mapped to more than 140 classes in UML representation [128]. While the reference implementation of FLAME was realized in Java, the UML model could be easily exported to other languages such as C++ for a different environment. Figure 4.20 shows an excerpt from the design data model describing target-device dependent module port attributes.

A subset of FLAME was already used successfully in Nimble. COMRADE completely relies on FLAME for all handling all hardware-related aspects of its operation, including kernel selection, control interface synthesis and operator hardware generation [115].

The collection of module generators targeted by COMRADE is the Generic Library for Adaptive Computing Environments (GLACE) [129], the first implementation of a hardware library compliant with the FLAME Library Specification [125] [126] [130]. It contains hardware implementations for all C operators, a sample of which is shown in Table 4.4.

In its current state of refinement, FLAME has proven extremely useful for isolating the compile flow from the intricacies of hardware generation. It allows the seamless access to circuit generators considerably more powerful than those developed in the predecessor system PARAMOG [131].

For the function library GLACE, a new version currently under development aims to improve both the usability as well as the efficiency of the generated modules. The first goal will be achieved by automatically extracting circuit characteristics such as logic delays

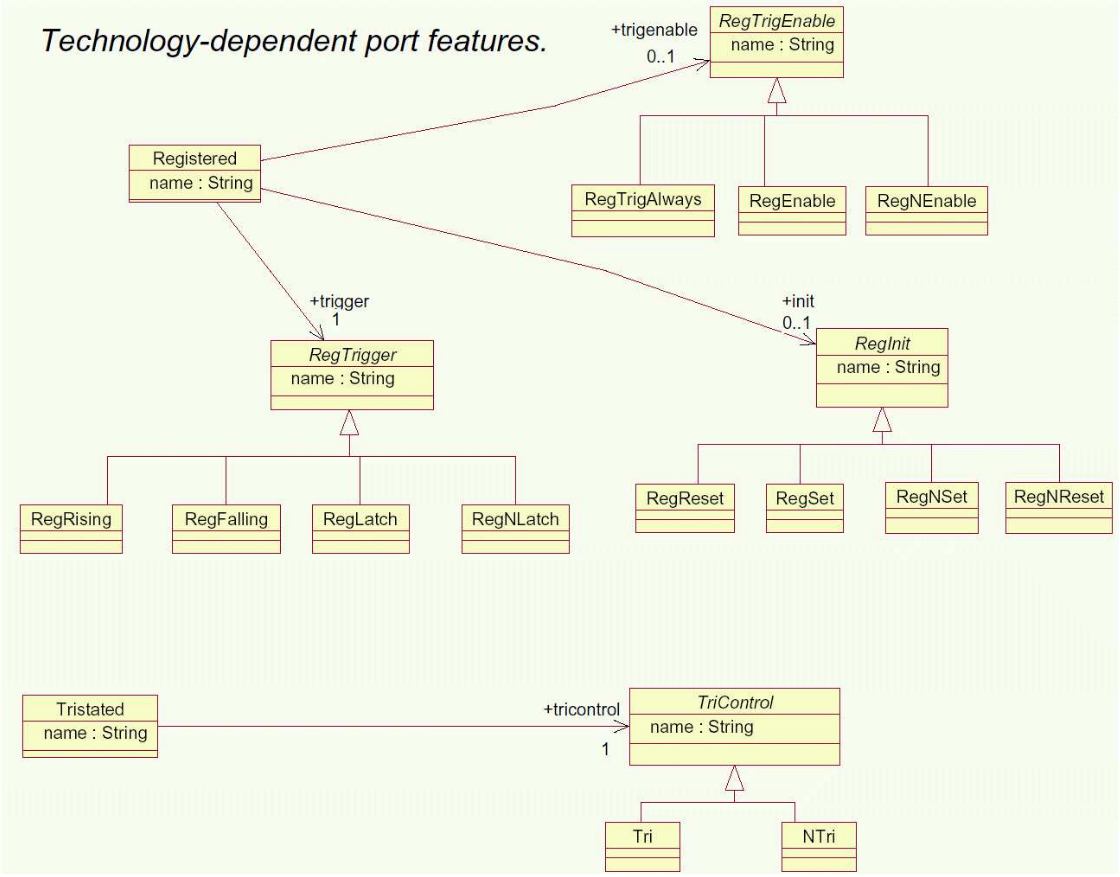


Figure 4.20: Excerpt from FLAME design data model

Description	Behavior Name and Logical Interface
Addition	`add(sum, [cout,] [ovfl,] a, b [, cin])`
Subtraction	`sub(diff, [bout,] [ovfl,] a, b [, bin])`
Multiplication	`mul(prod, [ovfl, ] a, b)`
Division	`div(quot, [zerodiv,] a, b)`
Modulus	`mod(rem, [zerodiv,] a, b)`
Negation	`neg(neg, [cout,] a [,cin])`
Absolute	`abs(abs, [cout,] a)`
Logical Shift Left	`lsl(lsl, din, bits)`
Logical Shift Right	`lsr(lsr, din, bits)`
Arithmetical Shift Right	`asr(asr, din, bits)`
Less-Than	`lt(lt, a, b)`
Less-Than-Equal	`le(le, a, b)`
Equal	`eq(eq, a, b)`
Not-Equal	`ne(ne, a, b)`
Greater-Than-Equal	`ge(ge, a, b)`
Greater-Than	`gt(gt, a, b)`
Logic	`logic(y, a, b, c, d, [e,] [f,] [g,]  ttable)`
Multiplexing	`mux(mux, a, b, [c,] [d,] [e,]  ..., sel)`
Register	`reg(q, d [,clk, en] [,lt])`

Table 4.4: Sample FLAME behavior names and interfaces

and placement topologies from the Java descriptions [121]. The second one consists of integrating logic synthesis and optimization capabilities [132] right into the module generators. This is different from our previous approach in SDI [133] [131], which centralized all logic optimization stages in the floorplanning step. The new approach enables better control of the optimization process, since each generator can now utilize the general optimization services in the best-fitting fashion. This intra-module logic synthesis also forms the basis of the exploitation of constant values on a per-bit basis (Section 4.6.2).

4.7.2 IP Integration

Despite of our best efforts to the contrary, we are fully aware that even a highly optimizing compiler will not be able to match the efficiency (area, performance, power, etc.) of a circuit handcrafted by an experienced hardware designer. In the software world, a similar situation has long existed with compiled code and handwritten assembler code. However, there, well defined application binary interfaces (ABI) exists that allow the seamless cross-language calling between both kinds of executable code. For applications fields as diverse as computer graphics and numerical algorithms, libraries of hand-optimized assembler-coded functions can easily be called from high-level languages such as C, Pascal, Java, etc.

In the hardware world, the situation is more difficult. No ABI-like standards exist for hardware functions. Furthermore, many current attempts to standardize the interfaces of hardware blocks, often called *intellectual property* (IP), are aiming at the more loosely coupled usage in context of a *system-on-chip* (SoC). The busses and protocols common for these applications (AMBA, CoreConnect, etc.) are much more heavyweight (split transactions, multiple transfer sizes, bursts, cache coherency, etc.) than the direct register-register transfers that would be used between datapath operators.

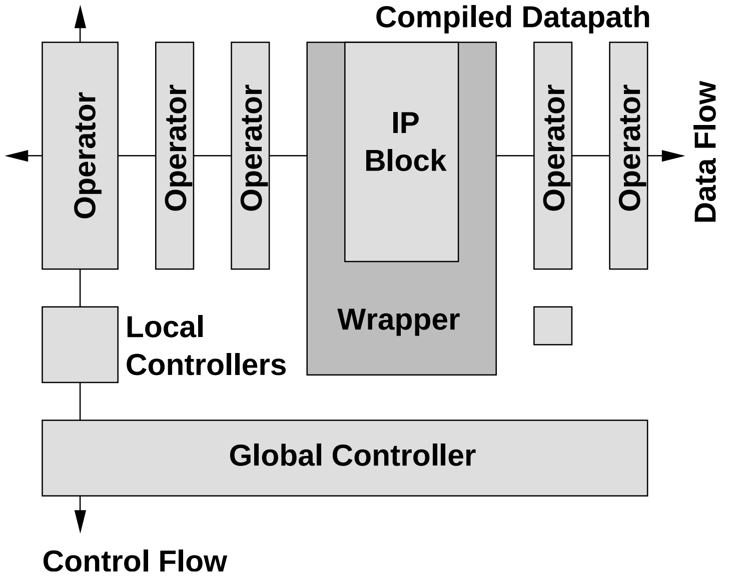


Figure 4.21: Tight integration of IP block into compiled datapath

However, the tight integration of handcrafted operators into a compiled datapath can significantly increase performance (Figure 4.21). Nimble had experimental capabilities to map designated functions with one or two arguments, and having a single result and no side-effects, to handcrafted hardware blocks supplied to the compiler as netlists. Communication was performed using direct register-register transfers inside of the datapath.

For an algorithm such as DES [134], the realization of the operations SBox, bit permutation, and parity counter as custom datapath operators resulted in a speed-up factor of over 400 compared to the version generated by compiling the C description of these functions. The embedding of larger IP blocks, e.g., realizing a complete FFT or even a complete MPEG-2 decoder, could lead to even greater speedups.

However, these blocks have both more complex interfaces as well as greater infrastructure demands. For example, the FFT block might need to accept 16 coefficients as a data stream, execute for a variable number of cycles, and then simultaneously unload the results and accept the next set of coefficients. The MPEG-2 decoder might require access to a dedicated 2M x 32b memory bank for use as scratch pad and frame buffer.

The UCODE primitives in FLAME [127] already allow the description of pipelined interface protocols such as those in the FFT example shown above. From the description in Figure 4.22 an appropriate interface controller for interfacing with the datapath controller (Section 4.6) can be efficiently synthesized in different styles, allowing a trade-off between RCU area and performance (Table 4.5).

```
; initialize
POSEDGE (CE 1) (SCALE_MODE 0)
        (FWD_INV 1) (START 1)
POSEDGE (START 0)
; start of steady-state
START
; wait for acceptance of first FFT block
CONTINUE (MODE_CE 1)
; write 16 time domain samples
POSEDGE *16 (DI_R[15:0] time_r[15:0])
            (DI_I[15:0] time_i[15:0])
; fork control flow for pipelining
RESTART
; wait for transformed data
CONTINUE (DONE 1)
; read 16 frequency domain samples
POSEDGE *16 (XK_R[15:0] freq_r[15:0])
            (XK_I[15:0] freq_i[15:0])
```

Figure 4.22: UCODE for wrapping 16-point FFT

Synthesis Style	Virtex-II Slices	Max. Clock [MHz]
Direct	25	467
Counter	13	248
SRL16	8	243

Table 4.5: Results of template-based synthesis

However, additional rules are required to fully integrate more complex IPs using an automatic compile flow. The FLAME Shared Access conventions [125] [126] discuss how a variety of light-weight communication protocols can be described using a simple bus architecture taxonomy.

Both of these approaches form the base of a complete solution. The Parametric C Interface for IP Cores (PaCIFIC) [135] integrates them with CoMAP, a previously developed system for the configuration management of parametrized IP blocks in an SoC context [136]. The result is an extension of the compile flow that extracts IP block instantiations from idiomatic C function calls. The hardware-software interfaces are then generated automatically from the interface description of the IP block. This description is formulated in a superset of the FLAME UCODE notation, allowing, e.g., for the transfer of complex data structures and exception handling.

The approach has been validated using manual prototyping. Figure 4.24 shows an example of a sample C program performing an FFT that has the call to the vfft16 function transparently mapped to an IP block running on the RCU. All hardware-software interfaces and module generator parameters can be deduced from the CoMAP database used internally in PaCIFIC. In contrast to the capabilities in Nimble, PaCIFIC allows the embedded IP blocks full access to the RCU environment. In the example, the MARC memory interface is used to run two data streams over the dram_in and dram_out arrays. Currently, COMRADE is being extended to automatically perform the PaCIFIC transfor-

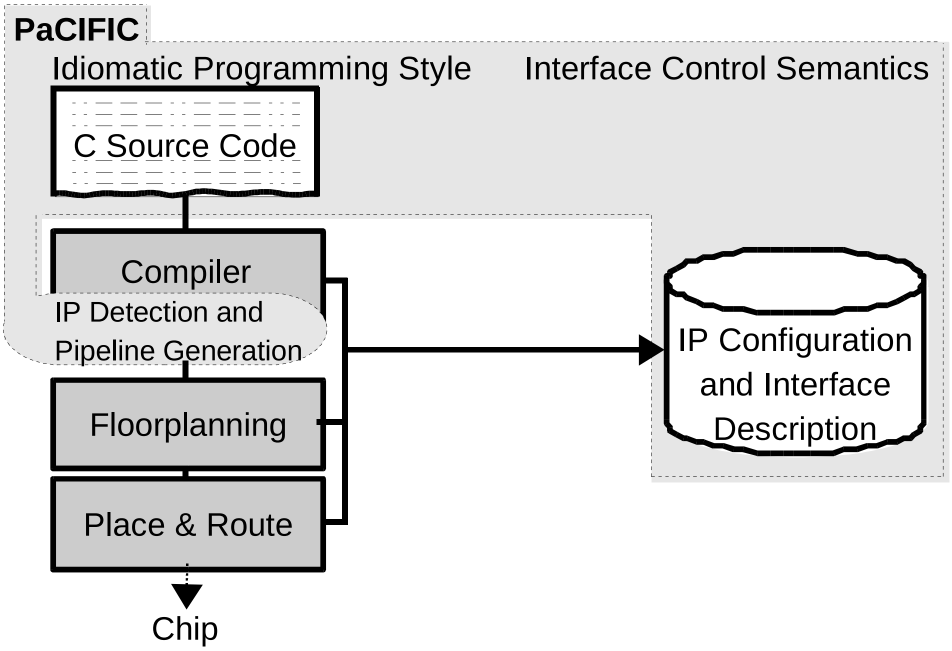


Figure 4.23: Integrating PaCIFIC into the compile flow

```
int main(int argc, char* argv[]) {
  FILE* infile, * outfile;
  int* dram_in, * dram_out;

  infile   = fopen("time.dat", "r");
  outfile  = fopen("freqspec.dat", "w");
  dram_in  = calloc(16384, sizeof(int));
  dram_out = calloc(16384, sizeof(int));
  fread (dram_in, 4, 16384, infile);
  vfft16(dram_in,  dram_out); /* HW call */
  fwrite(dram_out, 4, 16384, outfile);
...
}
```

Figure 4.24: Example for PaCIFIC-compliant source code

mations on the input source code.

5 Conclusions and Future Work

As we have detailed in the beginning, the tremendous progress in microelectronics no longer linearly translates into increased computer performance. While we can build chips with billions of transistors, the processor architectures implemented on them still follow patterns developed in the mid-1940s for vacuum-tube based systems. These patterns were incredibly successful for 60 years, but their applicability has begun to diminish now.

As an alternative, we have suggested to use the silicon real estate to implement processors following a different paradigm: Instead of temporally distributing a computation on shared compute units, it is distributed spatially across dedicated compute units. The latter approach leads to higher efficiency due to increased parallelism and reduced administrative overhead. However, in order for spatially distributed processors to remain as universal as their traditional brethren, they require the underlying device to be reconfigurable. Only in this fashion can different algorithms be executed on the same hardware.

Conventional processors and reconfigurable compute units complement each other well: The computation-intense kernels of an application can be spatially mapped to the reconfigurable unit, while the less critical or unsuitable parts remain on the CPU. Together, both processors form an adaptive computing system. However, the way the two units are coupled with each other and the rest of system must be matched to the data-transfer characteristics of the target applications. Mismatches here can lead to significant slowdown compared to a conventional computer due to excessive communications overhead.

Practical experiments have shown the advantages of adaptive computing systems for a variety of application fields: Very regular signal processing algorithms can profit as well as heavily control dominated applications. Gains can be achieved both in terms of compute performance as well as reduced power consumption. This outlook is even more promising when considering that most of the reconfigurable devices in use today have not been designed with compute applications in mind. As our experience with designing reconfigurable compute fabrics grows, so will the potential gains over standard processors.

What is lacking, however, are abstractions and software tools to make the potential of adaptive computers accessible to applications programmers. While specialized languages allow the efficient programming of reconfigurable compute units even today, they either require skills not generally available to today's software developer (e.g., experience in hardware design), or are limited to a specific application domain (e.g., digital signal processing).

The complexity involved in developing an automatic compiler targeting adaptive computers is significant: In addition to conventional compiler technologies, issues of automatic hardware synthesis both at the architectural and logic level also have to be considered now. Further intricacy is involved in partitioning the application between the conventional and the reconfigurable processor, and generating communications interfaces both in hard- and software. A completely novel topic even in hardware design is the full exploitation of reconfigurability, which in itself can occur in various forms (e.g., static and dynamic, complete or partial, etc.).

Our most current compile flow, COMRADE, is a third generation system. While built on the experiences with its predecessors, GarpCC and Nimble, it has made significant advances in a variety of areas. They range from novel internal representations suitable both for software and hardware optimization to improved architecture synthesis for the reconfigurable compute unit. The hardware library has become completely dynamic, replacing static data files and lookup tables with active algorithmic descriptions that can fully express even complex spatially distributed computing structures.

However, many worthwhile aspects have not been considered yet. Future work spans topics as diverse as improved automatic parallelization, memory localization, and more precise program analysis for better trade-off decisions. On the hardware side, promising topics include the automatic integration of even complex existing IP blocks, more powerful module generators creating even better circuits for operators, and specialized floorplanning tools that map all components of a reconfigurable compute unit to the underlying device with great efficiency.

As in the reconfigurable computing device architectures, software flows targeting adaptive computers are just in their infancy. In contrast to conventional computers, where even small gains are ever harder to realize, the entire field of reconfigurable computing is still a largely undiscovered country, rich with potential for unlocking the von Neumann shackles that have begun to restrain computer architecture innovation.

Bibliography

[1] Weik M.H., "The ENIAC Story", *J. of the American Ordnance Association*, January/February 1961 1

[2] Jones S.W., "Exponential Trends in the Integrated Circuit Industry", `http://www.icknowledge.com/trends/Exponential2.pdf`, March 28, 2004 (document), (document), 1.1, 1.2

[3] International Technology Roadmap for Semiconductors, "Executive Summary 2003", `http://public.itrs.net/Files/2003ITRS/ExecSum2003.pdf`, 2003 1

[4] Moore G.E., "Cramming more components onto integrated circuits", *Electronics*, Vol. 38, No. 8, 1965 1.1

[5] Yang D.J., "On Moore's Law and Fishing", *US News & World Report*, July 10, 2000 1.1

[6] IC Knowledge LLC., "Can the semiconductor industry afford the cost of new Fabs?", `http://www.icknowledge.com/economics/fab_costs.html`, 2004 1.2

[7] McCrory J., "Moore Versus Murphy: How 90 nm Affects Innovation", `http://www.elecdesign.com/Articles/Index.cfm?AD=1&ArticleID=7910`, May 10, 2004 1.2, 1.2, 1.2

[8] Intel Corp., "Discover the Technology Behind the Mask", `http://www.intel.com/labs/features/si02031.htm`, 2004 1.2

[9] Rabaey J., "Ultra Deep-Submicron Design Challenges", *ASPDAC 2000 Tutorial*, `http://bwrc.eecs.berkeley.edu/People/Faculty/jan/presentations/dsm.pdf`, 2000 1.2

[10] IC Knowledge LLC, "How do reports of lower foundry yields at 130 nm translate into defect density", `http://www.icknowledge.com/economics/yields.html`, October 9, 2002 1.2

[11] Alfke P., "Supply Voltage Migration", *Xilinx Application Note 080*, September 7, 1997 (document), 1.3

[12] Doyle B. et al, "Transistor Elements for 30 nm Physical Gate Lengths and Beyond", *Intel Technology Journal*, Vol. 06, No. 02, May 16, 2002 (document), 1.4, 1.2

[13] De V., Borkar S., "Technology and Design Challenges for Low Power and High Performance", *Proc. Intl. Symp. on Low Power Electronic Design*, San Diego (CA, USA), 1999 (document), 1.5, 1.2

[14] Borodovsky Y. et al., "Lithography Strategy for 65nm Node", *Proc. Soc. of Photo-Optical Instrumentation Engrs.*, Vol. 4754, 2002 1.2

[15] Bjorkholm E., "EUV Lithography - The Successor to Optical Lithography", *Intel Technology Journal Q3 1998*, 1998 1.2

[16] Maurer W., "Application of Advanced Phase-Shift Masks", `http://www.mentor.com/products/ic_nanometer_design/news/phase-shift_masks.cfm`, September 1, 2003 1.2

[17] Ghani T. et al., "A 90 nm High Volume Manufacturing Logic Technology Featuring Novel 45nm Gate Length Strained Silicon CMOS Transistors, *Proc. International Electron Devices Meeting*, Washington (DC, USA), December 9, 2002 1.2

[18] Thompson S. et al., "130 nm Logic Technology Featuring 60nm Transistors, Low-K Dielectrics, and Cu Interconnects", *Intel Technology Journal*, Vol. 06, No. 02, May 16, 2002 1.2

[19] Miller M., "Manufacturing-aware design helps boost IC yield", *EEdesign*, September 9, 2004 1.2

[20] Hennessy J., Patterson D., "Computer Architecture: A Quantitative Approach", *Morgan Kaufmann Pub.*, 2002 1.3, 1.3, 1.3, 1.3, 1.3, 2.4.5

[21] Boggs D. et al., "The Microarchitecture of the Intel Pentium 4 Processor on 90 nm Technology", *Intel Technology Journal*, Vol. 08, No. 01, February 18, 2004 1.3

[22] de Vries H., "Looking at Intel's Prescott die", `http://www.chip-architect.com/news/2003_03_06_Looking_at_Intels_Prescott.html`, March 6, 2003 (document), 1.3, 1.6, 2.6

[23] Wawrzynek J., "Reconfigurable Computing", `http://inst.eecs.berkeley.edu/~cs294-3/lectures/intro.pdf`, January 21, 2004 (document), 1.7

[24] Intel Corp., "Intel Silicon Innovation To Shape Direction Of The Digital World", *Intel Developer Forum*, San Francisco (CA, USA), September 7, 2004 1.3

[25] Von Neumann J., "First Draft on a Report on the EDVAC", Contract No. W-670-ORD-492, Moore School of Electrical Engineering, Univ. of Penn., Philadelphia. Reprinted (in part) in *Randell, Brian. 1982. Origins of Digital Computers: Selected Papers, Springer-Verlag, Berlin Heidelberg, pp. 383-392*, 1945 1.3

[26] Berkeley Design Technology Inc., "The BDTImark 2000: A Summary Measure of DSP Speed", `http://www.bdti.com/bdtimark/BDTImark2000.pdf`, February 2003 1.3

[27] Berkeley Design Technology Inc., "BDTImark2000 Scores for Floating-Point Packaged Processors", `http://www.bdti.com/bdtimark/chip_float_scores.pdf`, November 2004 1.3

[28] Analog Devices Inc., " ADSP-TS201S: TigerSHARC Embedded Processor, 500/600 MHz, 24 Mbits, Preliminary Data Sheet", `http://www.analog.com/UploadedFiles/Data_Sheets/42292566465477642797139945514ADSP-TS201S_prh.pdf`, 2003 (document), 1.8

[29] NVidia Corp., "GeForce 6800 Ultra Specifications/Performance", `http://www.nvidia.com/page/geforce_6800.html`, November 2004 1.3

[30] Advanced Micro Devices Inc., "AMD Opteron Processor Key Architectural Features", `http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8826_8805,00.html`, November 2004 1.3

[31] Fan Z. et al., "GPU Cluster for High Performance Computing", *Proc. ACM/IEEE Supercomputing Conf.*, Pittsburgh (PA, USA), November 2004 1.3

[32] Bajaj C. et al., "SIMD Optimization of Linear Expressions for Programmable Graphics Hardware", *Computer Graphics Forum*, Vol. 23, No. 4, December 2004 1.3

[33] Analog Devices Inc., "TigerSHARC Processor Benchmarks", `http://www.analog.com/processors/processors/tigersharc/benchmarks.html`, November 2004 1.3

[34] Verhulst E., "Programmable dataflow architecture for signal processing", *Embedded Control Europe*, March 2004 1.3

[35] Borgatti M., Lertora F., et al."A reconfigurable system featuring dynamically extensible embedded microprocessor, FPGA, and customizable I/O". *IEEE J. of Solid-State Circuits*, Vol. 38, No. 3, March 2003 2.3.2

[36] eASIC Corp., "FlexASIC Product Brief", `http://www.easic.com/technolgy/index.html`, November 2004 2.3.2

[37] Trimberger S., *personal communications*, February 1999 2.3.2

[38] Bindra A., "Reconfigurable Architectures Chart A New Course For DSPs", *Electronic Design*, `http://www.elecdesign.com/Articles/Index.cfm?AD=1&ArticleID=2596`, August 5, 2002 2.3.2

[39] Koch A., "Adaptive Rechensysteme - Architekturen und Werkzeuge", *Proc. E.I.S. Workshop*, Darmstadt (Germany), September 1999 2.4, 4

[40] Koch A. , "Architectures and Tools for Heterogeneous Reconfigurable Systems", *Proc. IEEE Workshop on Heterogeneous Reconfigurable Systems-on-Chip*, Hamburg (Germany), April 2002 2.4, 2.4.4

[41] Compton K., Hauck S., "Reconfigurable Computing: A Survey of Systems and Software", *ACM Computing Surveys*, Vol. 34, No. 2, June 2002 2.2, 2.4, 4

[42] Tessier R., Burleson W., "Reconfigurable Computing for Digital Signal Processing: A Survey", *J. of VLSI Signal Processing*, Vol. 28, No. 1-2, May-June 2001 2.2

[43] Cadence Design Systems Inc., "Palladium", *data sheet*, 2004 2.4.1

[44] Intel Corp., "Intel Pentium 4 Processors 560, 550, 540, 530 and 520", *data sheet*, June 2004 2.4.3

[45] Cray Inc., "Cray XD1". *data sheet*, 2004 2.4.3

[46] Callahan T., Hauser J., Wawrzynek J., "The Garp Architecture and C Compiler", *IEEE Computer*, Vol. 33, No. 4, April 2000 2.4.4, 2.6, 2.7, 2.8, 2.9, 4.1, 4.2, 4.6

[47] Intrinsity Inc., "Intrinsity Processor User's Manual Version 0.2", *user manual*, 2003 2.4.4, 2.4.4

[48] Byatt M., "Data Plane Processing with Configurable Architectures", *ARM white paper*, November 2003 2.4.4, 2.4.4

[49] Mepham D., "The next Pentium 4 processor, Prescott arrives", `http://www.hardwareanalysis.com/content/article/1686/`, February 2, 2004 2.4.4

[50] Xilinx Inc., "Virtex-4 Family Overview", *advance product specification*, September 10, 2004 2.4.4

[51] ARM Ltd., "ARM7TDMI", `http://www.arm.com/products/CPUs/ARM7TDMI.html`, November 2004 2.4.4

[52] Razdan R., Smith M.D., "A High-Performance Microarchitecture with Hardware-Programmable Functional Units", *Proc. MICRO-27*, San Jose (CA, USA), November 1994 2.4.5, 2.4.5, 2.6, 2.7, 2.8

[53] Esparza J.E.C., "Evaluation of the OneChip Reconfigurable Processor", *master's thesis*, U Toronto (CA), 2000 2.4.5, 2.4.5, 2.4.5, 2.6

[54] Stretch Inc., "S5000 Software Configurable Processors", *data sheet*, 2004 2.4.5, 2.4.5, 2.6, 2.6, 3.4.4

[55] Jacob J.A., "Memory Interfacing for the OneChip Reconfigurable Processor", *master's thesis*, U Toronto (CA), 1998 2.4.5, 2.8

[56] Tensilica Inc., "Xtensa Microprocessor", *data book*, August 2002 2.4.5

[57] Embedded Microprocessor Benchmark Consortium, "EEMBC Telemark Benchmark Scores", `http://www.eembc.hotdesk.com/eembc-telecombenchmarkscores.html`, March 19, 2004 2.4.5

[58] Gokhale M. et al., "Building and Using a Highly Parallel Programmable Logic Array", *IEEE Computer*, Vol. 24, No. 1, January 1991 2.2, 2.5, 4

[59] Annapolis Micro Systems Inc., "WILDSTAR-II Pro for PCI Rev 1.0", *data sheet*, 2003 (document), 2.5, 2.7

[60] M2000, "The FlexEOS product", http://www.m2000.fr/products.htm, November 2004 2.6

[61] Elixent Ltd., "Elixent DFA-1000 Products", http://www.elixent.com/products/ip_products.htm, November 2004 2.6

[62] PACT XPP Technologies AG, "XPP Intellectual Property cores", http://www.xppip.com/xneu/t_ipcor.html, November 2004 2.6, 4.6

[63] ARM Ltd., "Embedded Cores", http://www.arm.com/products/CPUs/embedded.html, November 2004 2.6

[64] Tensilica Inc., "Xtensa Product Overview", http://www.tensilica.com/html/products.html, November 2004 2.6

[65] ARC Inc., "Configurable Cores", http://www.arc.com/configurablecores/, November 2004 2.6

[66] MIPS Technologies Inc., "MIPS Cores", http://www.mips.com/content/Products/Cores/32-BitCores, November 2004 2.6

[67] You C. et al., "A 5-20 GHz, Low Power FPGA Implemented by SiGe HBT BiCMOS Technology", *Proc. Great Lakes Symp. on VLSI*, Washington (DC, USA), April 2003 2.6

[68] Zhang H. et al., "A 1V Heterogeneous Reconfigurable Processor IC for Baseband Wireless Applications", *Proc. Intl. Solid State Circuits Conf. (ISSCC)*, San Francisco (CA, USA), 2000 2.6, 3

[69] Plunkett B., Watson J., "Adapt2400 ACM Architecture Overview", *QuickSilver Technology Inc. whitepaper*, November 2004 2.6

[70] Koch A., "A Comprehensive Prototyping Platform for Hardware-Software Codesign", *Proc. Workshop on Rapid Systems Prototyping*, Paris (F), June 2000 2.8, 2.8, 2.9, 3.1.2

[71] Koch A., Golze U., "A Universal Co-processor for Workstations", in *More FPGAs*, Abbingdon EE&CS Books, Oxford (UK), 1994 2.9

[72] Koch A., Golze U., "Practical Experiences with the SPARXIL Co-Processor", *Proc. 31st Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove (CA, USA), November 1997 2.9, 3.2.2

[73] Arnold J., Buell D., Davis E., "Splash 2", *Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1992 2.2, 2.9, 4

[74] Rupp C. et al., "The NAPA Adaptive Processing Architecture", *Proc. of the IEEE Symp. on FPGAs for Custom Computing Machines (FCCM)*, Napa (CA, USA), 1998 2.9

[75] OAR Corp., "Realtime Operating System for Multiprocessor Systems", `http://www.rtems.org`, November 2004 2.9

[76] Hoan, D.T., "Searching Genetic Databases on Splash 2", *Proc. IEEE Symposium on Field-Configurable Computing Machines*, 1993 3

[77] Yu C.W. et al., "A Smith-Waterman Systolic Cell", *Proc. Intl. Conf. on Field-Programmable Logic (FPL)*, Lisbon (PT), September 2003 3

[78] Shand M., Vuillemin J., "Fast Implementation of RSA Cryptography", *Proc. 11th IEEE Symposium on Computer Arithmetic*, 1993 3

[79] Lipmaa H., "AES Ciphers: speed", `http://www.tcs.hut.fi/~helger/aes/rijndael.html`, November 2004 3

[80] Jarvinen K.U. et al., "A Fully Pipelined Memoryless 17.8 Gbps AES-128 Encryptor", *Intl. Symp. on Field-Programmable Gate Arrays (FPGA)*, Monterey (CA, USA), February 2003 3

[81] Rabaey J., "Pleiades: Ultra Low Power Hybrid and Reconfigurable Computing", *Presentation at Berkeley Wireless Research Center Retreat*, June 1999 3

[82] Antonini M., Barlaud M., Mathieu P., Daubechies I.,"Image Coding using the Wavelet Transform", *IEEE Transactions on Image Processing*, No. 1, 1992 3.1.1

[83] Gädke H., Koch A., "Wavelet-based Image Compression on the Reconfigurable Computer ACE-V", *Proc. Intl. Conf. on Field-Programmable Logic (FPL)*, Antwerp (BE), September 2004 3.1.2, 4.4.4

[84] Schmidt C., Koch A., "Fast Region Labeling on the Reconfigurable Platform ACE-V", *Proc. Intl. Conf. on Field-Programmable Logic (FPL)*, Lisbon (PT), September 2003 3.2.2

[85] Ray T.S., Hart J.F., "Evolution of Differentiation in Multithreaded Digital Organisms", *Proc. 7th Intl. Conf. on Artificial Life*, 2000 3.3.1

[86] Böge M., Koch A., "A Processor for Artificial Life Simulation", *Proc. Intl. Workshop on Field-Programmable Logic (FPL)*, Glasgow (UK), September 1999 3.3.2

[87] Koren I., "Computer Arithmetic Algorithms", *A K Peters Ltd.*, 2002 3.4.1

[88] Volder J.E., "The CORDIC Trigonometric Computing Technique", *IRE Transactions on Electronic Computers*, Vol. EC-8, No. 3, September 1959 3.4.1

[89] Uytterhoeven G., Roose D., Bultheel A., "Wavelet Transforms using the Lifting Scheme", *Technical Report ITA-Wavelets Report WP 1.1, Katholieke Universiteit Leuven, Department of Computer Science*, Belgium, 1997 3.4.1

[90] The MathWorks Inc., `http://www.mathworks.com` 3.4.2

[91] Meyer-Bäse U., “Digital Signal Processing with Field Programmable Gate Arrays”, *Springer*, Berlin, April 2004 3.4.2

[92] Andraka R., “FPGAs Make a Radar Signal Processor on a Chip a Reality”, *Proc. Asilomar Conf. on Signals, Systems, and Computers*, Monterey (CA, USA), October 1999 3.4.3

[93] Kitagawa K. et al., “A Hardware Overview of the SX-6 and SX-7 Supercomputer”, *NEC Res. and Develop.*, Vol. 44, No. 1, January 2003 3.4.4

[94] Camposano R., Wolf W., “High Level VLSI Synthesis”, *Kluwer Academic*, 1991 4, 4.2

[95] Lin Y.-L., “Recent Developments in High-Level Synthesis”, *ACM Trans. on Design Automation of Electronic Systems*, Vol. 2, No. 1, January 1997 4, 4.2

[96] Bacon D.F., Graham S.L., Sharp O.J., “Compiler Transformation for High-Performance Computing”, *ACM Computing Surveys*, Vol. 26, No. 4, December 1994 4, 4.1, 4.2, 4.3

[97] Colwell R.P. et al., “A VLIW Architecture for a Trace Scheduling Compiler”, *IEEE Trans. on Computers*. Vol. 37, No. 8, August 1988 4

[98] Lowney P.G. et al., “The Multiflow Trace Scheduling Compiler”, *J. of Supercomputing*, Vol. 7, No. 1-2, March 1993 4, 4.2

[99] Teich J., “Digitale Hardware/Software-Systeme”, *Springer*, 1997 4, 4.2

[100] Koch A., “Adaptive Rechensysteme und ihre Entwurfswerkzeuge”, *Proc. 10th E.I.S. Workshop*, Dresden (Germany), April 2001 4, 4.2, 4.4.3

[101] Ye Z.A., Shenoy N., Banerjee P., “A C Compiler for a Processor with a Reconfigurable Functional Unit”, *Proc. ACM/SIGDA Intl. Symp. on FPGAs (FPGA)*, Monterey (CA, USA), February 2000 4.1

[102] Bondalapati K., Diniz P., Duncan P., Granacki J., et al., “DEFACTO: A Design Environment for Adaptive Computing Technology”, *Proceedings of the 6th Reconfigurable Architectures Workshop*, 1999 4.1

[103] Diniz P., Hall M., Park J., So B., Ziegler H, “Bridging the Gap between Compilation and Synthesis in the DEFACTO System”, *Proc. 14th Workshop on Languages and Compilers for Parallel Computing*, 2001 4.1

[104] Weinhardt M., Luk W., “Pipeline Vectorization for Reconfigurable Systems”, *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa (CA, USA), April 1999 4.1

[105] Weinhardt M., Luk W., “Evaluating Hardware Compilation Techniques”, *Proc IEEE Symp. on Field-Programmable Custom Computing Machines*, Napa (CA, USA), April 2000 4.1

[106] Maruyama T., Hoshino T., “A C to HDL Compiler for Pipeline Processing on FPGAs”, *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, Napa (CA, USA), April 2000 4.1

[107] Gokhale M., Stone J., Arnold J., Kalinowski, M., “Stream-Oriented FPGA Computing in the Streams-C High Level Language”, *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, Napa (CA, USA), April 2000 4.1

[108] Frigo J., Gokhale M., Lavenier D., “Evaluation of the Streams-C C-to-FPGA Compiler: An Applications Perspective”, *Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, Monterey (CA, USA) February 2001 4.1

[109] Cooper K.D., Torczon L., “Engineering a Compiler” *Morgan Kaufmann Pub.*, 2004 4.2

[110] Muchnick S.S., “Advanced Compiler Design and Implementation”, *Morgan Kaufmann Pub.*, 1997 4.2

[111] Wolfe M., “High Performance Compilers for Parallel Computing”, *Addison-Wesley*, 1996 4.2, 4.3

[112] Kasprzyk N., Koch A. “Advances in Compiler Construction for Adaptive Computers”, *Proc. Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, Las Vegas (NV, USA), June 2001 4.2

[113] Kasprzyk N., Koch A., Golze U., Rock M. “Eine effiziente Kontrollfluss-Repräsentation für die Erzeugung von Datenpfaden”, *Proc. 11th E.I.S. Workshop*, Erlangen (Germany), March 2003 4.2

[114] Kasprzyk N., Koch A., Golze U., Rock M. “An Improved Intermediate Representation for Datapath Generation”, *Proc. Intl. Conf. on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas (NV, USA), June 2003 4.2

[115] Koch A., Kasprzyk N., “Module Generators Driving the Compilation for Adaptive Computing Systems”, *Proc. IEEE Intl. Symp. on Field-Configurable Computing Machines (FCCM)*, Napa Valley (CA, USA), April 2002 4.2, 4.7.1

[116] Kasprzyk N., Koch A., “Verbesserte Hardware-Software-Partitionierung für Adaptive Computer”, *Proc. Conf. on Architecture of Computing Systems (ARCS)*, Augsburg (Germany), March 2004 4.2, 4.4.4, 4.5, 4.5, 4.5.1

[117] Rock M., Koch A., “Architecture-Independent Meta-Optimization by Aggressive Tail Splitting”, *Proc. Euro-Par*, Pisa (I), August 2004 4.2

[118] Li. Y., Callahan T., Darnell E., Harr R., et al., “Hardware-Software Co-Design of Embedded Reconfigurable Architectures”, *Proc. Design Automation Conference (DAC)*, 2000 4.2, 4.5.1

[119] Lange H., Koch A., “Memory Access Schemes for Configurable Processors”, *Proc. Intl. Conf. on Field-Programmable Logic (FPL)*, Villach (AT), August 2000 4.6.3

[120] Moll M., "Kontrollflussabhängige Analyse von Feldzugriffen in Schleifen unter Verwendung der Omega-Bibliothek", *diploma thesis*, TU Braunschweig, Abt. E.I.S. (Germany), November 2004 4.6.3

[121] Hutchings B., Bellows P., Hawkins J., Hemmert, S., et al., " A CAD Suite for High-Performance FPGA Design", *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa (CA, USA), April 1999 4.7.1, 4.7.1

[122] Koch A. "Generator-based Design Flows for Reconfigurable Computing: A Tutorial on Tool Integration using FLAME", *Proc. PACT98 Workshop on Reconfigurable Computing*, Paris (F), October 1998 4.7.1

[123] Koch A., "Enabling Automatic Module Generation for FCCM Compilers", *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa (CA, USA), April 1999 4.7.1

[124] Koch A., "On Tool Integration in High-Performance FPGA Design Flows", *Proc. Intl. Workshop on Field-Programmable Logic (FPL)*, Glasgow (UK), September 1999 4.7.1

[125] Koch A., "Creation and Embedding of Complex Parameterized Hardware Objects", *Proc. Workshop on Engineering of Reconfigurable Hardware/Software Objects*, Las Vegas (NV, USA), June 2000 4.7.1, 4.7.1, 4.7.2

[126] Koch A., "Compilation for Adaptive Computing Systems Using Complex Parameterized Hardware Objects" *J. of Supercomputing*, Vol. 21, No. 2, February 2002 4.7.1, 4.7.1, 4.7.2

[127] Koch A., "FLAME: A Flexible API for Module-based Environments", *E.I.S. Technical Report 2004-01*, Tech. Univ. Braunschweig (Germany), Dept. for Integrated Circuit Design (E.I.S.), November 2004 4.7.1, 4.7.2

[128] Rumbaugh J., Jacobson I., Booch G., "The Unified Modeling Language Reference Manual 2nd ed.", *Addison-Wesley*, 2004 4.7.1

[129] Neumann T., Koch A., "A Generic Library for Adaptive Computing Environments" *Proc. Workshop on Field-Programmable Logic (FPL)*, Belfast (UK), August 2001 4.7.1

[130] Koch A., "FLAME Library Specification", *E.I.S. Technical Report 2004-02*, Tech. Univ. Braunschweig (Germany), Dept. for Integrated Circuit Design (E.I.S.), November 2004 4.7.1

[131] Koch, A., "Regular Datapaths on Field-Programmable Gate Arrays", *doctoral thesis*, Tech. Univ. Braunschweig (Germany), September 1997 4.7.1, 4.7.1

[132] Cong J., Peck J., Ding Y., "RASP: a general logic synthesis system for SRAM-based FPGAs", *Proc. Intl. Symp. on Field Programmable Gate Arrays (FPGA)*, Monterey (CA, USA), February 1996 4.7.1

[133] Koch, A., "Module Compaction in FPGA-based Regular Datapaths", *Proc. 33rd Design Automation Conference (DAC)*, Las Vegas (NV, USA), 1996 4.7.1

[134] Schneier B., "Applied Cryptography 2nd ed.", *John Wiley & Sons*, January 1996 4.7.2

[135] Lange H., Koch A., "Hardware/Software-Codesign by Automatic Embedding of Complex IP Cores", *Proc. Intl. Conf. on Field-Programmable Logic (FPL)*, Antwerp (BE), September 2004 4.7.2

[136] Lange H., Radetzki M., "IP Configuration Management with Abstract Parameterizations", *Proc. Intl. Workshop on IP Based SoC Design*, Grenoble (F), 2002 4.7.2

List of Included Publications

1. Koch A., Golze U., “Practical Experiences with the SPARXIL Co-Processor”, *Proc. 31st Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove (CA, USA), November 1997
2. Koch A. “Generator-based Design Flows for Reconfigurable Computing: A Tutorial on Tool Integration using FLAME”, *Proc. PACT98 Workshop on Reconfigurable Computing*, Paris (F), October 1998
3. Koch A., “Enabling Automatic Module Generation for FCCM Compilers”, *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa (CA, USA), April 1999
4. Koch A., “On Tool Integration in High-Performance FPGA Design Flows”, *Proc. Intl. Workshop on Field-Programmable Logic (FPL)*, Glasgow (UK), September 1999
5. Böge M., Koch A., “A Processor for Artificial Life Simulation”, *Proc. Intl. Workshop on Field-Programmable Logic (FPL)*, Glasgow (UK), September 1999
6. Koch A., “Adaptive Rechensysteme - Architekturen und Werkzeuge”, *Proc. E.I.S. Workshop*, Darmstadt (Germany), September 1999
7. Koch A., “A Comprehensive Prototyping Platform for Hardware-Software Codesign”, *Proc. Workshop on Rapid Systems Prototyping*, Paris (F), June 2000
8. Koch A., “Creation and Embedding of Complex Parameterized Hardware Objects”, *Proc. Workshop on Engineering of Reconfigurable Hardware/Software Objects*, Las Vegas (NV, USA), June 2000
9. Lange H., Koch A., “Memory Access Schemes for Configurable Processors”, *Proc. Intl. Conf. on Field-Programmable Logic (FPL)*, Villach (AT), August 2000
10. Koch A., “Adaptive Rechensysteme und ihre Entwurfswerkzeuge”, *Proc. 10th E.I.S. Workshop*, Dresden (Germany), April 2001
11. Kasprzyk N., Koch A. “Advances in Compiler Construction for Adaptive Computers”, *Proc. Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, Las Vegas (NV, USA), June 2001
12. Neumann T., Koch A., “A Generic Library for Adaptive Computing Environments” *Proc. Workshop on Field-Programmable Logic (FPL)*, Belfast (UK), August 2001

13. Koch A., “Compilation for Adaptive Computing Systems Using Complex Parameterized Hardware Objects” *J. of Supercomputing*, Vol. 21, No. 2, February 2002

14. Koch A. , “Architectures and Tools for Heterogeneous Reconfigurable Systems”, *Proc. IEEE Workshop on Heterogeneous Reconfigurable Systems-on-Chip*, Hamburg (Germany), April 2002

15. Kasprzyk N., Koch A., Golze U., Rock M. “Eine effiziente Kontrollfluss-Repräsentation für die Erzeugung von Datenpfaden”, *Proc. 11th E.I.S. Workshop*, Erlangen (Germany), March 2003

16. Kasprzyk N., Koch A., Golze U., Rock M. “An Improved Intermediate Representation for Datapath Generation”, *Proc. Intl. Conf. on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas (NV, USA), June 2003

17. Schmidt C., Koch A., “Fast Region Labeling on the Reconfigurable Platform ACE-V”, *Proc. Intl. Conf. on Field-Programmable Logic (FPL)*, Lisbon (PT), September 2003

18. Kasprzyk N., Koch A., “Verbesserte Hardware-Software-Partitionierung für Adaptive Computer”, *Proc. Conf. on Architecture of Computing Systems (ARCS)*, Augsburg (Germany), March 2004

19. Rock M., Koch A., “Architecture-Independent Meta-Optimization by Aggressive Tail Splitting”, *Proc. Euro-Par*, Pisa (I), August 2004

20. Gädke H., Koch A., “Wavelet-based Image Compression on the Reconfigurable Computer ACE-V”, *Proc. Intl. Conf. on Field-Programmable Logic (FPL)*, Antwerp (BE), September 2004

21. Lange H., Koch A., “Hardware/Software-Codesign by Automatic Embedding of Complex IP Cores”, *Proc. Intl. Conf. on Field-Programmable Logic (FPL)*, Antwerp (BE), September 2004

22. Koch A., “FLAME: A Flexible API for Module-based Environments”, *E.I.S. Technical Report 2004-01*, Tech. Univ. Braunschweig (Germany), Dept. for Integrated Circuit Design (E.I.S.), November 2004

23. Koch A., “FLAME Library Specification”, *E.I.S. Technical Report 2004-02*, Tech. Univ. Braunschweig (Germany), Dept. for Integrated Circuit Design (E.I.S.), November 2004