ACRES Architecture and Compilation

Boon Seong Ang, Michael Schlansker
HP Laboratories Palo Alto
HPL-2003-209(R.1)
April 2, 2004*

reconfigurable computing, spatial compilation, pipelined interconnect, distributed control, remote branch architecture, reconfigurable memory

High-performance computing engines often provide product-defining functionality within consumer devices. These devices are traditionally implemented using either ASICs or embedded processors. ASICS are inflexible and require high design cost while embedded processors provide inadequate compute power and efficiency for specialized applications. This work describes the ACRES (Architecture and Compiler for REconfigurable Systems) platform that combines the flexibility of a programmable technology and the efficiency of custom hardware without incurring high-cost, high-risk chip development. The ACRES platform consists of a programmable computing fabric architecture and an associated spatial compiler. A spatial compilation procedure ties together novel configurable elements including interconnect, memory and distributed control. A key compilation feature is early spatial planning that influences subsequent architectural decisions. Operation scheduling follows and uses accurate latency information to generate efficient execution schedules. This paper describes the motivating assumptions, key ideas and technologies advocated in the ACRES platform. Implementation and evaluation is the topic of future work.

* Internal Accession Date Only
Approved for External Publication

ACRES Architecture and Compilation

Boon Seong Ang and Michael Schlansker

Abstract

High-performance computing engines often provide product-defining functionality within consumer devices. These devices are traditionally implemented using either ASICs or embedded processors. ASICS are inflexible and require high design cost while embedded processors provide inadequate compute power and efficiency for specialized applications. This work describes the ACRES (Architecture and Compiler for REconfigurable Systems) platform that combines the flexibility of a programmable technology and the efficiency of custom hardware without incurring high-cost, high-risk chip development. The ACRES platform consists of a programmable computing fabric architecture and an associated spatial compiler. A spatial compilation procedure ties together novel configurable elements including interconnect, memory and distributed control. A key compilation feature is early spatial planning that influences subsequent architectural decisions. Operation scheduling follows and uses accurate latency information to generate efficient execution schedules. This paper describes the motivating assumptions, key ideas and technologies advocated in the ACRES platform. Implementation and evaluation is the topic of future work.

Keywords: Reconfigurable computing, spatial compilation, pipelined interconnect, distributed control, remote branch architecture, reconfigurable memory.

1 Introduction

As VLSI density increases and cost decreases, we see a trend toward the use of digital computer systems to support many aspects of our everyday lives. Applications such as imaging, video, communications, and networking have been revolutionized by inexpensive and specialized digital processing systems. Embedded digital systems commonly replace previous generation analog systems to provide lower cost and unique product enhancing features that were previously impossible. At the core of these consumer products, custom embedded systems incorporate inexpensive, high-density circuitry to produce levels of computational power that, for highly specialized applications, can surpass supercomputer levels of performance. This is due, in part, to the highly parallel nature of these applications. Custom hardware solutions, usually in the form of application-specific integrated circuits (ASICs), are designed to exploit this parallelism using specialized hardware circuitry. For very high volume products, ASICs continue to provide a viable solution. Unfortunately, ASICs require chip development that is expensive, high risk, and slow to market. While the cost is acceptable for high-volume products with stable functionalities, many products that do not fit these criteria cannot be brought to market with ASICs.

A common alternate approach for building embedded systems is to employ general-purpose, embedded processors. This vastly reduces hardware development, replacing it

with software/firmware development which is much less costly, faster to market, and generally incurs lower risk because firmware is easily upgraded. General-purpose processors provide standardized data-paths and powerful instruction units. With sophisticated compilers, these processors execute a broad variety of complex algorithms in software. However, general-purpose processors do not scale efficiently. For some high performance applications, embedded processors either cannot deliver the required level of performance, or can only do it with designs that are either too expensive, or consume too much power. In comparison, ASICs use statically customized control and data-paths to scale efficiently to levels of performance that cannot be achieved with general-purpose processors. Highly-parallel hardware is greatly simplified when memory, data-transfer, arithmetic, and control are statically customized for an application by hardware designers.

This work is motivated by the following goal: achieve much of the flexibility of a programmable technology and the efficiency of custom hardware without incurring high-cost and high-risk chip development. The ACRES approach combines high-level synthesis and compilation technology with architectural innovation that exploits features from general-purpose processor, ASIC, and FPGA (field programmable gate arrays) architectures. The goal is to deliver levels of cost, performance and programmability that were not previously possible. This paper describes key ideas that form a foundation for the ACRES platform. ACRES' high-level concepts have been carefully designed and provide an excellent starting point for a next-generation programmable architecture. However, much of the implementation and evaluation of an ACRES platform remains to be done. Consequently, this paper will, at times, describe a number of alternative implementations for which the relative merits remain to be evaluated.

In the rest of this document, Section 2 discusses the motivation and assumptions driving the ACRES platform design. Section 3 follows with an overview of the ACRES platform, including the computing fabric, the coordination and control model, and the spatial compiler. Detailed descriptions of ACRES' control, interconnect, and configurable memory technologies follow. First, distributed control is the topic of Section 4, which focuses on a flexible remote branch model and associated hardware that enables both tightly coupled VLIW-style parallelism as well as loosely coupled MIMD-style parallelism. Section 5 describes the pipelined interconnect technology, the tile abstraction, and how spatial compilation utilizes the abstraction. Section 6 is devoted to ACRES' reconfigurable memory technology. Section 7 describes related work and Section 8 concludes with highlights of ACRES' unique features.

2 Background and Assumptions

The ACRES platform is motivated by a number of observations about current technology trends. The trajectory of chip technology is well understood as device feature size decreases steadily in a trend that doubles the number of available on-chip transistors every one and a half to two years. Today's (2003) top-of-the-line microprocessors such as the Itanium2 6M, incorporate around half a billion transistors. This trend is expected to continue, in the near term with additional innovation in silicon technology and farther out with new, more exotic technology such as electronic nanotechnology. While this provides enormous opportunities, it also raises a number of difficult challenges.

Communication is becoming a primary consideration in micro-architecture. Compared to transistors, wires are increasingly costly. As chip technology scales down to produce smaller, faster transistors, overall physical chip size typically remains constant to incorporate more functions on a single chip. As a result, the time to send a signal from one side of the chip to the opposite side, relative to transistor switching time, increases. Whereas smaller transistors switch faster, thinner wires actually have higher resistance. Attempts at reducing resistance by making wires taller introduce cross-talk and increased capacitive loading when parallel wires run in close proximity. As a result, the relative speed of communication with respect to that of computation is deteriorating. A similar effect is happening with energy consumption, as the energy consumption for communication increases relative to that for computation.

The increasing number of available transistors is exposing scaling limitations of traditional microprocessors. Traditional microprocessors exploit only modest parallelism. For superscalar architectures, the complexity of coordinating the issue, execution and retirement of multiple instructions in parallel is a key limiting factor. For both superscalar and VLIW architectures, several key components including the register files, the instruction issue unit, and the memory are centralized and do not scale efficiently. In addition, executing only a single thread of execution on a traditional microprocessor also limits the degree of sustained parallelism.

It is instructive to contrast difficult-to-scale microprocessors with more readily scalable ASICs. Unconstrained by the structure of a pre-defined platform, ASICs exploit multiple forms of parallelism present in specialized applications. Specialization for a particular application is used to simplify control and data-path elements. The micro-architecture of an ASIC is tailored to efficiently implement its specific computation with specialized and distributed components for control, memory, interconnect and compute. Centralization is avoided and replaced by distributed and localized structures that collaborate to implement overall functionality. It would be very beneficial if these positive attributes of ASICs could be harnessed in a programmable technology. This work addresses this goal by integrating and building upon prior reconfigurable, CAD, and compiler research.

A serious challenge for spatial computing – the parallel, distributed and specialized-for-application style of computation commonly exploited in ASICs – is its design process. ASICs and reconfigurable systems based on FPGA-style technology are commonly designed using a complex, low-level hardware design methodology. In contrast to software design methodology, current hardware design methodology exposes many

detailed design decisions to designers. Examples of low-level manual specifications include RTL cycle-level design specifications and chip floor-planning. These specifications are tedious and do not port across implementation targets.

Low-level specification also limits the ability of synthesis and compilation tools to optimize a design. An example is the task of reaching timing closure: because designs are specified at the clock cycle level, automation tools have limited flexibility in changing clock cycle boundaries. As a result, human is kept tightly involved in the loop and must make manual changes at the specification level as a design iterates towards timing closure. Had the specification been done at a higher level of abstraction, an automated tool could potentially automate or assist this iteration process. We believe that for this and a number of other reasons, raising the level of a design's specification is crucial to making spatial computing accessible to many more applications.

We are interested in a programmable technology because of important benefits that are apparent in systems that currently employ embedded processors. A programmable technology allows the development cost of the actual underlying hardware to be amortized over many designs. Chip development cost has increased as chip technology heads towards ever smaller device sizes. Fabrication masks have become increasingly expensive, and many effects have to be carefully modeled and checked during hardware design to minimize the chance of problems arising from these effects. Programmable technology typically also incurs shorter development time, and allows easy upgrades or bug fixes. Our goal is to achieve the benefits of programmability while providing the efficiency and high-performance of spatial computing.

3 Overview of the ACRES Reconfigurable Platform

The ACRES reconfigurable platform consists of the architecture for a programmable computing fabric as well as an associated spatial compiler. The two work collectively to deliver the benefits of ACRES. The computing fabric architecture is structured in ways that facilitate spatial compilation, while the spatial compiler automates the process of exploiting computing fabric features.

3.1 The ACRES Computing Fabric

A key approach underlying ACRES is the promotion of static distribution and local reuse of both data and control. This serves two objectives. First, if control and data are stored close to their use, communication costs decrease, and parallel systems are inherently more scalable and efficient. Second, localization promotes specialization. Memory and arithmetic structures that process specific rather than generic data can be specialized to actual need, such as limiting data-path connectivity, partitioning memory and reducing bit-width of data transfer and arithmetic structures.

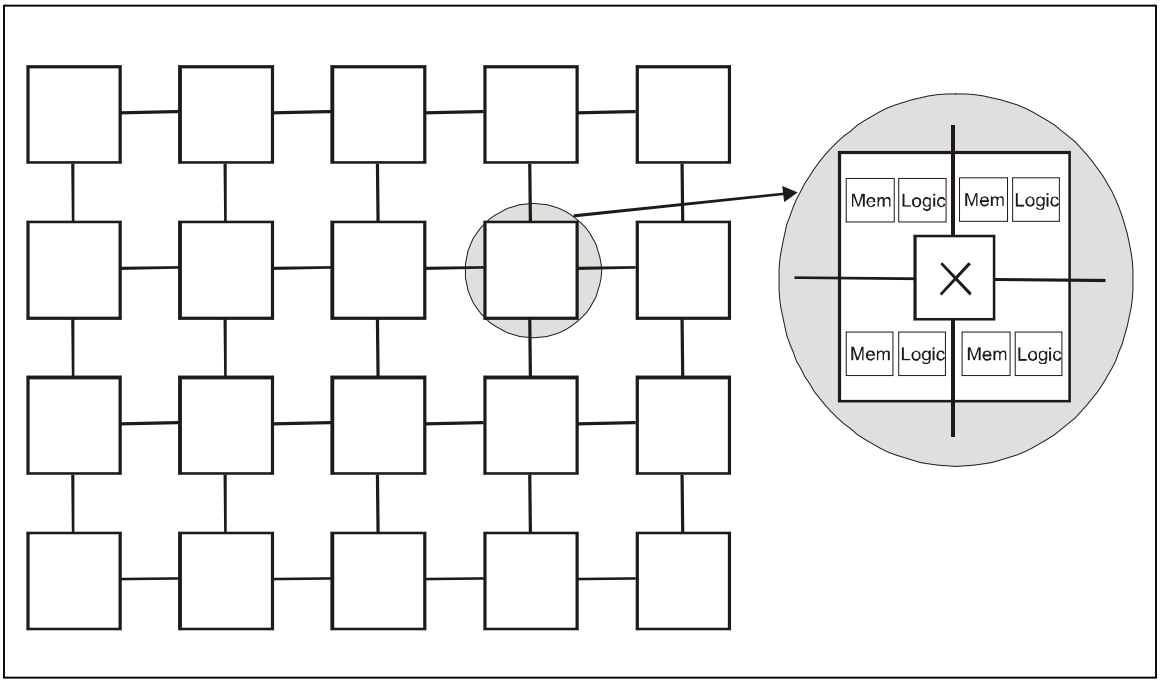


Figure 1: An ACRES computing fabric.

To facilitate the distribution and localization of data and control, an ACRES computing fabric is composed of repeating subunits called tiles as shown in Figure 1. Each tile has a collection of heterogeneous components such as compute, logic, memory, interconnect and control elements that are configurable to varying degrees. As an example, one can imagine using one of today's moderate-size FPGAs as a single ACRES tile. A typical ACRES chip is composed of multiple identical tiles laid out in a repetitive structure, such as a two-dimension array. At this coarse level, an ACRES chip appears homogenous. This organization ensures that any sub-part of a chip larger than a tile is self-contained,

capable of independent execution and control. Interconnection between tiles further enables multiple tiles to be leashed together either in loose or tight collaboration.

The ACRES hardware is (for now) synchronously clocked across the entire computing fabric. Synchronous clock distribution is well understood and hardware designers can synchronously clock very large chips. Synchronous clocking provides important benefits. VLIW and systolic processors use synchronous clocking to support complex algorithms with very simple control. Synchronous designs efficiently share logic and wires for complex algorithms that require a changing pattern of operations from cycle to cycle. Additional hardware asynchrony can be incorporated into ACRES if that proves essential.

To alleviate long distance communication inefficiency, and to reduce the physical wiring resources devoted to communication, ACRES adopts a two-tier interconnect model that pipelines and time-multiplexes long connections. This two-tier interconnect model distinguishes between intra-tile and inter-tile communication. All inter-tile communication is pipelined, with pipeline stages inserted at regular interval. In a simple model with square tiles, that interval is the linear dimension of a tile. Thus, ACRES introduces a very simple timing model allowing compilation tools to reason about time without analog modeling and simulation. Specifically, given a target system clock speed, each side of a square tile corresponds roughly to the signal propagation distance within one system clock cycle. In contrast to inter-tile communication, intra-tile communication involves source and destinations close enough to complete within one clock cycle.

An ACRES computing fabric's decomposition into synchronous tiles with pipelined interconnect provides a number of advantages. The approach assures that long data transfer paths support streams of pipelined operands traversing their length. Furthermore, combinational nets do not span tiles and thus, timing analysis can be separately performed within each tile for the intra-tile nets. Perhaps the biggest benefit of the decomposition of the ACRES fabric into tiles is that it provides fixed, realistic and simple geographic timing assumptions that the ACRES compiler uses to shape application-specific system architectures. Most design systems create a design with little notion of geography, and then place and route the resulting hardware schematic. In the ACRES spatial compiler, floor-planning-level information is used to automatically shape the topology of an application-specific processor's data-path and to define the processor's execution schedule and control so that they are amenable to the timing limitations that arise from the geographic constraints of the chosen floor plan.

ACRES incorporates a reconfigurable memory architecture that enables flexible mapping of logical memory objects to RAM blocks. The underlying assumption is that embedded applications present opportunities for statically distributing data into distinct memory structures that can be accessed in parallel. Traditional FPGAs provide RAM blocks with configurable width and depth via statically configured interconnect. ACRES builds upon this foundation by augmenting the memory access paths with switch and register elements that promote the dynamic sharing of these paths. Furthermore, the spatial compiler deals with assignment, detailed scheduling and timing issues of shared memory access paths. Using pipelined, time-multiplexed interconnect technology and static scheduling, ACRES' reconfigurable memory technology provides efficient parallel access to memory objects.

3.2 ACRES Control and Coordination Model

Spatial computing requires control and coordination beyond that commonly encountered in compilers for traditional microprocessors. A traditional microprocessor is equipped with a centralized instruction issue unit, coupled with one or more branch units, to control computation on the microprocessor. The instruction issue unit issues instructions that control the cycle-by-cycle operation of the microprocessor's resources, while the branch units direct the flow of control to determine a sequence of issued instructions.

As the number of functional units on programmable devices increases, controlling these devices efficiently and flexibly becomes a problem. A traditional branch unit steers execution in time, with effect over global spatial scope, *i.e.* over all of the microprocessor's resources. Future ACRES devices contain ample resources to configure a huge number of functional units. A flexible control scheme is needed to allow these units to be used either separately or collectively in accordance with application needs. The ACRES platform adopts a control architecture called the *Remote Branch Architecture* which extends the notion of branching to include steering execution over space. This allows changes in the amount of resource that are allocated to a logical flow of control. Used as the target control model by the ACRES spatial compiler, this control architecture can be efficiently implemented on an ACRES computing fabric.

The remote branch architecture can be viewed abstractly as replacing the single point of control used in traditional microprocessors with a multitude of spatially distributed control units, each controlling a small number of functional units in its geographical vicinity. These separate *processing cells* are able to *operate both independently*, collaborating in a loose manner *and collectively*, executing in a statically orchestrated, tightly coupled manner. A processing cell is able to rapidly shift between a loosely coupled and tightly coupled mode of collaboration with another processing cell.

At any moment in time, ACRES processing cells may be grouped into DVLIW clusters (Distributed VLIW clusters). A single DVLIW cluster may contain as few as a single processing cell or as many as all the processing cells within a device. The processing cells within a single cluster collectively act as a single processor, with tight coupling and static scheduling employed across them. Separate DVLIW clusters are loosely coupled and follow independent flow of control. This partitioning of cells into DVLIW clusters is done *logically*, permitting rapid and low cost repartitioning of the groupings as illustrated in Figure 2. Central to this spatial control architecture is the processing cell's ability to *issue remote branches* to a *select* number of destination processing cells and to *incorporate remotely initiated branches* from a number of sources.

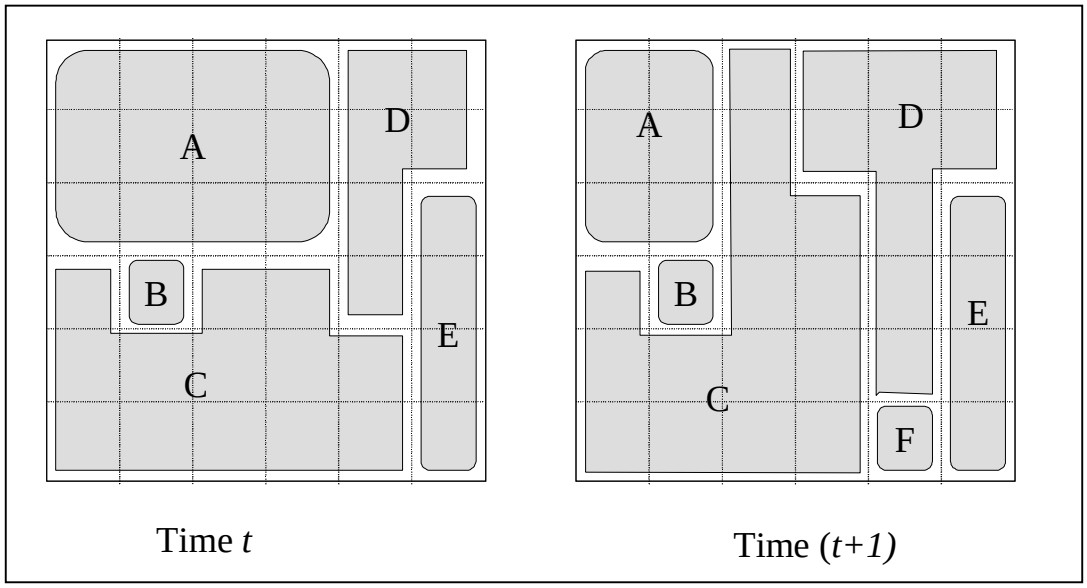


Figure 2: Partitioning of a system into multiple DVLIW clusters and repartitioning of the system over time.

ACRES uses two control mechanisms: configurable control, and dynamic control. At a low level, the ACRES computing fabric employs configurable control similar to that used in FPGAs to configure application specific components, such as state-machines, custom data-paths, and custom instruction decoders. While configurable control is not changed rapidly, dynamic control is employed where control needs to change rapidly, such as on a clock cycle by clock cycle basis. The ACRES architecture implements dynamic control using both state-machines and microprocessor-style, memory-based instructions. Each has benefits. State-machines are simple and efficient when used to implement repetitive sequences often encountered in code kernels. On the other hand, when there is high control variability, memory-based instructions quickly become a very efficient way of encoding and implementing complex control sequences.

3.3 The ACRES Spatial Compiler

The ACRES spatial compiler is geared towards generating application-specific systems where data and control are stored near their uses, and transmitted over small distances. This is accomplished in a multi-phase process as illustrated in Figure 3. First, the ACRES compiler transforms the application and distributes arithmetic, memory, and control within a virtual architecture that is tailored to the application. Then, a spatial plan is developed that maps the application specific virtual architecture onto the physical architecture to minimize the distance traveled by data and control. Next, transfer times that are established with knowledge of a spatial plan, are extracted by the spatial compiler and used to calculate a final schedule. In this way, the architecture and its execution schedule are carefully tuned to accommodate geographic layout and fabric timing characteristics.

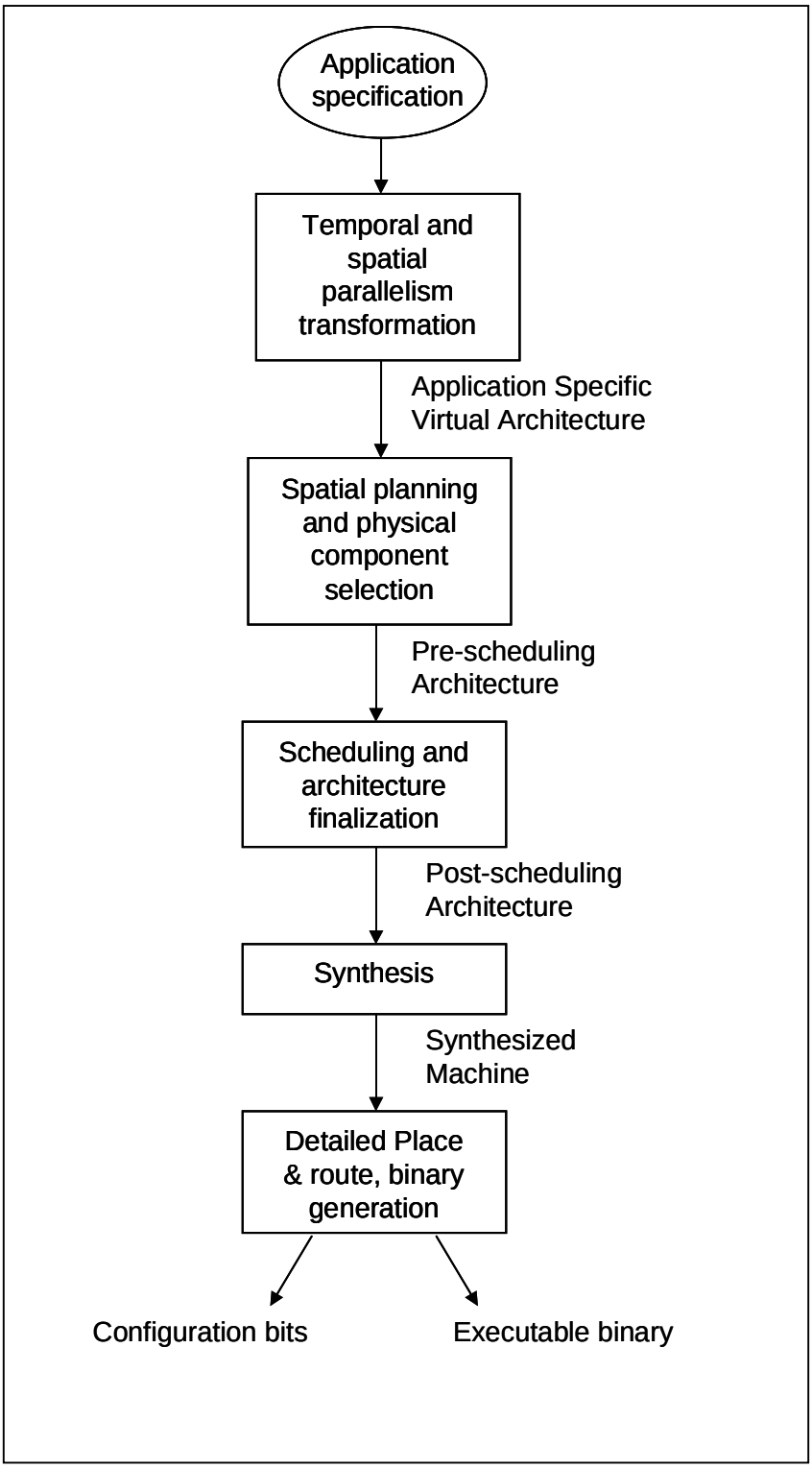


Figure 3: ACRES spatial compilation flow.

Traditional compilers optimize code for a fixed operation repertoire and for a fixed processing topology. To exploit the benefits of spatial computing, the ACRES spatial compiler statically determines both an operation repertoire and a processing topology that is optimized for the given application. The repertoire and topology are typically very limited, unlike those needed for general-purpose computations. The ACRES computing fabric is statically configured to exploit efficiencies that stem from both the restricted use of operations as well as the restricted topology for data flow between operations.

The ACRES compiler defines an *application specific virtual architecture* -- a virtual processing graph that explicitly represents application parallelism and opportunities for the spatial distribution of computation. While programs may initially be specified as conventional sequential programs, the ACRES compiler uses explicit representations for VLIW-, SIMD-, and MIMD-style parallelism in order to expose opportunities for spatial processing. The result is a virtual processor that exposes enough spatial parallelism to allow the exploitation of available silicon area.

Traditional serial programs operate on centralized memory and a central register file. When programs execute sequentially, centralized storage offers numerous advantages when performing tasks such as register allocation. However, when programs execute on highly parallel hardware, these advantages are outweighed by the high costs for transferring many operands in parallel into and out of any central file. Programs, as represented within the ACRES compiler, may also operate on distributed register and memory structures that expose opportunities for decentralized memory reference. The ACRES compiler transforms programs to promote the localized use of registers and memory. Where possible, programs are transformed so that operands are copied out of centralized memory structures and repeatedly referenced within distributed storage structures thus allowing efficient and decentralized parallel processing.

In a subsequent spatial planning step, the ACRES compiler maps a distributed virtual architecture and distributed virtual storage onto the physical tiles provided by the ACRES fabric. These tiles provide the compute, memory, and interconnect resources needed to execute the computation. The resulting placement in two-dimensional silicon allows further reasoning about timing details needed to complete the design of a high-performance processor. While consideration of communication delay within a tile is deferred until much later in the spatial compilation process, communication delay between tiles is explicitly represented as an integer number of clock cycles needed to support any long distance (inter-tile) transfer. This provides a temporal framework for further compilation.

While other application-specific processing approaches determine a system architecture and its software without any awareness of the final physical placement of components, the ACRES compiler uses spatial information to determine both a final system topology as well as a final program schedule. On an ACRES fabric, wires that interconnect tiles are often shared among multiple logical signals (similar to the idea of virtual wires [4, 66]) that might traverse each wire at disjoint moments in time. Rather than first determining a final application-specific system interconnect and then placing and routing all components within the fabric, the system interconnect topology is finalized with an awareness of the physical placement of system structure.

Before the ACRES spatial compiler proceeds with scheduling execution, a set of physical components, such as functional units, have to be selected. While we expect this step to happen after virtual architecture generation but before scheduling, the exact phase ordering of this step with respect to the overall compilation flow is a topic for further research. One possibility is to first determine the set of physical components and map the application specific virtual architecture to these components prior to spatial planning. Another choice is to do it coincidently with spatial planning, *i.e.* as virtual operations are mapped to tiles, previously instantiated components are re-used, or physical components are instantiated if necessary.

The current design expects a *pre-scheduling architecture* by the time the spatial compiler begins scheduling. The pre-scheduling architecture specifies a set of physical components, an assignment of physical component to tiles, and an assignment of virtual operations to candidate physical components. This last assignment needs to be specific enough so that a virtual operation is unambiguously assigned to a tile. However, if there are several physical components within the tile that are capable of performing the operation, the exact choice can be left to the scheduler.

After determining where processing is performed, where operands are stored, and which route logical signals use to traverse the inter-tile interconnect, the ACRES compiler creates efficient program schedules that accommodate flight times needed for long distance data transport. This allows the construction of extensible systems where all logical signals that span multiple tiles are pipelined to maintain a high clock frequency.

Scheduling transforms the pre-scheduling architecture to *post-scheduling architecture*. By determining the time and the specific functional unit on which an operation is done, scheduling also determines the number of registers needed to store temporaries, and the static interconnect needed to support the required data flows. A detailed program schedule that is compatible with this system architecture is then translated into code and state machines to control dynamic execution. The post-scheduling architecture can be viewed as similar to RTL (Register transfer level) machine specification.

The ACRES platform supports the use of either memory-based instructions or state machines to control execution. One aspect of the post-scheduling architecture, different from traditional hardware RTL specification, is that control may be encoded with memory-based instructions. Memory-based instructions are emitted as object code while state machines are translated into configuration bits. Instruction-based control requires the synthesis of an instruction unit and its object code [2]. In addition to object code, any instruction-based control specified in the post-scheduling architecture is processed to form appropriate instruction memory, instruction sequencing, and instruction issue units.

The synthesis and place & route phases of spatial compilation are close to traditional CAD. Synthesis determines all low-level logic for both control units and data-paths. As a part of synthesis, technology mapping maps each low-level logic element into actual components that are provided by the fabric. The synthesis process produces a *synthesized machine*. The synthesized machine is then placed and routed. This phase can operates separately on individual tiles as the spatial planning phase has already determined in which tile each logic element or memory resides. Traditional place and route technology is suitable for this local tile-by-tile place and route pass. In a final step, the ACRES

compiler emits configuration bits that describe the data path and control hardware as well as the object program that must be loaded into instruction memories. Configuration bits and object programs are then downloaded to customize the ACRES fabric at run time.

4 Spatial and Temporal Control and Coordination

ACRES implements control scheme spanning a spectrum from fully spatial control to fully temporal control. Each node in a fully spatial dataflow processor executes only a single arithmetic operation. Fully spatial control is simple and static, data is transferred between nodes in a statically fixed pattern. Fully spatial control is described using configuration bits that specify the chosen static data-path. Fully spatial systems are highly scalable but not very flexible.

Fully temporal control is best illustrated by RISC architectures where operations execute sequentially on a centralized processing unit that provides all needed functions and storage. Operands are specified in a flexible manner using register and memory addresses. Conditional branching provides powerful sequencing to support complex software. Fully temporal control is specified using a conventional object program consisting of linear code sequences connected by conditional branches. Fully temporal systems are flexible but not very scalable.

The ACRES compiler blends features from both spatial and temporal architectures and designs systems with control architectures that lie along the spectrum between the two extremes. Given an application, an architecture is statically specialized to that application but also incorporates dynamic switching and control elements needed to support application branches. The data-path is specialized so that each function unit is connected only to those function units that actually produce and consume needed operands. Control is also specialized and specifies only those function unit operand and operation choices that are needed for the given code.

For conventional architectures, temporal control, in the form of a sequence of instructions, is fetched from a central memory and transmitted to multiple function units. While this approach is favored for instruction units within conventional microprocessors, the approach does not scale and is not suited for highly spatial processors executing within a computing fabric. Instead, the ACRES architecture supports the spatial distribution of control in a framework called the remote branch architecture.

4.1 Remote Branch Architecture

Consider an example highly spatial ACRES virtual processor. The virtual processor executes as a lock-step or VLIW-style processor that is geographically distributed within a computing fabric. This processor is an application-specific virtual processor that must be mapped onto an application-independent physical fabric. The ACRES spatial compiler divides the processor into spatially distributed processing cells where each processing cell is controlled by locally stored instructions or state machines. A remote branching capability facilitates the coordination between the distributed processing cells. When a processing cell executes a branch, it may change flow of control, not only for itself, but also for other processing cells. The branch is transmitted over a carefully controlled scope to all units that need to respond to the branch. Upon receiving the branch, affected processing cells transfer local execution to the new branch target to begin a coordinated task.

The ACRES control strategy allows the distribution of control for both lock-step VLIW or SIMD and loosely coupled MIMD style architectures. The system is synchronously clocked, and branch propagation takes a predictable number of cycles. This allows the system to behave as a lock-step VLIW processor even though control is physically distributed. When a branch is transmitted to all processing cells, all cells respond, thus allowing the ensemble to act as a single processor that responds to a single (possibly delayed) branch. Instructions are pre-dispersed and spatially placed close to the function units that execute them. We call this style of execution DVLIW (Distributed VLIW). Branches may also be transmitted over much more limited scope. Thus, in an extreme case, a branch is sent only from a processing cell to itself. This allows the construction of MIMD architectures that are needed to support independent threads of computation.

When a virtual processor is mapped onto a computing fabric, control can be implemented using either state machines or instruction units. In either case, the ACRES spatial compiler uses a unified distributed control abstraction of processing cells interacting through branch commands. The control implementation choice, whether state machine or instruction unit, is made during the spatial compilation process. It is possible that among the cells collaborating as a single VLIW-style processor, some are controlled with state machines, while others with instruction units.

When using state machines, a control sequencer is customized to exactly implement the required program steps. For extremely spatial and non-temporal programs, state machines become very simple. Control is constant, and the same control is executed on every cycle. More often, initialization and other requirements necessitate some non-trivial control. Extremely temporal programs may require very complex state sequences. While it is possible to encode such programs using state machines, this is often far more expensive than representing lengthy programs in high-density instruction memories for processing with an instruction unit.

4.2 Control Transformation during Spatial Compilation

The ACRES execution model allows multiple spatially distributed processing cells to execute as a single accelerator. Processing cells may act independently as a MIMD machine or in a lockstep VLIW-like manner using DVLIW execution as depicted in Figure 4. Processing cells are synchronously clocked from a common source. Branch commands are transmitted between processing cells that collaborate as a single VLIW. A branch is transmitted from an originating processing cell's branch unit to one or more recipient processing cells through a branch transmission network. The set of recipients, or the spatial scope of the branch, is programmable and determines whether processing cells act in a coupled or independent manner.

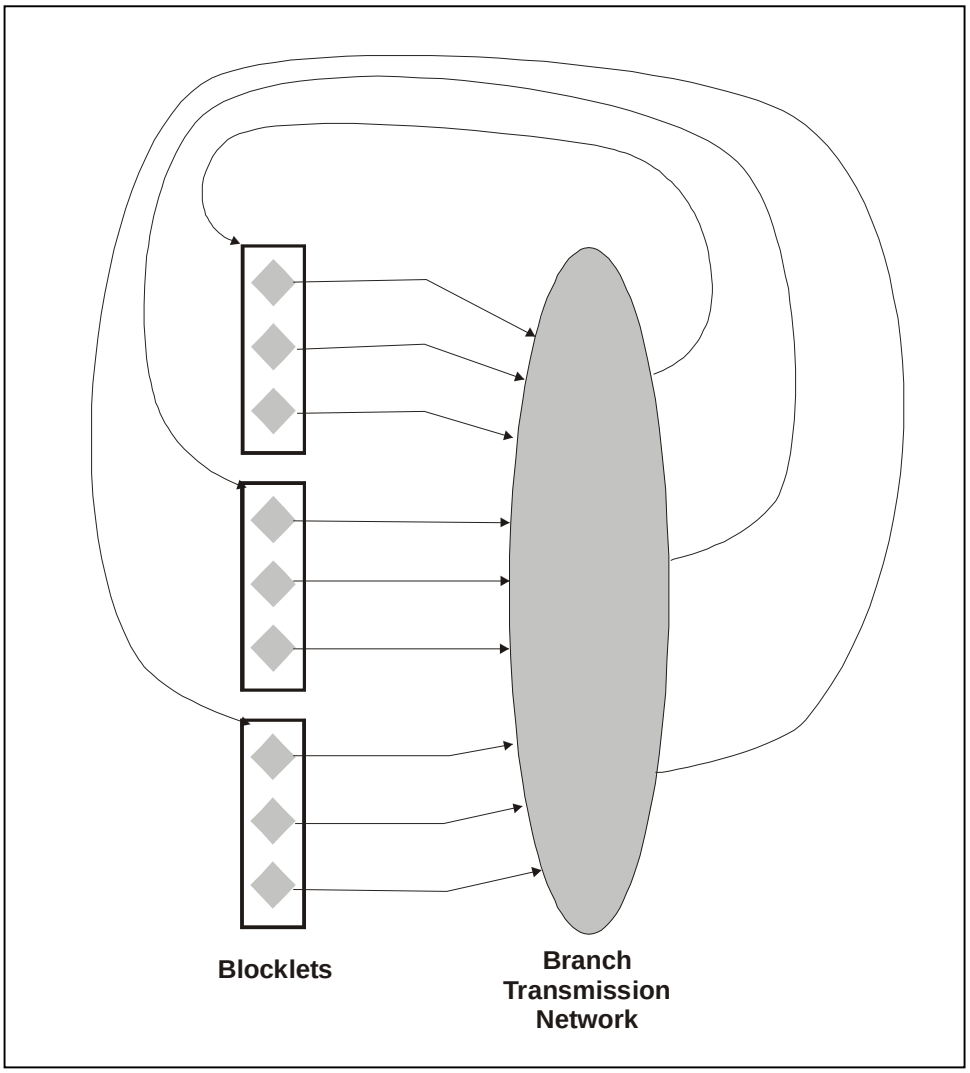


Figure 4: DVLIW execution.

During spatial compilation, the execution of a program thread is broken into single-entry multi-exit regions called blocklets (shaded diamonds in Figure 4) that are mapped onto concurrently executing processing cells (rectangles in Figure 4). A blocklet is a subset of a control block, such as a hyperblock [44], that is located within a single processing cell. A hyperblock begins execution when a branch jumps to the hyperblock by multicasting a branch command to each of the blocklets within the hyperblock. These blocklets begin execution with known delay offsets and collaborate in a synchronized manner to form a single spatially distributed hyperblock. Someplace within the hyperblock, a blocklet may branch and initiate the execution of one or more successor blocklets that implement the next hyperblock in the execution flow. Blocklets branch by transmitting a branch command through the branch transmission network to one or more processing cells. When a processing cell receives a branch command, it uses a branch target name, provided in the branch command, to identify the locally stored successor blocklet where execution continues.

The time of flight from a branch that initiates a hyperblock to each recipient blocklet may vary but is statically predictable. The ACRES compiler schedules code to correctly accommodate this branch latency while preserving a VLIW model of computation. Since any branch delay is predictable, the execution of blocklets can be predictably orchestrated to form a single logical hyperblock. This predictability enables coordination between collaborating blocklets without run-time synchronization.

ACRES adopts a synchronous VLIW-like control discipline for single threaded flow-of-control. Function units and memories are synchronously clocked with deterministic latency. Operations are statically scheduled to ensure that preceding operations complete operand processing before successor operations are initiated. Branches may have exposed latency and a compile time scheduler is crafted to ensure that operands that are guarded by a conditional branch are scheduled after any branch delay.

ACRES breaks computation into blocklets in order to preserve locality of control. Very early during ACRES' spatial compilation process, computation is mapped onto tiles in a spatial planning step (see Section 5 for further description of spatial planning and the tile abstraction). As a result of this mapping, code belonging to a hyperblock may be distributed across several tiles. ACRES introduces the notion of processing cells to take advantage of locality of control, with each processing cell controlling only resources residing within a close geographical locality. Our initial strategy is to have a direct correspondence between processing cell and tile. Thus at any one point in time the computation and private resources within each tile is controlled by a processing cell, and each processing cell is limited to a tile. Over time, through reconfiguration of static configuration, different processing cells may be programmed onto the same tile.

In a conventional processor with instruction-based control, a branch target address is used as a direct index into instruction memory. When a single branch target address is multi-cast to all processing cells that host its target blocklets, all instruction units in those processing cells must use a uniform program layout. Uniform layout requires that, for each hyperblock, blocklet entry addresses are the same across processing cells.

Uniform layout is inefficient because the amount of static code needed in each processing cell can vary dramatically. To eliminate uniform layout, ACRES branch units support non-uniform code layout by indirectly specifying the target of a branch using a table. A branch target provides the name of a target blocklet and a table translates the blocklet name into distinct physical RAM addresses within each recipient instruction unit. Since a separate table is provided in each processing cell, the program layout can vary from cell to cell.

A processing cell is either idle, or it is running. When idle, the processing cell remains idle until a branch is received. When running, a processing cell continues to execute straight-line code, beginning at a current location, until a branch is received. When a branch is received, a global name is translated to a local address within the processing cell and flow of control is transferred to the branch target where execution begins. An error results when a processing cell receives two conflicting branches on the same cycle.

Figure 5 presents an example of DVLIW control flow derived from a single execution thread. The example treats a counted loop consisting of start, body, conditional exit, and

end blocks as shown from top to bottom in the top left code panel and the bottom left control flow graph (CFG) panel. The loop is unrolled four ways, using standard unrolling techniques, to provide sufficient parallelism in the top right unrolled code panel. Multiples of four iterations are processed in parallel in the control flow graph shown on the right. Residual iterations are processed with a conventional sequential iteration (not shown).

In the bottom right spatial control flow graph, the loop is placed in blocklets that are distributed across multiple processing cells. The loop is spatially distributed using a network of branch commands sufficient to control execution across the utilized subset of the computational fabric. The start blocklet multicasts a branch to each of four body blocklets. Each body blocklet executes for a statically known number of cycles. The leftmost body blocklet casts a branch to the conditional blocklet which decrements the count and either multicasts a branch to all four body blocklets, or casts a branch to the exit blocklet. The loop can also be software pipelined to allow execution overlap between successive groups of four unrolled loop iterations and to further enhance parallelism [40].

<table>
<tr>
<td>

```
i=n;
<start>
do {
   <BODY>;
   i--;
}
while (i>0) <conditional>
<end>
```

</td>
<td>

```
if(I>3) {
   do {
      <BODY>;
      <BODY>;
      <BODY>;
      <BODY>;
      i-=4;
   } while (i>3)
}
<residual iters not shown>
```

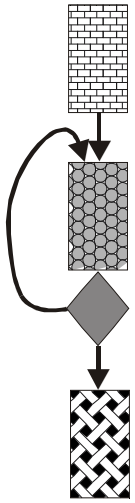
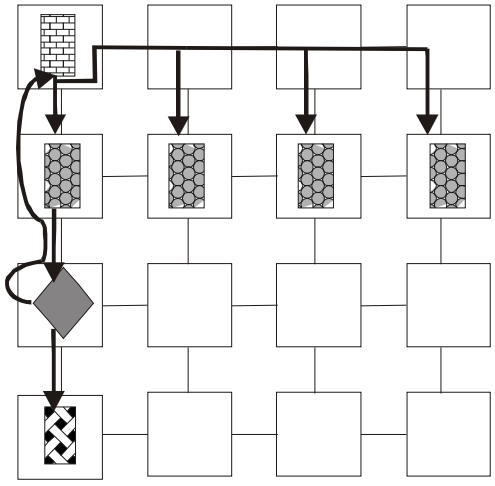
</td>
</tr>
<tr>
<td>Original Code</td>
<td>Unrolled Code</td>
</tr>
<tr>
<td>  </td>
<td>  </td>
</tr>
<tr>
<td>Original loop CFG</td>
<td>Spatial CFG</td>
</tr>
</table>

Figure 5: Spatial flow of control.

While Figure 5 illustrates spatial flow of control, data flow is not shown. An operation executed by a controller within a tile may reference data stored in a memory or register that is local to that tile or, it may reference shared data residing in another tile. In this case, a cross tile access path is defined to support the data transfer. (Further details about inter-tile data transfers are described in Section 5.) In general, the use of shared data references raises questions of synchronization and resource contention. For this simple spatially distributed loop, synchronization is managed in a VLIW-like fashion using static scheduling to ensure that values are produced before they are consumed. If an inter-tile data reference requires more than one cycle, the static schedule must provide the requisite time-of-flight in the form of an explicit latency between the producer and the consumer. Resource contention can also be managed using static scheduling techniques. For example, when multiple operations reference a single memory port, a static schedule can ensure that multiple references do not both utilize the same port on the same cycle.

The DVLIW branch described above is used as a uniform dispatch mechanism for both single and multiple thread computation. The discussion above explains how multiple processing cells can collaborate as a single VLIW. A set of collaborating processing cells is called a VLIW instruction cluster. Processing cells within each cluster collaborate to execute a single VLIW program.

The ACRES system can be configured as a multiprocessor consisting of one virtual processor per VLIW instruction cluster. Branches within a cluster are broadcast to processing cells within that cluster but not to other clusters allowing each cluster to act as a separate processor. While most branches are transmitted within clusters, a processing cell may also transmit a branch to one or more processing cells within another cluster. The effect of this branch is that execution will begin within the remote cluster. This dispatches a new thread to begin within a distinct virtual processor.

Virtual processors may share data held in registers or memory within the ACRES fabric. Issues of synchronization and resource contention between virtual processors can be addressed by constructing synchronization primitives needed to support multithreaded flow-of-control. Signals, semaphores or other synchronization operations are implemented to coordinate multiple threads.

4.3 Remote Branch Architecture Usage Examples

This section illustrates with two examples how distributed code mapped onto an ACRES computing fabric uses ACRES remote branch architecture for coordination. The first example is a trivial piece of code that shows how a single thread of execution is split across three processing cells operating as a VLIW-style cluster. The second example illustrates both VLIW and MIMD style execution, and transitions between them.

4.3.1 Example 1: VLIW-style Cluster

Figure 6 shows a code fragment with four basic blocks numbered 1 to 4 that are mapped to three processing cells pc1, pc2 and pc3 shown in Figure 7. For simplicity, the example employs basic blocks instead of hyperblocks as the unit of compiler manipulation.

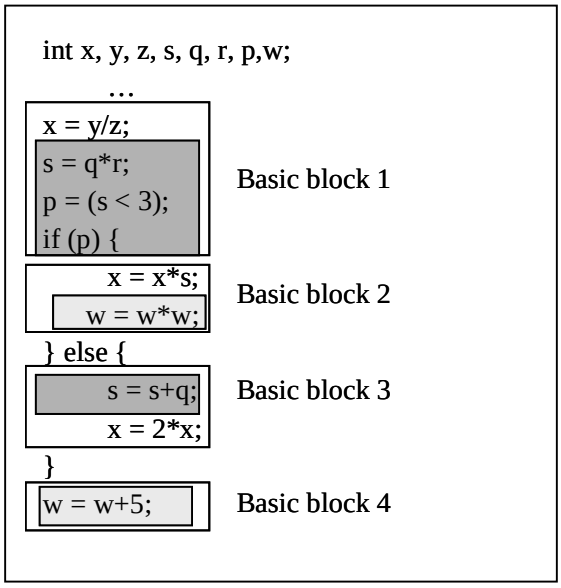


Figure 6: Example code that will be mapped to three processing cells.

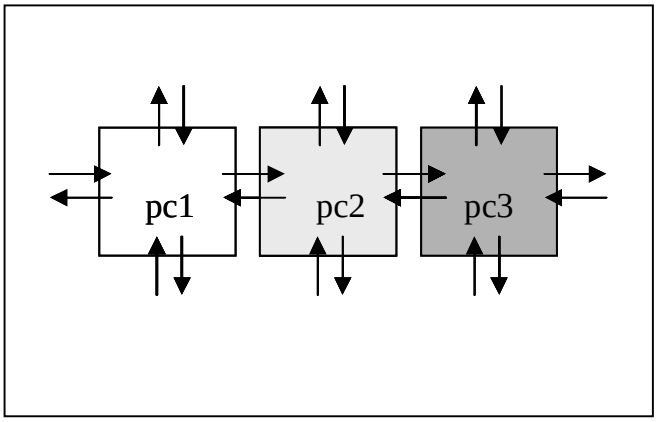


Figure 7: Three directly interconnected processing cells used to host the code in Figure 6.

Figures 6 and 7 are shaded to indicate which code statements are assigned to each processing cell – the darkly shaded statements will execute on pc3, the lightly shaded ones on pc2, and the un-shaded ones on pc1. Figure 8 illustrates in greater detail how the code fragment in Figure 6 is implemented on processing cells pc1, pc2, and pc3. In Figure 8, instruction blocks bb1 of processing cell pc1, bb1' of processing cell pc2 and bb1'' of processing cell pc3 correspond to basic block 1. Similarly, blocks bb2, bb2', and bb2'' correspond to basic block 2; blocks bb3, bb3', and bb3'' correspond to basic block 3; and blocks bb4, bb4', and bb4'' correspond to basic block 4.

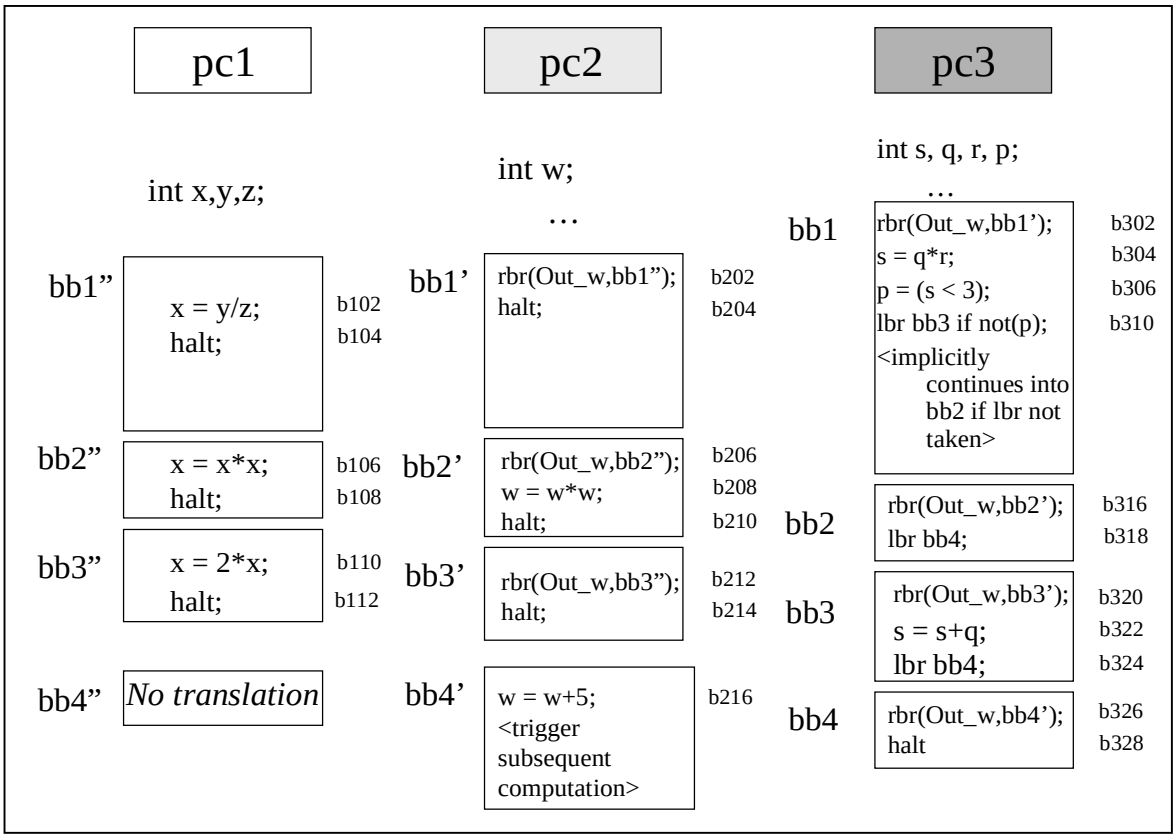


Figure 8: Mapping of code from Figure 6 onto the three processing cells in Figure 7.

Figure 8 shows that, besides the original code in Figure 6, cells pc1, pc2, and pc3 have additional code for coordinating execution. For example, the "halt" instruction on lines b104, b108, b204, etc., the "rbr" (remote branch) instruction on line b202, b206, b302, etc., and the "lbr" (local branch) instruction on lines b310, b318, etc., are "control coordination" code.

A "halt" instruction halts execution of the program on a processing cell. A "rbr" instruction sends a remote branch command to one or more processing cells, with a virtual branch target tag that identifies the code to be invoked at the destination(s). The "rbr" instruction may also carry additional data. In this example, because processing cell pc3 controls execution of processing cells pc2 and pc1 when execution is initiated for basic blocks 1, 2 and 3, processing cell pc3 is the origin of several chains of remote branch commands. For example, processing cell pc3 issues the command "rbr (out_w, bb1')" on line b302, and sends this command to processing cell pc2 on its west. The parameter "bb1'" in the command specifies the address for the target processing cell to start executing at virtual address or block "bb1'". The "lbr addr" instruction is a local branch instruction that transfers local execution to an address "addr" in the same processing cell.

The relative timing of code running on processing cells pc1, pc2 and pc3 are carefully planned so that the resulting execution respects latency constraints. For example,

external remote branch commands arriving at a target are arranged so that they do not prematurely terminate execution at the target processing cell. In order to simplify explanation, however, detailed timing consideration is not explicitly mentioned in the following description.

Execution of the example in Figure 6 starts at processing cell pc3. However, since the instruction "x = y/z" is to be executed on line b102 in processing cell pc1, processing cell pc3 issues appropriate commands for that to happen. Processing cell pc3, on line b302, issues a command "rbr" to processing cell pc2, which, in turn, sends another "rbr" command on line b202 to processing cell pc1. Because processing cell pc2 is on the west side of processing cell pc3, processing cell pc3 specifies the "out_w" parameter for the command to be sent to the cell on its west. Processing cell pc3 also specifies the virtual branch target address bb1'. Processing cell pc3 then continues to execute instructions "s = q * r", "p = (s < 3)", and "lbr bb3 if not(p)", on lines b304 to b310.

For basic block 2, processing cell pc3 issues appropriate commands so that the instruction "x = x * x" is executed by processing cell pc1. Execution on processing cell pc3 enters block bb2 when the conditional local branch on line b310 is not taken, and local execution proceeds sequentially into block bb2. At address bb2 on line b316, processing cell pc3 issues a command "rbr (out_w, bb2')" to trigger execution at address bb2' on processing cell pc2. At address bb2' on line b206, processing cell pc2 in turns issues a command "rbr (out_w, bb2'') to initiate execution at address bb2'' on processing cell pc1. Finally, the instruction "x = x * x" on line b106 is executed at address bb2'' of processing cell pc1.

Similar to the instructions "x = y/z" and "x = x * x," since the instruction "x = 2 * x" on line b110 is to be executed by processing cell pc1, processing cell pc3 issues appropriate commands for that to happen. Execution on processing cell pc3 is transferred to block bb3 when the conditional local branch command "lbr bb3 if not(p)" finds p to be false, thus resulting in a taken branch. At the branch target bb3, processing cell pc3 sends a command "rbr" to initiate execution at address bb3' on processing cell pc2. Processing cell pc2 in turn issues a command "rbr" on line b212 to trigger execution at address bb3'' on processing cell pc1, which then executes the instruction "x = 2 * x."

In the above example, processing cell pc2 and pc1 do not participate in the local branching decision of processing cell pc3, e.g., when processing cell pc3 issues the commands "lbr bb3 if not(p)" on line bpc1. Processing cell pc2, at address bb1' on line b202, relays information received from processing cell pc3 to processing cell pc1, without acting on the received information. In addition, in some cases, processing cell pc2, as a receiving processing cell, also performs its own tasks, *e.g.*, executing the instruction "w = w * w" on line b208, after initiating the remote branch command "rbr" on line b206. Processing cell pc3 stops its execution by issuing the "halt" command on line b328.

Block bb4'' showing no instruction indicates that processing cell pc1 has no role in this block bb4''. As a result, performing a lookup table in processing cell pc1 for block bb4'' will result in a failure to find a match, in which case the remote branch command has no effect.

The above example uses static timing analysis and schedule, as opposed to dynamic synchronization, to ensure that execution of a block is completed before a new remote branch command arrives. For example, execution of block bb1'' of processing cell pc1 is complete before the command to trigger execution of block bb2'' on the same processing cell pc1 arrives. In coming up with an appropriate schedule, the compiler may have to delay initiating a remote branch command to ensure that it does not arrive prematurely.

The above example also illustrates the splitting of a block into blocklets. Where the original code in Figure 6 comprises of one block, the actual implementation utilizes up to three processing cells, with each processing cell executing one or more blocklets. Generally, a blocklet is triggered by an arriving remote branch command and terminates at a "halt" instruction. Thus Figure 8 shows code representing one blocklet on processing cell pc3, 4 blocklets on processing cell pc2, and 3 blocklets on processing cell pc1.

It should be pointed out that the above example uses direct connection between neighboring processing cells for transmission of remote branch commands. An alternative supported by ACRES is to use a special branch transmission network that connects a processing cell to every processing cell. The transmission scope of each branch command is specified in the command and implemented by switches within this transmission network.

In this simple example, branch commands are passed, but no data is transferred from cell to cell. Generally, the ACRES architecture allows data to be passed between cells both in registers and through memory. The data transport described in section 5 can be used to transfer data from a register within one cell to a register in another cell. Similarly, the memory transport discussed in Section 6 allows non-local data transfer to RAMS. In both cases, data is transferred using paths that are not used for branching. It is also possible to design branch-transport architectures that transport branches as well as data. This will not be described in this document.

4.3.2 Example 2: MIMD and VLIW Modes of Operations

The following example illustrates MIMD and VLIW modes of operations and transition between the modes. Figure 9 shows a parallel program that first sorts two arrays A and B, and then multiplies corresponding elements of the two arrays, leaving the results in a third array C.

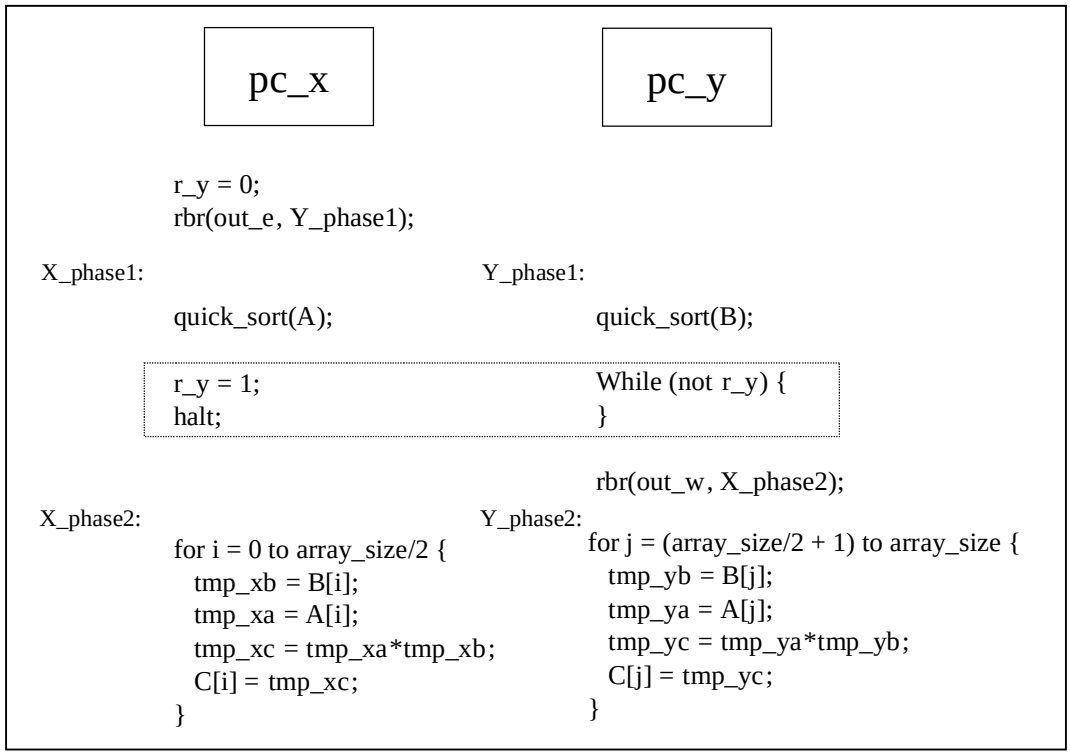


Figure 9: A parallel program for illustrating both VLIW- and MIMD-style execution on ACRES, and transition between the two styles.

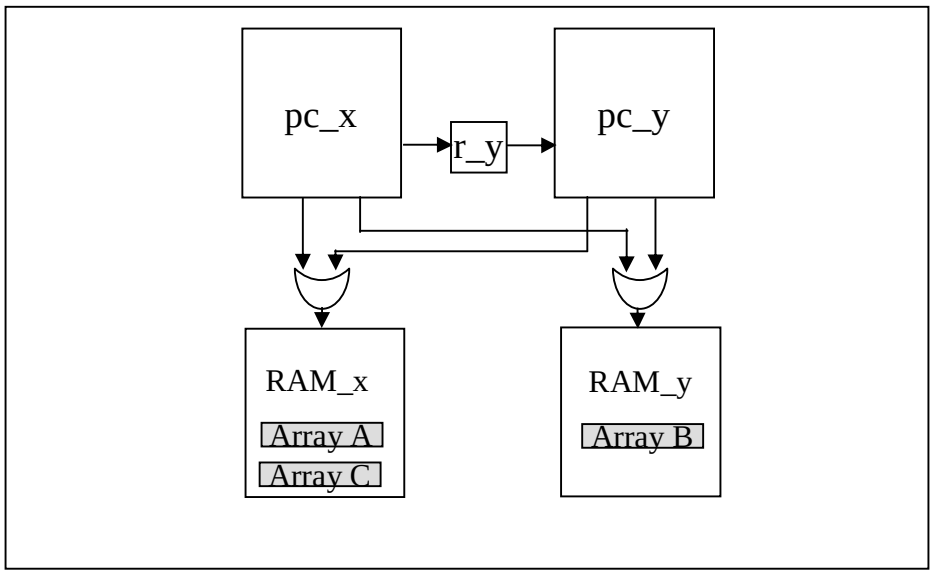


Figure 10: An application specific ACRES virtual architecture generated for the computation in Figure 9.

Figure 10 shows an ACRES virtual architecture generated for the program of Figure 9. It includes two processing cells pc_x and pc_y, two virtual RAMs RAM_x and RAM_y, and a synchronization register r_y. Processing cells pc_x and pc_y share virtual RAMs

RAM_x and RAM_y. Each RAM has one port capable of performing both read and write operations, and, in each clock cycle, each RAM performs one memory operation. Processing cells pc_x and pc_y resolve potential RAM access conflicts either through static scheduling when they operate under the VLIW mode, or through dynamic synchronization when they operate under the MIMD mode.

Processing cells pc_x and pc_y also share synchronization register r_y, which has a read port and a write port. The read port is accessed by processing cell pc_y, and the write port by processing pc_x. Each port can be accessed once in each cycle. A write in cycle *t* is visible to a read performed the next cycle, e.g., $t + 1$. A read that occurs in the same cycle *t* gets the value previously stored in the register.

Each processing cell has private resources, including local registers for storing temporary values. In Figure 9, the temporary values have variable names prefixed by "tmp_". In addition, each processing cell has private functional units, *e.g.* multipliers, for implementing the needed computation.

When execution begins, arrays A and C are stored in virtual ram RAM_x and array B is stored in virtual ram RAM_y. Initial execution occurs on processing cell pc_x. Computations happen in two phases. Phase one is triggered when processing cell pc_x sends a remote branch command to processing cell pc_y to trigger execution at Y_phase1, while execution on processing cell pc_x continues sequentially into X_phase1. Phase one execution occurs in an MIMD mode, with each of processing cells pc_x and pc_y performing a quick-sort on arrays A and B, respectively. Processing cell pc_x accesses virtual ram RAM_x while processing cell pc_y accesses virtual ram RAM_y during this phase. Consequently, there is no conflict for accesses to the two virtual rams.

At the end of phase one, processing cells pc_x and pc_y perform dynamic synchronization using synchronization register r_y, which is initialized to zero before the beginning of phase one execution. When processing cell pc_y finishes phase one execution, it repeatedly checks the value of register r_y until it finds a one in register r_y. Concurrently, when processing cell pc_x finishes phase one execution, it writes a one into register r_y and halts. When processing cell pc_y finds a one in register r_y, it knows that processing cell pc_x has completed phase one. Processing cell pc_y then initiates phase two execution by sending a remote branch command to processing cell pc_x.

Phase two execution adds corresponding elements of arrays A and B to produce elements of array C. Processing cell pc_x performs the multiplication for elements in the first half of the arrays, while processing cell pc_y performs the multiplication for elements in the second half of the arrays. Both cells pc_x and pc_y write their results directly into array C. The second phase execution occurs under VLIW mode and takes advantage of static scheduling of VLIW mode to statically orchestrate the memory accesses.

Table 1 shows a possible schedule for accessing the virtual RAMs. To avoid clutter, looping details, such as loop index increment, and loop termination testing is left out. In that schedule, there is at most one memory access to RAM_x in each clock cycle. Similarly, there is at most one memory access to RAM_y in each clock cycle.

Table 1: A RAM access schedule for the code of Figure 8 executing on the virtual architecture shown in Figure 9.

Time	pc_x	pc_y
t	Read B[i]; (RAM_y)	
t+1	Read A[i] ; (RAM_x)	Read B[j] ; (RAM_y)
t+2	(perform multiply)	Read A[j] ; (RAM_x)
t+3	write C[i] ; (RAM_x)	(perform multiply)
t+4	Read B[i+1]; (RAM_y)	write C[j] ; (RAM_x)
t+5	Read A[i+1] ; (RAM_x)	Read B[j+1] ; (RAM_y)
t+6	(perform multiply)	Read A[j+1] ; (RAM_x)
t+7	write C[i+1] ; (RAM_x)	(perform multiply)
t+8	Read B[i+2]; (RAM_y)	write C[j+1] ; (RAM_x)
t+9	Read A[i+2] ; (RAM_x)	Read B[j+2] ; (RAM_y)
t+10	(perform multiply)	Read A[j+2] ; (RAM_x)
< and so on until loop terminations>		

To achieve the schedule in Table 1, the scheduler counts the cycles of various operations on processing cells pc_x and pc_y, starting from the point where processing cell pc_x initiates the command rbr(out_e, Y_phase2). That is a common reference starting time from which two sequences of actions and their timings are followed. No-ops are inserted in the code of processing cells pc_x and pc_y as appropriate so that the resulting code exhibits the relative timing as indicated in the schedule of Table 1.

The example of Figure 8 illustrates MIMD mode execution and how it is started from a degenerate VLIW mode of execution involving only one processing cell. The example then shows how the MIMD mode of execution ends through dynamic synchronization, and then transitions into a VLIW mode of execution involving multiple processing cells,

in this case two processing cells pc_x and pc_y. Finally, the example also illustrates how VLIW mode of execution exploits static scheduling to ensure that concurrent accesses to shared resources, the RAMs in this example, do not result in any conflicts.

5 Pipelined, Time-multiplexed Interconnect

ACRES introduces a new approach to data transport for reconfigurable systems that improves performance while lowering total wire costs. It uses an innovative architecture for data-transport that is different from traditional reconfigurable device transport architecture, as well as a novel design procedure to take advantage of this unconventional architecture. Specifically, ACRES exploits pipelining of communication paths to reduce the impact of long communication paths on system cycle time. It further multiplexes each data transport segment to better utilize data transport wires.

While neither pipelining nor time-multiplexing are new ideas on their own, this aspect of the ACRES platform is unique in that the pipelining and time-multiplexing decisions are taken only after the overall design undergoes a coarse placement, or what we call spatial planning. In so doing, the actual pipelining along a communication path is driven by knowledge of the spatial locations of the communication's data source(s) and destination(s). Similarly, segment-level time-multiplexed sharing opportunity is derived from the chosen spatial plan. Of course a good spatial plan should, among other things, take into account opportunities for data transport sharing, their impact on performance and perform placement in a way that facilitates sharing and improves performance.

Another unique departure from traditional design procedures under this approach is that final schedule of operations on the machine is only determined after both the spatial planning and the finalization of the data transport architecture. This confers two benefits. Firstly, by performing scheduling of other operations this late in the design flow, the scheduler can make use of the actual transport latency information that comes out of pipelining the data transport path for a known spatial plan. The relative ordering and timing of operations can then be adjusted to accommodate data transport latencies. Secondly, the scheduler is leveraged to schedule data transfers on each shared data transport segment.

In contrast, traditional chip design flow is phased, with abstractions between the phases that inhibit the approach advocated in ACRES. A very common chip design approach is to specify design at the register transfer level (RTL) in a hardware description language such as Verilog or VHDL. An RTL design is then fed through a hardware synthesis tool followed by place and route tool. Such a design flow is common for both ASICs and FPGAs. In such a design flow, an RTL design specifies the operation of the hardware at clock cycle level, *i.e.* what should happen in each clock cycle. Subsequent steps in the design flow have very little flexibility in altering what happens in a clock cycle, or inserting additional clock cycles. Any such adjustments, such as classic retiming [39, 42, 51], are very local and have to be done under very stringent restrictions of preserving execution order and the relative latency (in number of clock cycles) of different paths that diverge and subsequently merge.

By the time the traditional design flow reaches place and route, a design is represented at a very low level of abstraction. Data transport requirements are represented at the level of unshared wires, which are the entities routed by the place and route tool. Any decisions to pipeline or time-multiplex data transport, and the scheduling of operations in the machine, are all frozen in the RTL specification. With the traditional design and

synthesis flow, it is not possible for a compiler to automatically use placement information to revise pipelining and multiplexing decisions. When an operation schedule is fixed in the RTL it becomes very difficult to accommodate changes to interconnect latency in subsequent phases of the design flow.

5.1 An Example ACRES Computing Fabric

This section describes the ACRES computing fabric (version RC-1) used to illustrate inter-tile pipelined and multiplexed interconnect. An RC-1 reconfigurable chip contains various configurable computation, memory, interconnect and control elements. Examples of possible components include look-up tables (LUTs) that can be programmed to implement any combinational logic function, special arithmetic support such as carry chains, specialized (hardwired) functional units such as multipliers, reconfigurable interconnect such as wire segments connected via programmable switches, and memory blocks. Such components are common in field programmable gate arrays (FPGAs), and in more computation-oriented reconfigurable architectures aimed at the DSP market. It is expected that depending on the application domain, other special components may be included; for example, multipliers or content addressable memories.

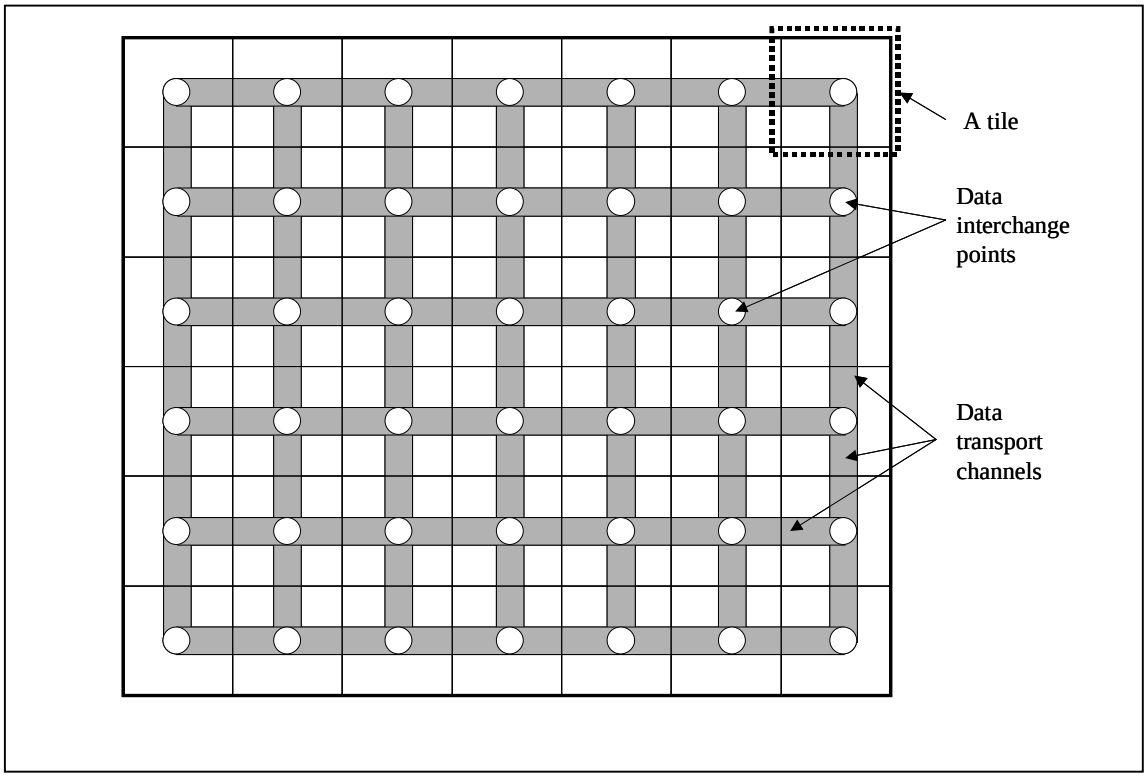


Figure 11: An RC-1 chip consisting of a 6x7 array of tiles. The circles represent data interchanges, interconnected by transport segments marked in gray.

A unique aspect of RC-1 is the division of an RC-1 chip into tiles as shown in Figure 11. Typically, each tile contains a mix of various components and is thus heterogeneous. In a typical member of the RC-1 family, the same tile is replicated many times in, say, a two dimensional grid of tiles. A tile is sized so that typical data transport within the tile is

expected to stay within a target system clock cycle time, while data transport that goes beyond a tile is likely to exceed that clock cycle time. As will become clearer when spatial compilation is described below, the tile abstraction provides a convenient coarse-grain unit of placement for optimizing data transport.

Another interesting aspect of RC-1 is the inter-tile data transport support. As shown in Figure 11, the 2-dimensional array of tiles in an RC-1 chip is complemented with a 2-dimensional network of data transport channels. Each data transport channel contains data transport segments. The end-points of the data transport segments are interchanges, where data can enter or leave the inter-tile data transport network. Interchanges are also data switching points; for example in Figure 11, each interchange permits data coming in from one direction to go onto data transport segments in any direction. Data can also be stored into temporary registers and thus delayed for multiple clock cycles at each interchange. The tile abstraction conveniently serves as a reference framework for positioning data transport interchanges.

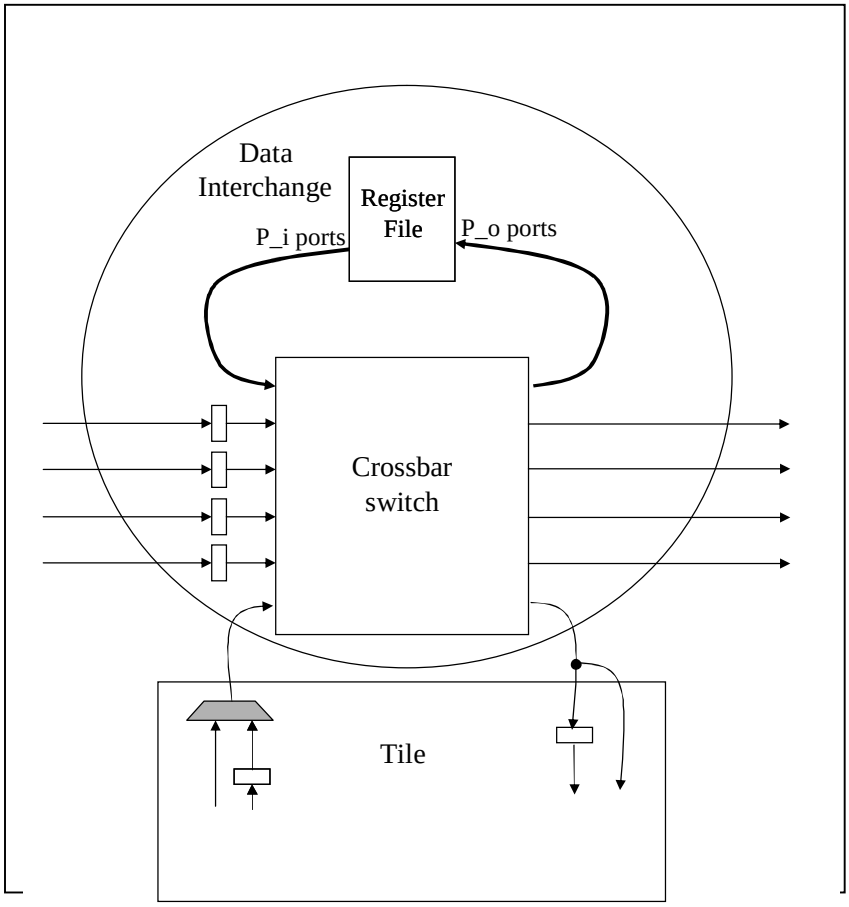


Figure 12: A data interchange.

Figure 12 illustrates the data path of a simple data interchange design. The data paths are assumed to be of a bit-width suitable for the design, for example, 4-bit or 8-bit nibbles. Multiple paths can be bundled together, or a path may be reused multiple times, to support wider paths according to application needs. At the center of the design is a crossbar switch that allows data from any of the four in-coming data segments, the local

tile, or the register file within the interchange to be sent to any one (or multiple) of the out-going data segments, or into the tile.

The register file in the data interchange supports P_o write ports and P_i read ports. P_o must at least be 1, but provides the greatest scheduling flexibility when P_o equals the number of in-coming data segments. When P_o is smaller than this number, only a limited number of in-coming data can be buffered into the register file each clock cycle, for multiple-cycle delay at the data interchange. Similarly, P_i must at least be 1, and has the greatest scheduling latitude when P_i equals the number of out-going data segments.

Data leaving the tile may optionally be latched into a register before entering the crossbar switch. Similarly, data arriving into the tile may optionally be latched. While Figure 12 indicates only 1 switch port going into the tile and another leaving the tile, a design may choose to have a larger number of either port types.

A typical design supports multiple data transport segments in each data transport channel. Such a design's interchange may increase the number of crossbar switch inputs and outputs to accommodate these additional segments. Alternatively, the design may employ multiple instances of the interchange design of Figure 12, each connecting a subset of the data transport segments. The latter design is more scalable as it requires smaller crossbars, but partitions the segments at each data interchange into sets. In such a design, there is much flexibility in wiring the segments between data interchanges. For example, a design may partition the data segments across the entire chip, essentially overlaying multiple independent data transport networks on the chip. Alternatively, the wiring of data segment between interchanges can enable crossing from one partition set to another to prevent the chip-wide segregation of data transport segment into non-overlapping partitions.

5.2 Data Transfer Control

Two different control methods are used to manage data propagation through the inter-tile interconnection network. Under the *programmatic mode of control*, data transfers through the inter-tile data transport infrastructure is statically scheduled to avoid conflicting access to common resource. This is in contrast to dynamic arbitration that performs allocation decisions at run-time, based on requests initiated at run-time by parties interested in using a common shared resource.

Under programmatic control, a locally sequenced program controls each crossbar and register file ensemble of the sort illustrated in Figure 12. We use the word program loosely to apply to both program execution mechanisms using instructions drawn from memory, as well as state-machines. State-machines may be constructed out of reconfigurable logic so that they are application specific. A characteristic of programmatic control is that a local thread of control provides control continuity in the absence of external commands, and possibly using locally stored state. Furthermore, when making its control decisions, the programmatic control need not consider the content flowing through the data transport segments.

A locally sequenced control program specifies on a clock cycle by clock cycle basis how its data interchange operates. For example, for the data exchange illustrated in Figure 12,

its control program specifies how the crossbar switch is configured, *i.e.* which input is connected to which output(s), whether any data should be stored into the register file, and if so, it names the destination register(s). The program also names the registers, if any, to read out of the register file. The program may also control whether each of the registers on the input to the crossbar switch latches in a new value in that clock cycle, or whether it holds the previously latched value.

To facilitate flexible local control over execution path, the locally sequenced control program is equipped with the ability to perform local conditional branches. A typical design includes conditional branch instructions, general arithmetic and logical operations to compute condition variables, and scratch registers. This enables it to, for example, locally manage loop counters, counting the number of iterations to locally determine when a loop should terminate.

A typical data transfer path involves multiple data interchanges. For a transfer to occur successfully over such a path, the programmatic controls in the data interchanges along the path work in tight synchrony to ensure that the correct operations happen in each cycle. The control of a data interchange upstream has to put out the correct value, while the data interchange downstream has to route/switch it, or store it into a register as appropriate. This tightly synchronized collaboration is statically choreographed, and encoded into the control programs running in the data interchanges.

To enable this local programmatic control mechanism to collaborate with the broader chip, the programmatic control mechanism allows externally initiated branching of its local thread of control. A remote-branch mechanism described in Section 4 is employed to achieve this remotely initiated control flow change. Essentially, the programmatic control at a data interchange receives branch commands sent from other control entities on the chip. Upon receiving a remote branch command, the programmatic control transfers its local thread of control to the branch target specified by the branch command, and continues execution from there.

A common situation has the source of a branch command multicast the branch instruction to all the data interchanges collaborating on a computation, causing them to branch to code sequences that have been statically planned to work in harmony. Thus, all the data interchanges on a data transfer path are among the destinations of a branch instruction.

While the simplest design supports only one programmatic control thread at each data interchange, some designs support multiple threads while still using static conflict resolution. The latter has the benefit of enabling computations that have different control flow, *i.e.*, they have different branching conditions, to share a data interchange. Static resource partitioning among otherwise independent threads can be achieved by either space or time partitioning, or both.

Time partitioning divides time into recurring sequences of time slots, with each time slot assigned to a thread. For example, thread 0 gets to use a data interchange resource on odd cycles, while thread 1 gets to use it on even cycles. Alternatively space partitioning allocates different resources within the data interchange to each thread. For example, the crossbar switch of Figure 12 may be space partitioned so that inputs 0 and 1, and outputs 0 and 1 are used by thread 0, while inputs 2 and 3, and outputs 2 and 3 are used by thread

1. Each thread of control clearly indicates the resources over which it is asserting control for each cycle, so that merging overall control from multiple threads is straight forward. Because resource allocation is done statically, dynamic arbitration and allocation is unnecessary.

Remote branch commands propagate through the inter-tile interconnection network under a *tag-based switching* control scheme similar to that used in packet switched networks. Unlike programmatic control, tag-based switching decides how a unit of data, typically referred to as a packet, is treated as it flows through a data interchange based on the value of a tag carried in the packet.

One common way of implementing the tag-based switching mechanism is to employ lookup tables. The tag carried on each packet is used as the key for a lookup in this table. Successful lookup yields information such as which output port, or ports of the data interchange that packet should be forwarded to.

In a simple implementation of tag-based switching, a packet propagating through data interchanges encounters a constant delay in each of the data interchanges. Some designs may find it useful to augment this base design with programmable delay, which enables a packet to be delayed at a data interchange by a pre-determined number of clock cycles beyond the constant delay. This is particularly useful when static allocation is used to avoid resource contention. The additional delay is determined through a combination of the tag carried by a packet, and values programmed into the lookup table at a data interchange. The lookup table is augmented to hold, for each tag entry, additional information such as (i) the number of cycles of additional delay to add to the constant transit delay through the data interchange, and (ii) the base buffer register name.

When a packet carries data spanning m clock cycles, and n cycles of delays are needed, $\min(n,m)$ contiguously-named registers from the register file are allocated to buffer the data undergoing the delay. The base buffer register name is the name of the first register among these $\min(n,m)$ registers.

A programmable delay, tag-based switching implementation includes mechanisms for reading out data after the desired delay, and sending it out the desired output port. This is implemented by associating with each register file read port a pending action queue that keeps track of read actions to be performed in some future time. As data is buffered into the register file at register r for an n cycle delay, a pending action to read out the data n cycles later from register r is queued with a read port. In each clock cycle, the pending action queue is advanced, and a pending action that has become current is performed. A similar queue is maintained for pending actions at the crossbar switch, with the slight complication that multiple actions using different resources may be performed at the crossbar each clock cycle.

Programmatic control is efficient when the control follows a locally determined path for a significant amount of time after it is triggered by an external source. An example is the execution of a counted loop body. In comparison, tag-based switching is appropriate for less repeated, externally determined transfers. Examples include data transfers in non-loop sections of execution, and remote branch control commands.

An implementation may choose to have separate program controlled and tag-based switching networks. Alternatively, the two propagation control mechanisms can be combined in one network in order to share hardware resources. In a combined design, the programmatic switch control mechanism has to be augmented to support tag-based switching. Many different hybrid design choices are possible and their relative merits are the subject of future work.

5.3 A Few Transfer Examples

Figure 13 illustrates how a simple expression may be mapped onto an RC-1 chip and makes use of the data transport mechanism. First let us consider the expression in isolation. The expression involves loading two values A and B, multiplying the two values together, and then adding the result of the multiplication to the value in a register. Taken in a larger context, this expression may be part of an inner-product loop that loads values from two arrays A and B, and then performs a multiply and accumulate (MAC).

In Figure 13, the arrays A and B are mapped to tiles (2, 2) and (4, 2) respectively, while the operations in the MAC are all mapped to tile (4,3). (We use a co-ordinate system to refer to tiles and data interchanges. The first number in the coordinate tuple refers to the row, while the second refers to the column.) Such a floor plan may be motivated by how other parts of the computation use the arrays A and B.

In order to perform the desired computation, values of arrays A and B have to be transported from their sources to the tile (4, 3). Let us assume that after the values of arrays A and B are read, they are registered before leaving the respective tiles. Furthermore, for simplicity, let us assume that the inter-tile data transport has only one data segment going in each direction in each channel. Also assume that the data segment width is exactly the one required for the values of A and B. Lastly, we assume that there is only 1 port leaving each tile and 1 port arriving at each tile from the tile's data interchange.

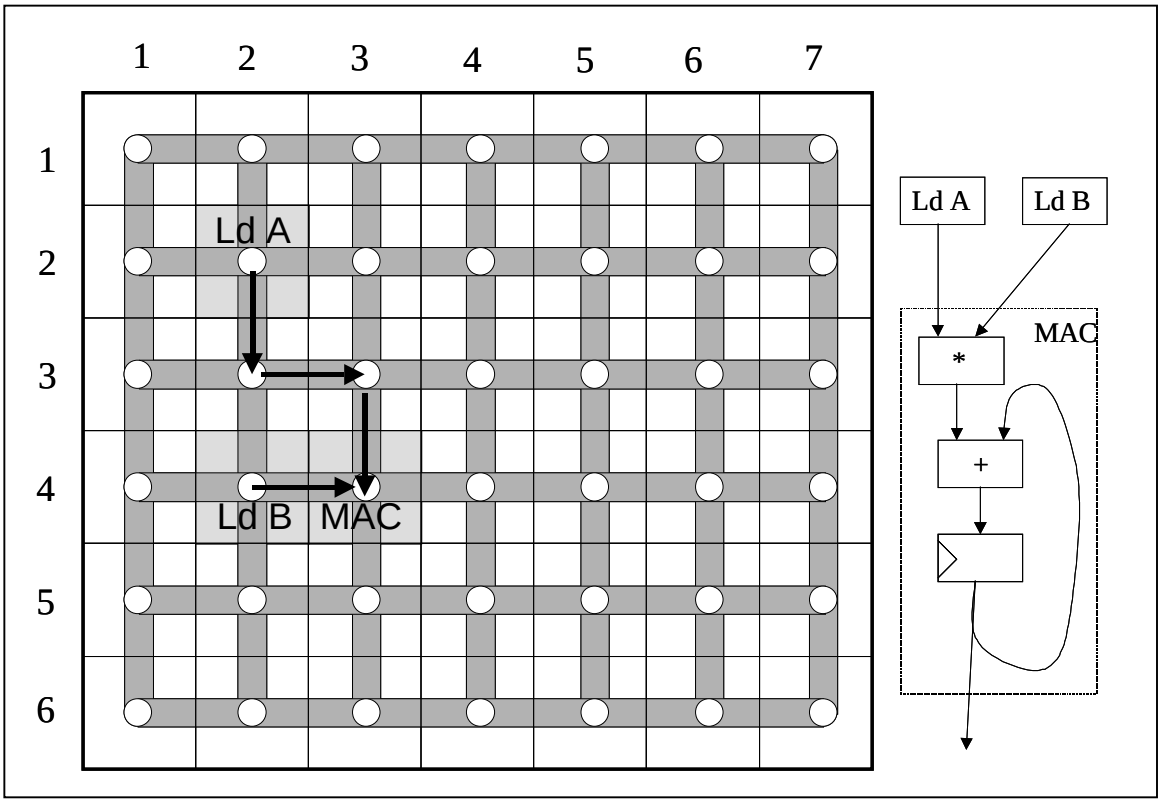


Figure 13: A simple data transfer example.

First, let us consider one instance of the loop body in isolation. One possible data transfer schedule is shown in Table 2.[1] By clock cycle 5 (just beyond what is shown in the table), both Va and Vb are in tile (4, 3) and the MAC can begin.

Table 2: A data transport schedule for the example in Figure 13.

Time	Activities	
0	In tile (2,2), Ld A and latch value Va into local register.	
1	Value Va copied through data interchange (2,2) to data interchange (3,2).	In tile (4,2), Ld B and latch value Vb into local register.
2	Value Va copied through data interchange (3,2) to data interchange (3,3).	Value Vb copied through data interchange (4,2) to data interchange (4,3).

[1] In tabulating schedules here and in subsequent examples, we have chosen to focus on data transfer and gloss over the computations.

3	Value Va copied through data interchange (3,3) to data interchange (4,3).	Value Vb copied through data interchange (4,3) into tile (4,3).
4	Value Va copied through data interchange (4,3) into tile (4,3).	

Next, we consider the expression together with the loop enclosing it. Assuming that the MAC can initiate a new operation as frequently as once every two cycles, the inner-product loop can be software pipelined with an initiation interval (II) of 2, *i.e.* every other clock cycle, a new iteration of the loop is initiated. (Imagine taking the above table and concatenating additional copies that are each offset by 2 cycles.) The fact that there is only one data port going from the inter-tile network into tile (4, 3) prevents the II from going any lower, even if more MAC units are added to tile (4, 3).

If a programmatic mode of control is used to control the data transfer of this loop body, each data interchange is sent a remote branch, using the tag-based control scheme, at the start of the loop to get the correct program for this loop body started. With synchronous, predictable timing across the system, these data interchanges then cooperate seamlessly to implement the desired schedule. Each programmatic control keeps a local loop count to know when the loop should terminate.

The example is enhanced in Figures 14 and 15 to illustrate the time multiplexed use of a link, allowing the shared use of a wire by multiple operands. Figure 14 augments the example with another expression in the same loop body. The expression loads from two arrays C and D, performs a division, and then stores the result in an array E. Assume that spatial planning placed arrays C, D and E in tiles (2, 2), (4, 5) and (3, 5) respectively. The division is performed in tile (3, 5). A possible way to route the data transport is as shown by the red arrows of Figure 12. Except that the values of arrays A and C contend for the tile-to-data transport port in tile (2, 2), the two expressions utilize independent resources.

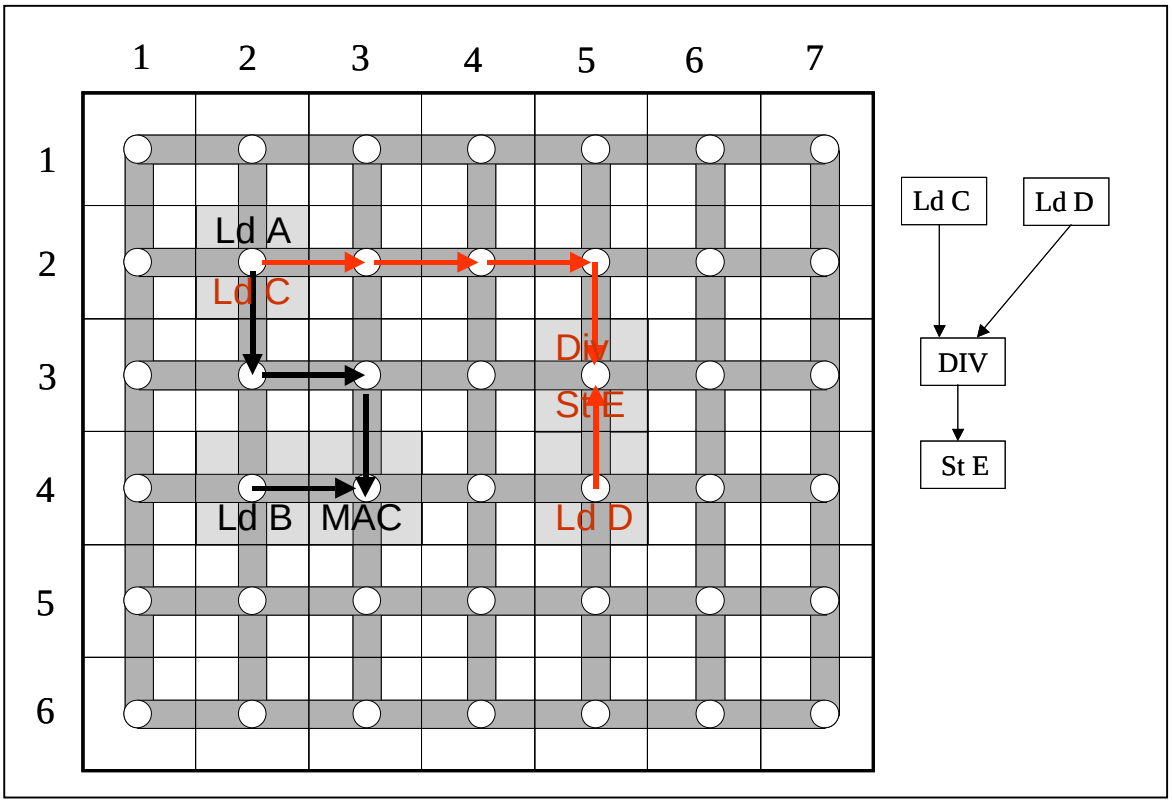


Figure 14: A more complicated data transfer example.

Table 3 shows a possible inter-tile data transfer schedule. The timing of activities in the 2nd and 4th activities column has some slack in that they can be delayed by up to two clock cycles without changing the overall time taken to execute an instance of both expressions. The scheduler can take advantage of this if other computation requires the use of shared resources and prefers one choice or the other. As before, in the context of a loop with these two expressions in the loop body, a software pipelined loop with initiation interval of two can be sustained, assuming the divider can sustain that throughput rate.

Table 3: A data transport schedule for the example in Figure 14.

Time	Activities			
0		In tile (4,2), Ld B and latch value Vb into local register.	In tile (2,2), Ld C and latch value Vc into local register.	In tile (4,5), Ld D and latch value Vd into local register.
1	In tile (2,2), Ld A and latch value Va into local register.	Value Vb copied through data interchange (4,2) to data interchange (4,3).	Value Vc copied through data interchange (2,2) to data interchange (2,3).	Value Vd copied through data interchange (4,5) to data interchange (3,5).
2	Value Va copied through data interchange (2,2) to data interchange (3,2).	Value Vb copied through data interchange (4,3) into tile (4,3).	Value Vc copied through data interchange (2,3) to data interchange (2,4).	Value Vb copied through data interchange (3,5) into tile (3,5).
3	Value Va copied through data interchange (3,2) to data interchange (3,3).		Value Vc copied through data interchange (2,4) to data interchange (2,5).	
4	Value Va copied through data interchange (3,3) to data interchange (4,3).		Value Vc copied through data interchange (2,5) to data interchange (3,5).	
5	Value Va copied through data interchange (4,3) into tile (4,3).		Value Vc copied through data interchange (3,5) into tile (3,5).	

Lastly, let us consider Figure 15, which does the same computation as Figure 14, but picks a different route for Vc. The example shows that since the data segments between data interchanges (2,2) and (3,2), and between (3,2) and (3,3) are only utilized half the time in Figure 14, these links are also used for Vc's data transport in Figure 15 (and Table 4)without affecting performance.

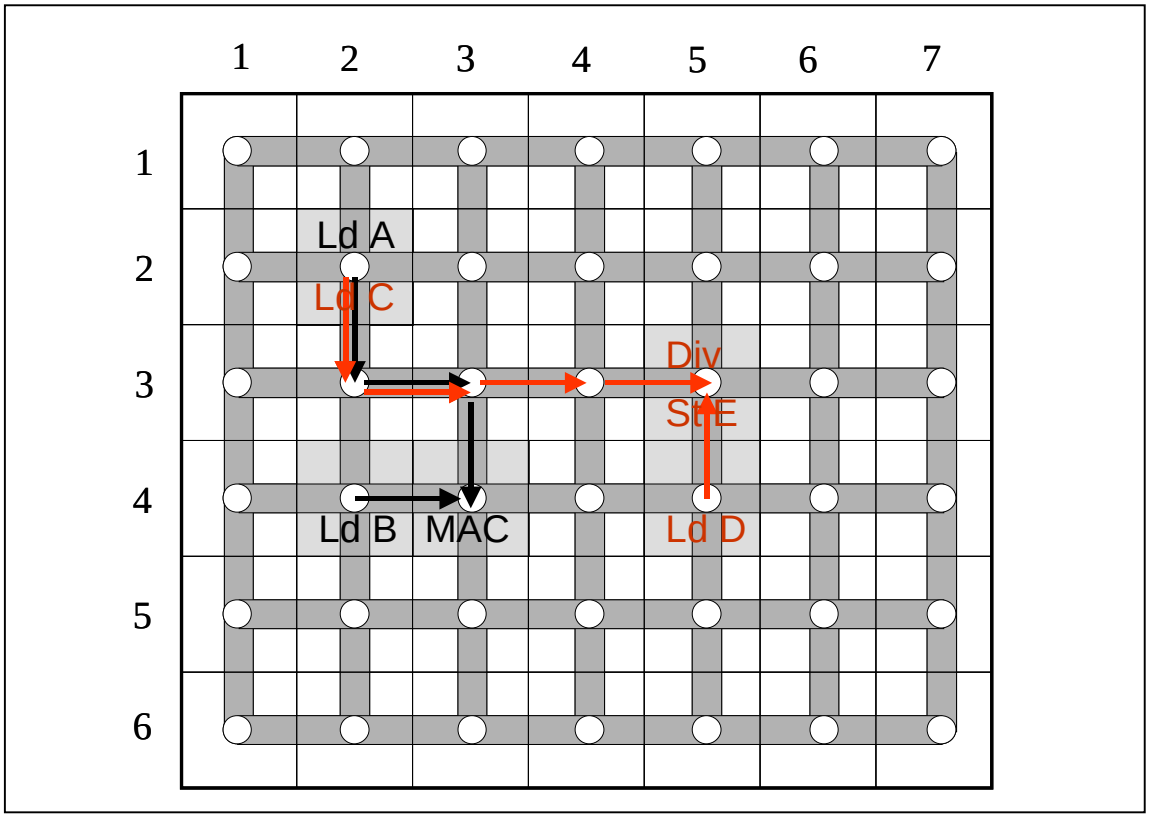


Figure 15: The computation in Figure 14 with different data transfer routes.

Table 4: A data transport schedule for the example in Figure 15.

Time	Activities			
0		In tile (4,2), Ld B and latch value Vb into local register.	In tile (2,2), Ld C and latch value Vc into local register.	In tile (4,5), Ld D and latch value Vd into local register.
1	In tile (2,2), Ld A and latch value Va into local register.	Value Vb copied through data interchange (4,2) to data interchange (4,3).	Value Vc copied through data interchange (2,2) to data interchange (3,2).	Value Vd copied through data interchange (4,5) to data interchange (3,5).

2	Value Va copied through data interchange (2,2) to data interchange (3,2).	Value Vb copied through data interchange (4,3) into tile (4,3).	Value Vc copied through data interchange (3,2) to data interchange (3,3).	Value Vb copied through data interchange (3,5) into tile (3,5).
3	Value Va copied through data interchange (3,2) to data interchange (3,3).		Value Vc copied through data interchange (3,3) to data interchange (3,4).	
4	Value Va copied through data interchange (3,3) to data interchange (4,3).		Value Vc copied through data interchange (3,4) to data interchange (3,5).	
5	Value Va copied through data interchange (4,3) into tile (4,3).		Value Vc copied through data interchange (3,5) into tile (3,5).	

5.4 Early Spatial Planning and Late Scheduling

A key component of ACRES' pipelined, time-multiplexed interconnection technology is its spatial compilation procedure. As described earlier, an important benefit of this technology is to perform spatial planning before committing to the pipelining depth and latency of data transport, and determining time-multiplexed sharing. Computation and data transfer scheduling then follows those steps. This section provides an instance of such a spatial compilation procedure. Many variations are possible and their relative merits remain to be investigated.

Consider a design flow that starts with a C-like program specification of the required computation. (The CAD industry sometimes refers to similar types of specification as behavioral specification.) Such a specification differs from RTL in that it does not specify action on a clock cycle by clock cycle basis. Instead, this specification prescribes a logical, partial ordering of actions encompassing memory loads/stores, arithmetic operations, etc.

A spatial compiler first performs various optimizations and code restructuring to remove redundant computation and to expose parallelism. These steps are very much orthogonal to pipelined interconnection. At the end of that process, we can view the program specification as having been transformed into a virtual architecture. (For an overview of ACRES compilation flow, please refer to Figure 3 of Section 3.3.)

The virtual architecture deals with virtual entities such as operations and memory objects. This is in contrast to hardware consisting of function units that execute the operations, and RAM blocks that contain the memory objects. At some point, the virtual entities have to be mapped to the hardware entities. One approach is to perform this mapping

prior to spatial planning. Spatial planning then performs a coarse placement/assignment of functional units and RAM blocks to tiles, under the constraint that each tile's resource is not over-committed. Tile resources include tile-internal compute, control and memory resources, as well as inter-tile communication bandwidth. Dealing with physical entity enables a relatively concrete estimate of the needed tile resource. The estimate, however, still has to work with some uncertainty. For example, resource for buffering temporary values is not known definitively until after scheduling.

Mapping virtual entities (operations and virtual RAMS) to physical entities is a complex phase of spatial compilation. At the end of this phase, every operation has been mapped onto a physical entity that executes that operation and, every virtual RAM has been mapped onto one or more physical RAMs that support that virtual RAM. Unlike traditional compilation which works with a fixed physical architecture, spatial compilation generates a physical architecture consisting of an application-specific network of physical entities. Even when the total amount of reconfigurable hardware is fixed, there are still many choices in how this fixed set of resources is configured into distinct physical entities and how these physical entities are interconnected.

Design procedures may optimize for different criteria. Some design procedures fix a performance target and minimize the amount of needed hardware while other design procedures optimize performance given fixed hardware resources. A flexible procedure should be capable of generating a set of design points that correspond to a Pareto curve so that different constraints and tradeoffs can be accommodated. Finding good procedures for doing this will be an important area of research.

An initial virtual-to-physical entity mapping strategy for software pipelined loops uses steady state performance target as constraint and generate Pareto design points. The process is repeated with different performance targets to generate the Pareto curve, similar to what is done in PICO to explore design space[1]. Given a software pipelined loop nest and a target initiation internal (II), there are known procedures for deducing the needed number of functional units. The results from such procedures are further adjusted to take into account the relatively high cost of multiplexing and de-multiplexing data-paths on reconfigurable hardware. Where multiplexing hardware cost is comparable to or exceeds the cost of simple functional units, it makes sense not to share them.

For non-loop code for which steady state analysis of software pipelines is not applicable, an initial mapping strategy starts with a design using the minimal number of functional units, and selectively augments the number of functional units to improve performance. The process takes a current design, identifies the FUs that are the most serious bottlenecks, and adds more FUs to assist in processing the work load of bottleneck FUs.

One approach is to augment the compiler intermediate form (IR) with resource constraint arcs. Given a schedule, the ACRES spatial compiler adds a resource constraint arc from an operation that is scheduled later than required by its data and control dependence because of resource contention. The arc goes from the delayed operation to an operation that uses the contended for resource during the times the delayed operation waited. After this has been done for the entire IR, the spatial compiler looks for critical paths through the IR that can be "cut" by removing some resource constraint arcs. Various heuristics can be used to select the most desirable candidates. Furthermore, getting a good

performance estimate of different choices will likely require going through the next few phases of compilation, until a schedule is obtained. Admittedly, this is an open research area with many interesting questions and choices.

A legal spatial plan has to ensure that the computation placed into each tile does not require more resources than available in the tile. Further optimization goals include keeping latency sensitive paths within tiles whenever possible. If it is not possible to keep a latency sensitive path within a tile, the path should be kept minimized, *i.e.* the source and destinations should be as few tiles away as possible. Efficient sharing of data segments and minimizing the spatial spread of a design provides additional objectives for placement decisions.

ACRES' time multiplexing of inter-tile interconnection brings the flexibility of delaying communication, so that a design will not fail to route at the inter-tile level, but instead suffers communication delay degradation. The spatial planner should therefore track inter-tile bandwidth usage as it places computation. To assist this process, the ACRES spatial compiler takes advantage of profiling, as is commonly done by VLIW compilers, to derive execution frequency and data transport bandwidth information for the virtual architecture. Inter-tile bandwidth usage tracking has to include a temporal component, so that bandwidth used in mutually exclusive timing epochs do not sum up. It also has to manipulate "fractional usage". These are necessary to account for the benefits of good multiplexed sharing. More than any aspect of spatial planning, accurately tracking inter-tile bandwidth presents the biggest departure from classic placement and route. Using such novel cost metrics, traditional place and route heuristics such as simulated annealing are used to perform spatial planning.

After coarse placement, virtual data communication (*i.e.* communication specified in the virtual architecture) crossing tile boundaries is routed through the inter-tile data transport infrastructure. Aside from the traditional routing goal of reducing route distances, inter-tile routing may also favor routes with interconnect sharing opportunities. The route of Figure 15 above is one such example.

The following are several possible heuristics for finding data communication that may be good candidates for sharing data transport segments. For software pipeline loops with initiation interval (II) greater than 1, each virtual data-path in the virtual architecture requires only 1/II total bandwidth of a full-width (*i.e.* same width as the virtual data-path) data segment. Hence, for II > 1, different operands within the same software pipelined loop are good candidates for sharing data transport segments.

More generally, if the compiler can determine that two virtual data-paths are never active simultaneously, they can be mapped on to the same data segment without any danger that contention decreases performance. Sources of such exclusion information include different branches of a conditional, and sections of code that are sequential due to either control/data-dependence or compiler-imposed sequentiality.

Once the routes are chosen, the program intermediate representation is augmented with explicit data communication operations. For data transfers controlled by programmatic control, explicit copy operations that move data through data interchanges are inserted. For tag-switched data transfers, tag-formation and data transfer initiation operations are

added. At this point, the virtual architecture has been transformed into a post-placement and pre-scheduling architecture. All operations have been placed within a tile, and all data transfers between tiles have been placed on a channel.

Scheduling happens next. As preparation for actual scheduling, the latency of each operation on the hardware is extracted. This includes the latency of each tag-switched data transfer operation (*i.e.* the number of pipeline stages between a source and each destination), and the latency of copy operations at each data interchange (including delay through data segments). A resource map for each operation is also generated for the scheduler to statically ensure that resources are not over committed. Traditional instruction scheduling technology, such as modular scheduling for software pipelined loops [52]and list scheduling [20], can then be applied to generate an overall schedule of operations, including data transfers.

Once scheduling is done, further data path synthesis is performed to determine temporary buffering needs. At this point, the design has essentially been converted into an RTL-like specification. Synthesis beyond this point follows a more classic flow, including the generation of control state machines, and a more classic place and route for each tile.

The above spatial compilation flow maps virtual entities to physical entities before spatial planning. In reality, what is desired is a mapping of virtual entities to tiles during the spatial planning step. As long as that can be done properly, the exact binding of virtual entity to hardware entity can be delayed until as late as scheduling. It is often beneficial for the scheduler to have the final say on how virtual operations are bound to functional units. Therefore, in an alternate approach, the spatial planner deals with virtual entity, deciding which tile each virtual entity should map to. During this process, the spatial planner also incrementally considers a preliminary (and non-binding) virtual to physical entity mapping to ensure that available tile resource can support the tile placement under consideration.

5.4.1 Example Illustrating Flexibility of Early Spatial Planning and Late Scheduling

The following example illustrates how a scheduler may respond to different placements under the early spatial planning, late scheduling approach. Consider the following computation:

```
1. uAddr = a+4;
2. wAddr = b+8;
3. u = load(ObjectU, uAddr);        /* each object, ObjectU and ObjectW,  */
4. w = load(ObjectW, wAddr);        /* has its own address space. */
5. r = a+b;
6. s = 5+c;
7. x = u+r;
8. y = w+s;
9. z = s+r;
```

Next, consider the following three spatial plans and corresponding execution schedules. In all three cases, all the additions are performed in the same tile. That tile, tile A, also holds the values of free variables coming into this computation, *i.e.* the values of a, b, and c. The data objects ObjectU and ObjectW are placed in two other tiles, tile B and tile C respectively. In the three cases considered in this example, tiles B and C are at different distances from tile A.

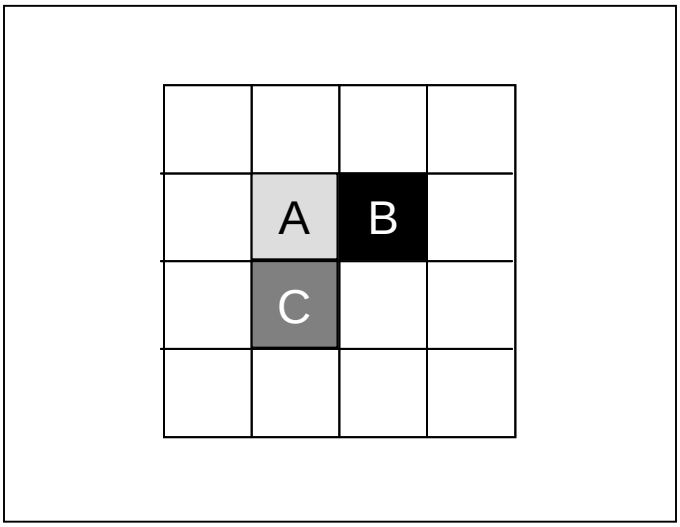


Figure 16: Spatial layout under Case 1.

In Case 1 illustrated in Figure 16, both tiles B and C are 1 tile away from tile A. Table 5 shows a possible schedule with minimum execution time.

Table 5: An execution schedule for case 1 spatial layout.

Time	In tile C	Between A and C	In tile A	Between A and B	In tile B
1			uAddr = a+4;		
2			wAddr = b+8;	uAddr transfers from A to B	
3		wAddr transfers from A to C	r = a+b;		u = load(uAddr);
4	w = load(wAddr);		s = 5+c;	u transfers from B to A	
5		w transfers from C to A	x = u+r;		
6			y = w+s;		
7			z = s+r;		

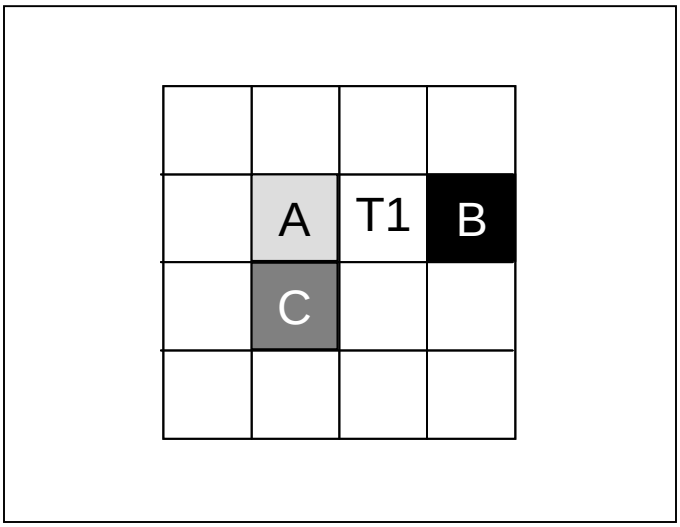


Figure 17: Spatial layout under case 2.

In case 2 illustrated in Figure 17, tile B is 2 tiles away from tile A, while tile C is still 1 tile away from tile A. Table 6 is a possible schedule. Note that the value u will not be available in tile A until time 7. Hence, the scheduler moves the computation of x=u+r from time 5 to 7, and moves the computation of z=s+r from time 7 to 5.

Table 6: An execution schedule for case 2 spatial layout.

Time	In tile C	Between A and C	In tile A	Between A and B	In tile B
1			uAddr = a+4;		
2			wAddr = b+8;	uAddr transfers from A to T1	
3		wAddr transfers from A to C	r = a+b;	uAddr transfers from T1 to B	
4	w = load(wAddr);		s = 5+c;		u = load(uAddr);
5		w transfers from C to A	z = s+r;	u transfers from B to T1	
6			y = w+s;	u transfers from T1 to A	
7			x = u+r;		

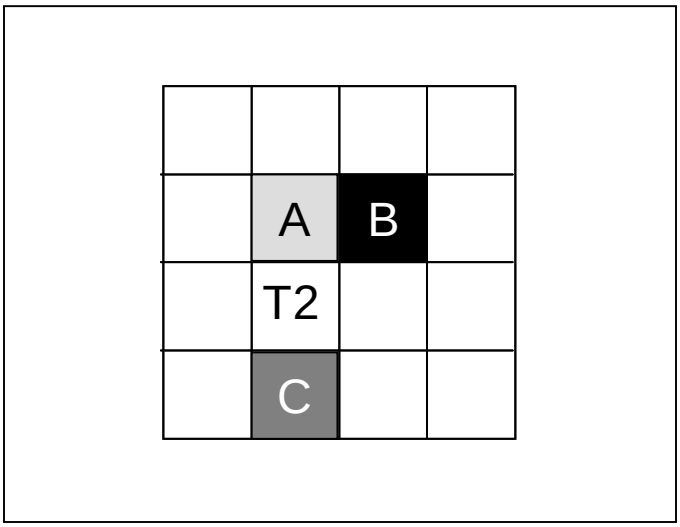


Figure 18: Spatial layout under Case 3.

In Case 3 illustrated in Figure 18, tile B is 1 tile away from tile A, but tile C is 2 tiles away from tile A. Table 7 shows a possible schedule. The computation of wAddr is now done before the computation of uAddr so that it can be sent out earlier to tile C. Computation of z=s+r is again placed in time 5, while the computations of x=u+r and y=w+s are done in time 6 and 7 respectively.

Table 7: An execution schedule for case 2 spatial layout.

Time	In tile C	Between A and C	In tile A	Between A and B	In tile B
1			wAddr = b+8;		
2		wAddr transfers from A to T2	uAddr = a+4;		
3		wAddr transfers from T2 to C	r = a+b;	uAddr transfers from A to B	
4	w = load(wAddr);		s = 5+c;		u = load(uAddr);
5		w transfers from C to T2	z = s+r;	u transfers from B to A	
6		w transfers from T2 to A	x = u+r;		
7			y = w+s;		

The above example illustrates the benefits of performing schedule after spatial planning, using good clock-cycle granularity timing information derived from the spatial plan. If the schedule had been fixed prior to placement to say the first schedule, but the placement is subsequently unable to place both ObjectU and ObjectW close to tile A, then the resulting system will most likely have to settle for a longer clock period.

5.5 Multi-tile Functional Units

Functional units that span multiple tiles introduce interesting challenges. The above compilation flow requires knowledge of inter-tile interconnection usage by the time the post-placement and pre-scheduling architecture is generated. Using profiling techniques similar to those used in VLIW compilers, the approach described above will derive the inter-tile interconnection bandwidth from the data flow represented in the compiler IR. This works fine as long as an FU does not span multiple tiles. When it does, a method of dealing with FUs having internal interconnect that spans tile-boundaries is needed. Furthermore, if inter-tile pipeline stages are inserted within a multi-tile FU by the spatial compiler, a technique has to be in place to ensure that correctness is preserved. The following are two possible approaches to solve the problem.

One approach sub-divides a multi-tile FU into a number of micro-FUs that are each smaller than a tile. When mapping virtual architecture to a pre-scheduling "physical" architecture, the compiler can look inside an FU macro, and place each micro-FU into a tile. Inter-tile interconnection that is within the FU macro is visible to the spatial compiler because they are between the micro-FUs. Furthermore, during scheduling, the spatial compiler takes each micro-FU as the scheduling unit, and deals with further insertion of temporary buffer to equalize parallel paths that fork and then re-merge.

Another approach uses pre-pipelined multi-tile FUs. Again, one can view the internal pipelines as dividing the FU into sub-pieces, with signals between the sub-pieces registered. Global floor planning has to keep each sub piece within a tile and may be further constrained regarding the relative placement of the sub-pieces. Furthermore, the internal flow of operands within a multi-tile FU, including those that span tiles, is not scheduled like conventional operands between FUs. Instead, once computation is launched from the FU's inputs, it arrives at each subsequent pipeline stage after a fixed delay. Under this approach, the FU is much like a hard-macro, whereas the first approach treats it as a soft-macro.

5.6 Low-cost Pipelined, Time-multiplexed Interconnect Hardware

Pipelined, time-multiplexed data transport hardware is commonly found in expensive supercomputing and high-end networking systems. Significant hardware costs come from dynamic arbitration needed to access output ports, and dynamic buffer management (allocation and deallocation) needed to temporarily buffer data that does not have immediate access to desired outputs. Dynamically switched data paths also generally support rapidly reconfiguring the switch between many possible input-to-output-port connection patterns. This flexibility is made possible using expensive data-path and control.

While high cost dynamic switches are appropriate for interconnecting expensive larger-scale processing systems, they are too expensive to use as native low-level interconnect between inexpensive compute and memory blocks within a reconfigurable computing system. To address this problem, ACRES introduces a new switch architecture that combines static and dynamic switching to reduce hardware cost while permitting flexible pipelining and time-multiplexing of data transport paths. In contrast to prior dynamically

switched networks, this hybrid approach uses both configuration control and dynamic control over data-paths to deliver the required data path connectivity.

Configuration control provides a cheaper means of customizing and controlling hardware when configurations do not changed rapidly. Under those circumstences, control can be implemented with less hardware and power. Static analysis and design techniques are very effective in predictable embedded applications. For many pipelined interconnects, data path sharing is statically planned to ensure that there is no danger of conflict at run-time, eliminating the need for dynamic arbitration of output ports. Furthermore, any buffering at intermediate points along a transport path is statically allocated, avoiding the need for complex dynamic buffer management hardware.

In comparison, dynamic control is appropriate when control changes rapidly, such as from one clock cycle to another. However, dynamic control need not have the full range of flexibility. This departure from full flexibility can be exploited through static compile time analysis and synthesis to greatly reduce dynamic switching costs. For example, when executing loops, highly optimized dynamic sequences can be constructed that efficiently execute a stream of loop iterations. Dynamic control only needs to deal with the variations encountered within the loops. Furthermore, cycle-by-cycle schedules are statically optimized to maximize transport utilization, eliminate transport arbitration and minimize transport buffering.

5.6.1 Example Usage Scenarios

This section presents a number of usage scenarios beyond those described in Section 5.3, with the goal of explaining our expectation that the need for complex dynamic switching is rare. The general operating assumption is that a relatively small amount of code is configured onto the hardware at any time. For some simple applications, this may be all the code there is, *e.g.* in some DSP or media processing applications. In other cases where more code is involved, reconfiguration can be done between significant runs, *e.g.* when the application code moves from one loop nest to another. When reconfiguration is done infrequently, the relatively high overhead of loading configuration is amortized over the relatively long run time between reconfigurations. Under these assumptions, a specific configuration for a piece of hardware need only support the relatively small dynamic variation that is seen within a small amount of code. Across larger time scales, variation within additional code is supported using reconfiguration.

Simple Pipelining

One very common use of inter-tile data transport is simply to provide a pipelined communication path. In the body of a highly parallel software pipelined loop with initiation interval (II) of 1 (*i.e.* execution of loop iterations overlap, with a new iteration started each clock cycle), a new piece of data is shipped down each logical communication path every cycle. If the communication path crosses tile boundaries, it is pipelined. But unless the inter-tile interconnect is clocked faster than the general system, there is no spare capacity in utilized data segments for sharing with other logical communication. Figure 19 below illustrates one such case. There is no dynamic switching, and propagation control is extremely simple.

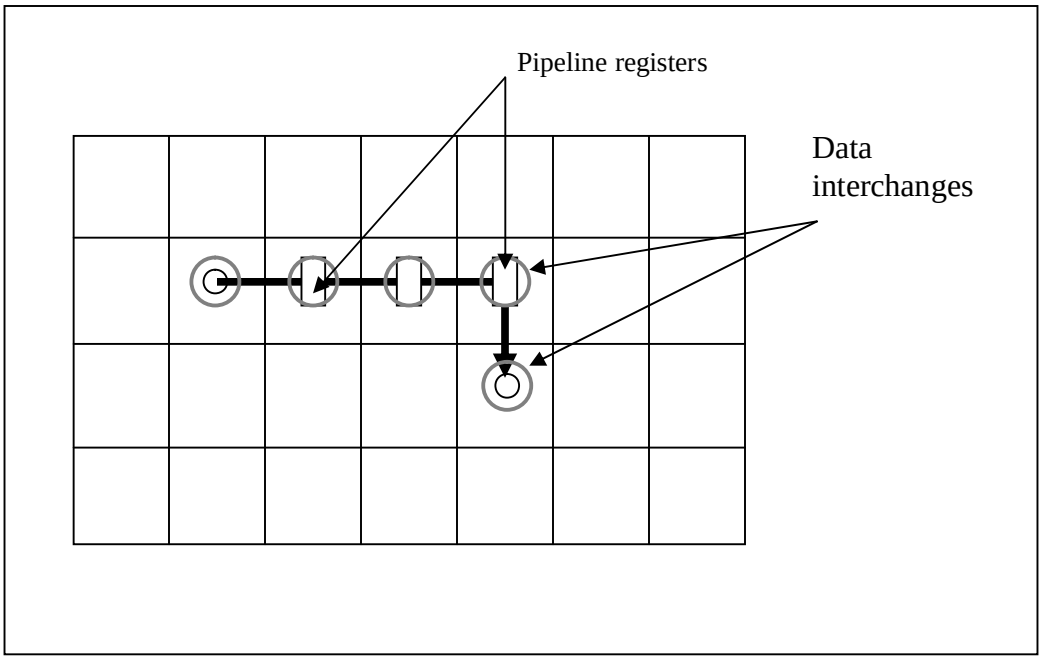


Figure 19: Communication path with simple pipelining

Pipelining and Switching without Arbitrary Delay Buffers

Even in cases where a data transport path is shared, it is likely that a long inter-tile path will only involve multiplexing at a few points. Consider the example in Figure 20 where there are two transport paths each spanning 5 tiles. The two logical communication paths share large segments of their physical communication paths. Only two multiplex points that require switches and dynamic control are needed even though there are six data interchanges on each communication path. This greatly reduces the number of multiplexing points and the fan-in/fan-out at each multiplexing point. This example can be well supported by a hybrid static/dynamic architecture where a data transport path is statically configured to support only a few paths and incorporates dynamic switching exactly where needed.

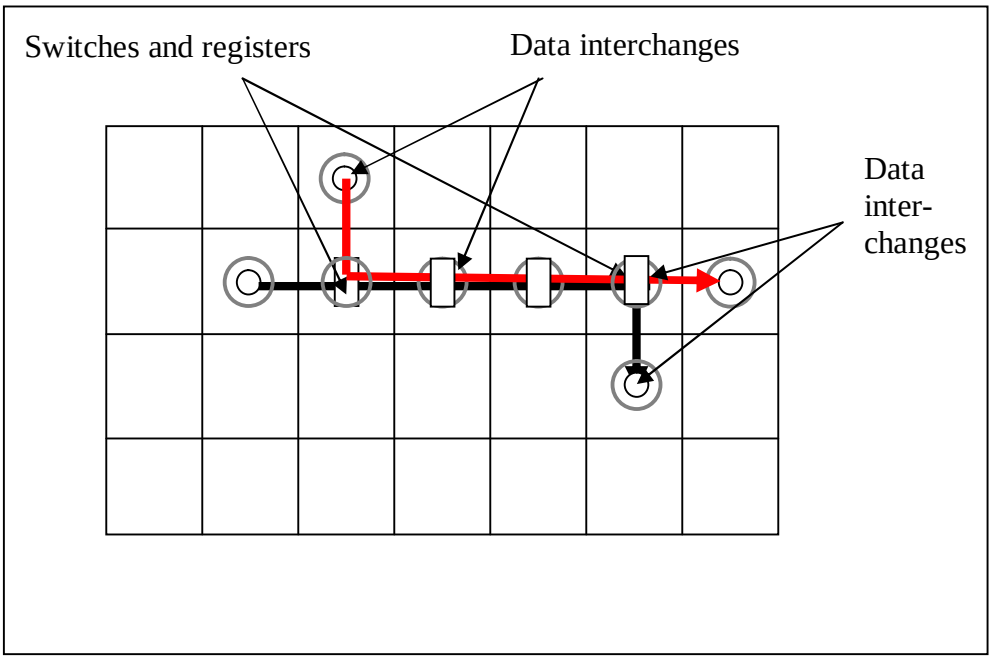


Figure 20: Data transport path shared between two logical communication paths.

The static analysis and design of multiplexed data transport can also simplify data buffering requirements. Referring back to the example of Figure 20, if we assume that the black and red path are both parts of a software pipelined loop with initiation interval of two, a simple, efficient schedule is to allocate alternate clock cycles of each data segment to the other logical communication. Thus a shared segment may devote the odd cycles to the black communication and the even cycles to the red communication. Its neighbor will do the opposite, with odd cycles for red communication and even cycles for black. By doing that, each data interchange only need one register to hold the data that arrives each cycle.

Pipelining and Switching with Extra Arbitrary Delay Buffer

Figure 21 illustrates a more complex example where additional intermediate buffering, beyond a simple pipeline stage register, is needed along a transport path. Three logical communication paths are marked in red, black and blue in the Figure 21. These are assumed to be part of a software pipeline loop with initiation interval of 2. The “e” and “o” markings next to each segment along the red, black, and blue paths indicate the clock cycle (“e” for even, “o” for odd) on which the respectively colored logical communication uses each segment. The data interchange in the shaded tile takes in a piece of the blue communication on an even cycle, but buffers away that piece of data at the data interchange the next cycle, and only moves it on to the next data segment at the next even cycle.

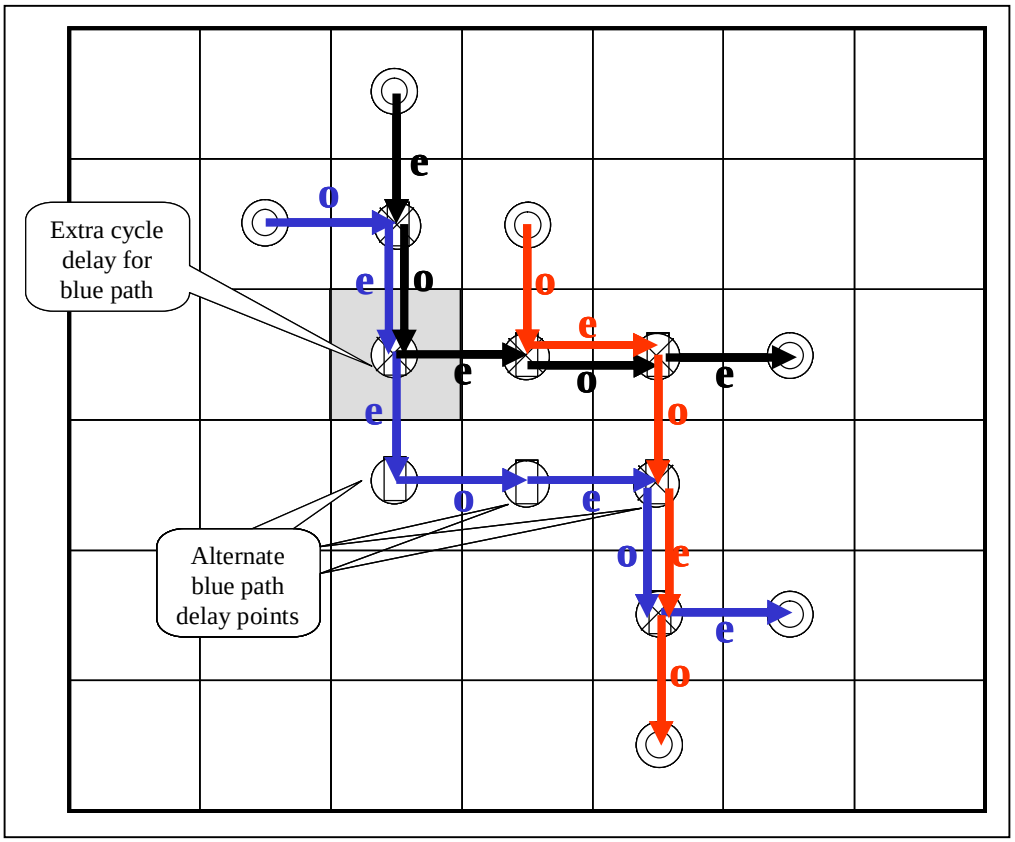


Figure 21: Example to illustrate need for extra delay and buffering at data interchange.

In this example, the exact location along the blue path where the extra delay occurs is flexible – it can be inserted anywhere along the blue path from the shaded tile to the data interchange just before the red and blue paths share a data segment. Alternatively, an extra delay can be inserted on the red path. Regardless of the exact choice, without changing routes, an extra cycle delay must be inserted on a communication path to make full use of data segments bandwidth. The example illustrates two points: (i) extra buffering beyond simple pipelining may be needed at a data interchange; (ii) it takes fairly complex interaction among multiple logical communication paths to require this extra buffering.

5.6.2 Data interchange Designs Exploiting Configuration Control

Section 5.1 describes a data interchange with full switching flexibility. In the rest of this section, alternate data interchange designs are described that reduce cost using more limited data switches. Also discussed are the effects of these switching limitations on usage flexibility.

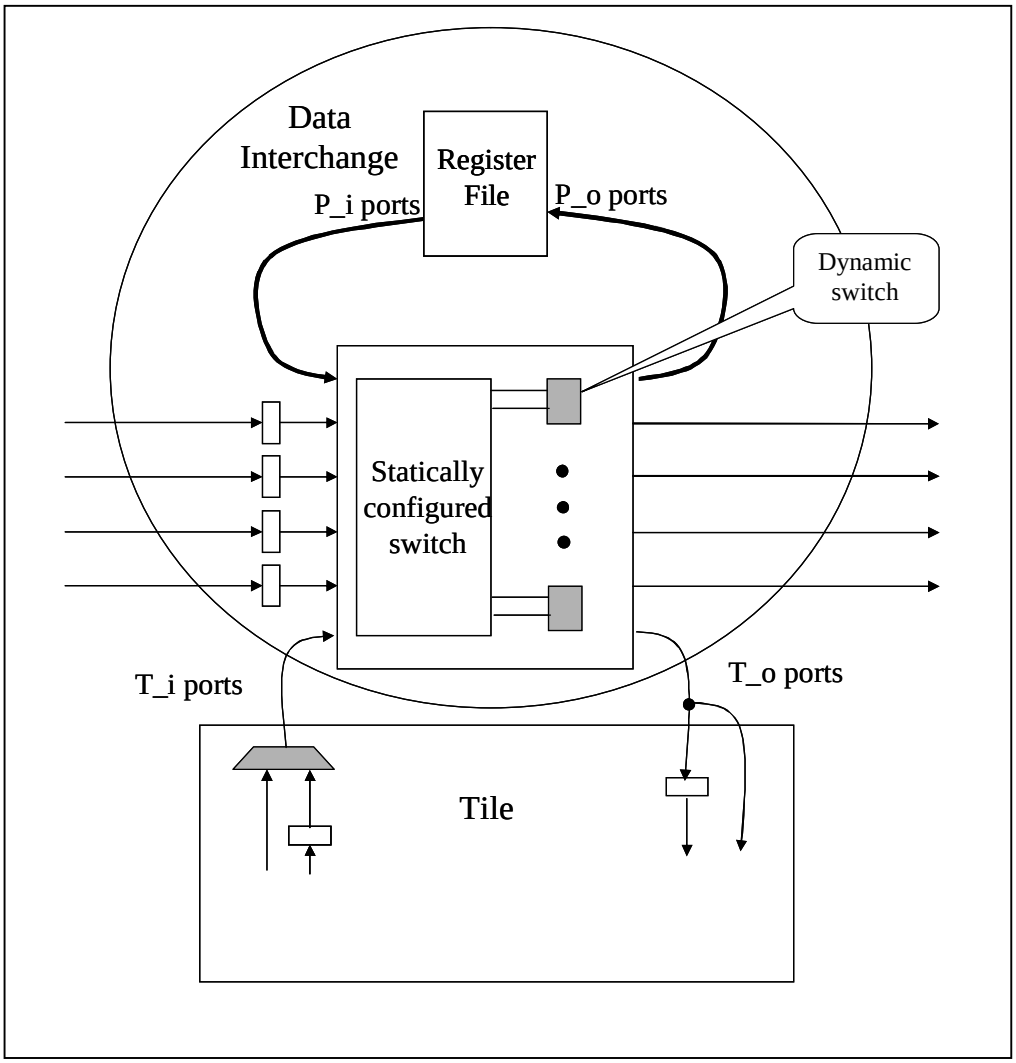


Figure 22: A combination of statically configured and dynamically controlled switches replaces the dynamically controlled full crossbar.

Figure 22 shows a data interchange that is similar to the one in Figure 12 of Section 5.1, but with the internal switch decomposed into a statically configured portion, and a series of small, dynamically controlled switches. As an example, each output is connected to its own dynamically controlled switch. The number of inputs to each dynamic switch only needs to be the small fan-in commonly encountered when dynamically multiplexing among only a few logical paths. The configuration controlled switch programs the actual source of inputs to the dynamic switch, but on a longer time scale using configuration control. In some designs, the configuration controlled switch may offer limited connectivity rather than full connectivity. For example in a multiple segments per data channel situation, *i.e.* multiple parallel segments going in the same direction, configuration control may allow only a subset of the segments in the same channel to be connected to a particular dynamic switch.

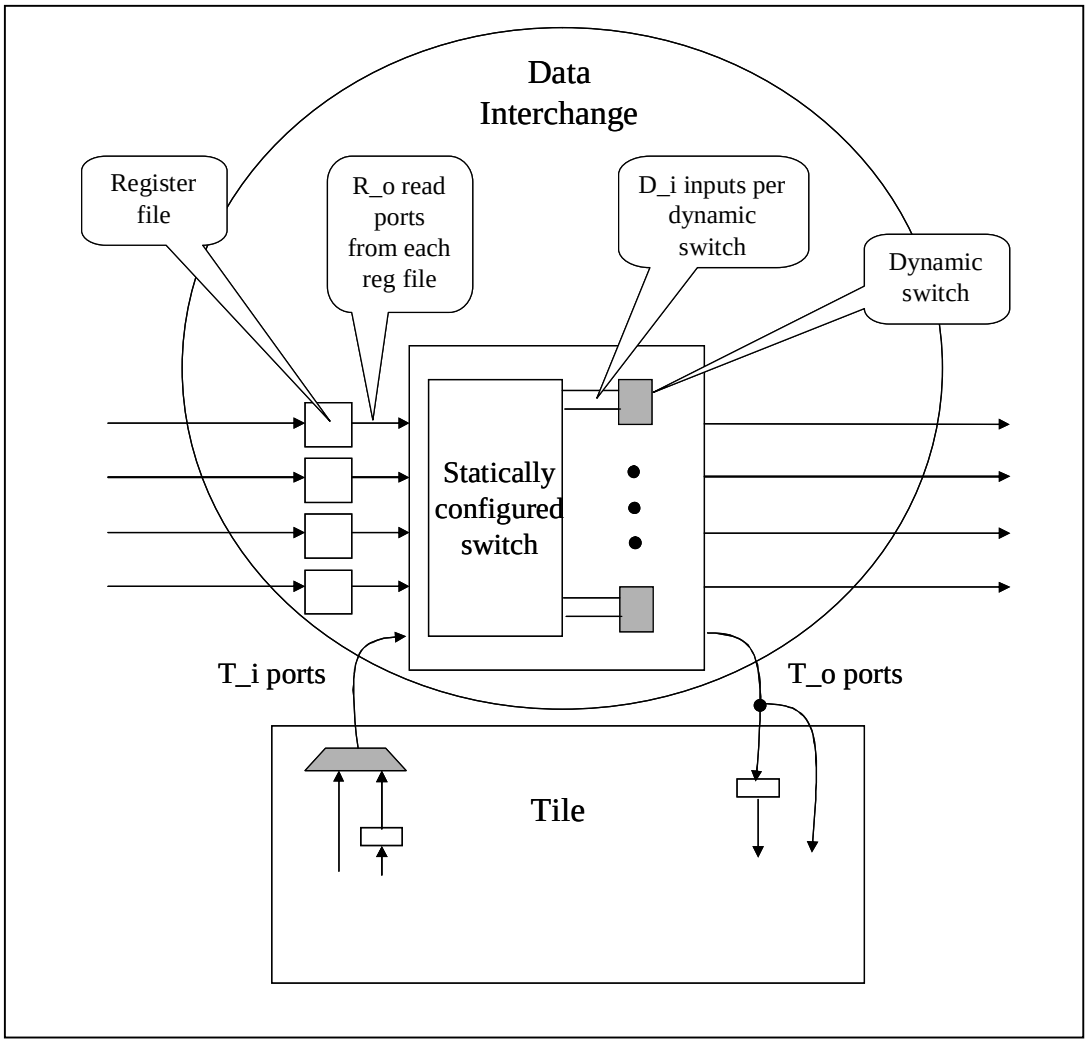


Figure 23: Distributed register files with smaller number of read and write ports replace a centralized, highly ported register file.

Figure 23 shows yet another data interchange design. Unlike that in Figure 22, the previously centralized register file is now distributed. The number of read ports from each distributed register file affects the number of values that can be read out of each register file in the same cycle, for immediate transmission out of the data interchange. That number is expected to be small in practice.

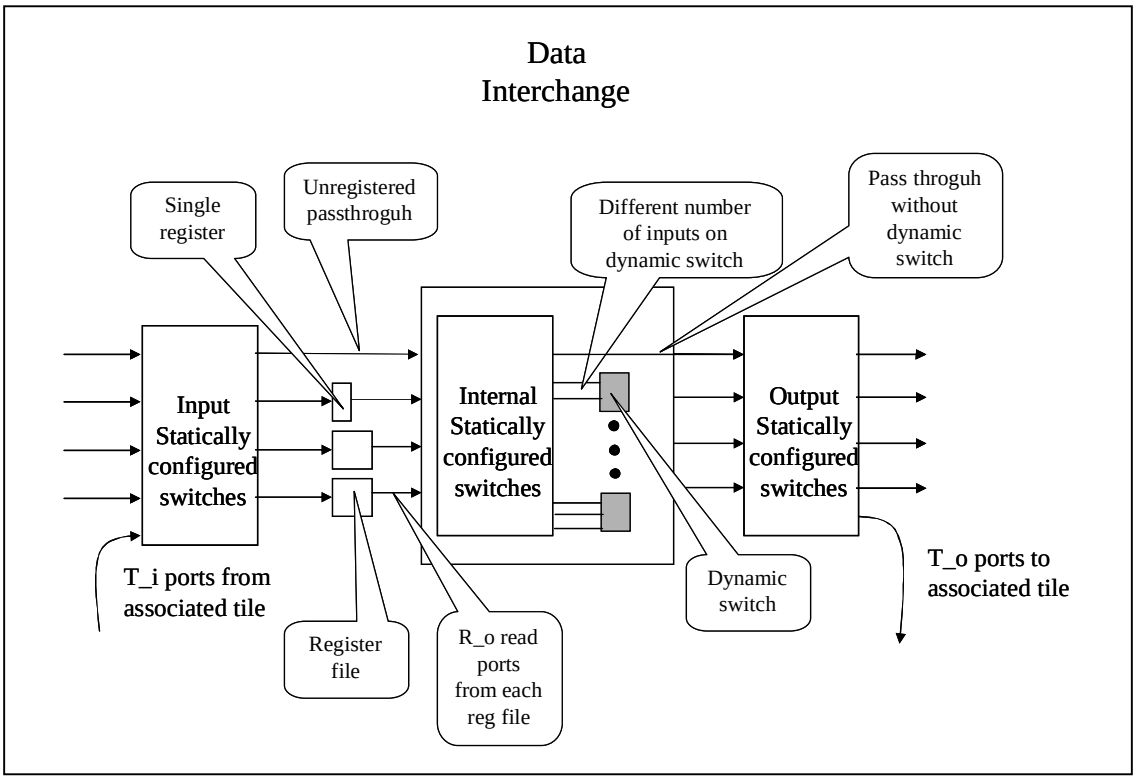


Figure 24: A variety of buffering and dynamic switching options are made available, together with statically configured switches. The design strives for cheaper hardware, while exploiting static analysis to reduce any impact on performance.

Figure 24 shows a data interchange design evolved from the design of Figure 23. One change from Figure 23 is instead of providing uniform buffering for every input, Figure 24's design includes a mix of single register buffers, register file buffers, and unregistered paths. Similarly, closer to the output side, the design includes dynamic switches with different number of inputs, and also paths that are passed through without dynamic switching.

In another departure from the design of Figure 23, the design in Figure 24 has three groups of statically configured switches. One addition is between the data interchange's inputs and the buffering options. This allows inputs to be statically configured to use the type of buffering (or no buffering) as required. Similarly, the statically configured switches just before the output enables the outputs to be statically connected to the dynamic switch of appropriate number of inputs (or no dynamic switch).

A design that provides a range of buffering choices may choose not to use the static input switch, but instead have a fixed partition of buffer resource types among inputs. For example, among the inputs from the same channel (*i.e.* same direction), some will be wired straight through with no buffering, others with singleton registers, and yet others with register files. Such a design is less flexible but also consumes less hardware. The output side can be similarly designed to use less hardware.

5.7 Mini-tiles: Refinement of the Tile Abstraction

The RC-1 computing fabric uses a tile abstraction whose linear dimension is related to the distance traversable within one target system clock cycle. This abstraction, while simple, has a limiting assumption that each pre-fabricated, reconfigurable device has one target clock speed. ACRES introduces a mini-tile abstraction to overcome this limitation.

Let us consider an RC-2 family of chips based on the mini-tile abstraction. An RC-2 chip is partitioned into mini-tiles, as illustrated in Figure 25. For the RC-1 family, each logical tile consists of exactly one physical tile. The RC-2 family extends this and allows a logical tile to be a compound tile consisting of an extended square region of adjacent physical mini-tiles. The RC-2 chip in Figure 25 looks almost identical to the RC-1 chip in Figure 11 with the exception that local interconnect extends across mini-tile boundaries. Similar to a tile, a mini-tile is the coarse-grain unit for placement during spatial planning. Unlike a tile, non-pipelined interconnect is allowed to cross from one mini-tile into its neighbors. Depending on the target system clock cycle time, local, non-pipelined interconnect is allowed to span up to several mini-tiles consistent with the target clock cycle time.

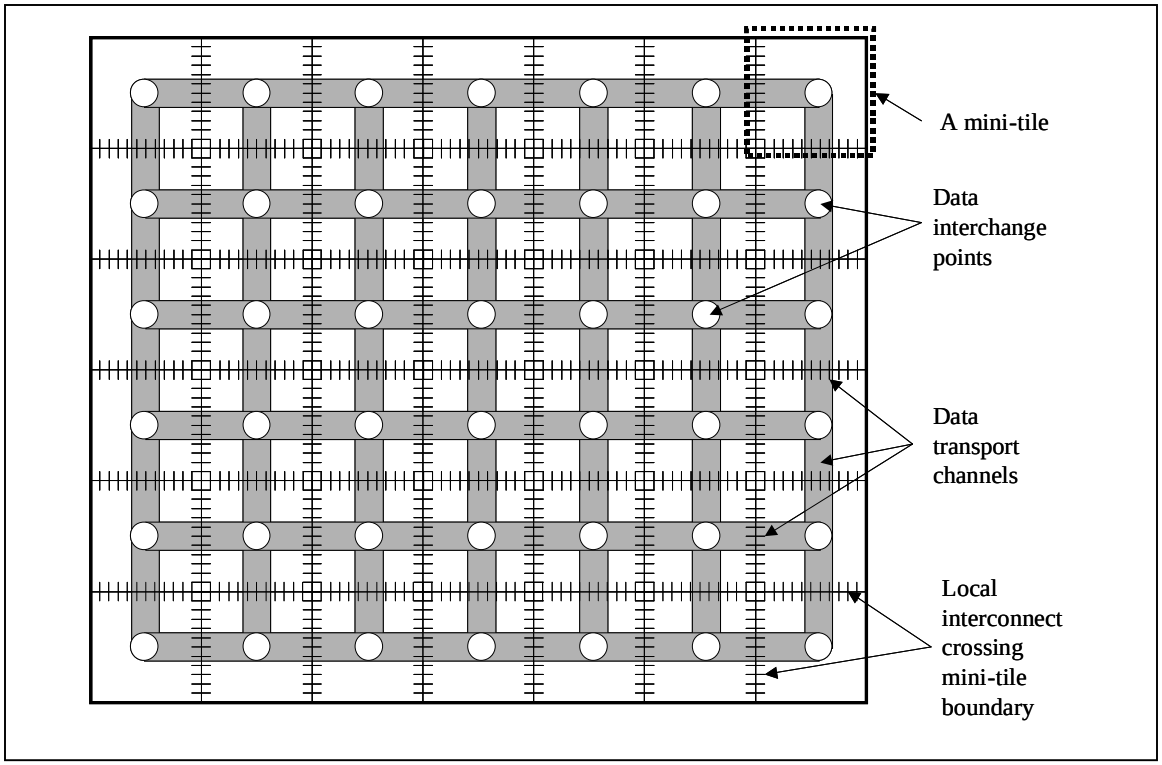


Figure 25: An RC-2 chip with mini-tiles.

As an example, a first design with a high-speed clock might allow local, non-pipelined interconnect to span only a single mini-tile while a second design with a lower-speed clock might allow non-pipelined interconnect to span a 2x2 block of mini-tiles. In this second design, each logical tile is implemented with a 2x2 block of mini-tiles.

Forming logical tiles out of multiple mini-tiles requires additional provisions in the inter-tile interconnect infrastructure. When a logical tile is formed out of *n*x*n* mini-tiles, an

inter-tile communication path crosses *n* data interchanges when traversing straight across a tile. Without additional provisions, such a path will incur *n* system clocks of delay, an undesirable outcome. ACRES provides two options to deal with this longer than desired delay.

One solution is to employ multiple clocks in the system, allowing the inter-tile interconnect to be clocked at a faster clock than the system clock. A system with logical tile formed out of *n*x*n* mini-tiles, for example, may have an inter-tile interconnect clock that is *n* times faster than the system clock. The net result is that even though straight traversal across a logical tile incurs *n* interconnection clocks, the actual time involved is the same as a single system clock. Careful orchestration of clock domain crossings as data enters and leaves the inter-tile interconnection is needed, however. Furthermore, because of rounding effects, the overall inter-tile traversal time, expressed in terms of system clock cycles, may incur an additional rounding up system clock cycle. Because the inter-tile interconnect is clocked at a higher clock rate, this design has the potential to support higher throughput than the next solution. Appropriate buffering/queuing structures such as those used in Virtual Wires [4] is needed at the clock domain crossings to take advantage of the higher inter-tile communication throughput.

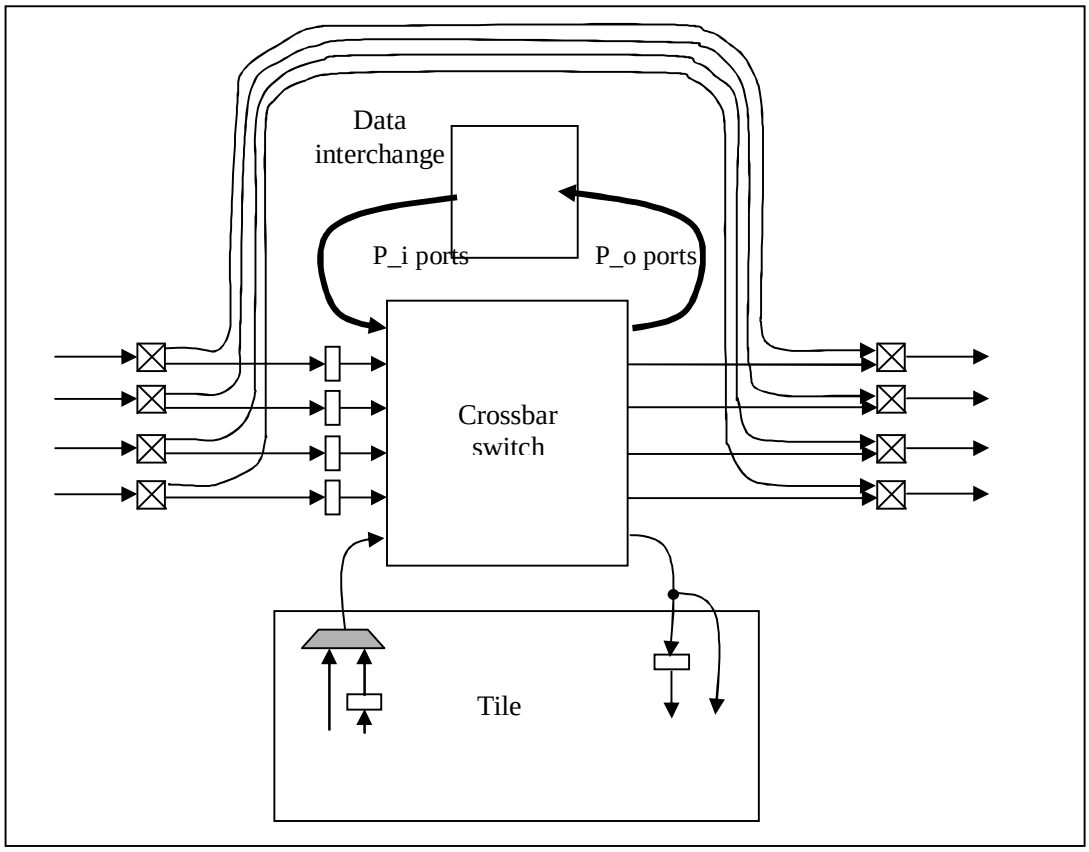


Figure 26: A data interchange design that supports mini-tiles.

A second, conceptually simpler, solution is to augment the data interchange design of Figure 12, with configuration controlled bypasses as shown in Figure 26. The statically configured bypasses allow an in-coming data segment to connect statically to an out-

going segment without going through a pipeline register and the crossbar switch. An implementation may combine the bypass function with the switch function to arrive at a design that requires less hardware (See for example the design in Figure 24).

When *n*x*n* mini-tiles are composed into a single logical tile in this second solution, (*n*-1) data transport points on any run of data transport segments can be by-passed so that data transport encounters only 1 pipeline stage while crossing *n* mini-tiles. As a further benefit, this design allows inter-tile transport to enter or leave each tile at up to *n*x*n* different points within a tile composed of *n*x*n* mini-tiles. This provides a higher degree of placement precision and flexibility than the original tile design that may be exploited by the ACRES spatial compiler.

This second solution has a downside that it over-provisions the latches and switches within a data interchange, particularly when the tile size chosen for a design comprises a reasonably large number of mini-tiles. In mitigation, a design may choose to make data interchange resources part of the mini-tile internal resource when they are not used for inter-tile transport.

The ACRES spatial compiler treats the compound tiles formed from the mini-tiles as non-overlapping *n*x*n* blocks of mini-tiles. Such a view keeps synthesis steps simple. For example, as in the case of the original tile design, issues pertaining to local interconnect within a tile is independent of similar issues for another tile. This effectively divides the problem into multiple smaller, independent ones. Detailed place and route, for example, can be done independently for each tile, which is clearly a benefit.

5.8 Span: Relexation of the Tile Abstraction

The tile abstraction, even when augmented with mini-tiles, has the limitation that a source communicating with a destination that happens to be just across the tile boundary is constrained to pipeline the communication, even though the distance traversed is clearly less than the intra-tile distance across one tile. Removing this limitation will require a fair degree of departure from the tile abstraction.

One way to remove this limitation uses the same hardware as mini-tiles, but a different abstraction in the spatial compiler. The underlying mini-tiles continue to provide a grid for coarse-grain placement. Furthermore, the spatial compiler continues to use a discretized unit, pegged to mini-tile dimensions, for reasoning about the latency of inter-tile communication. The spatial compiler therefore avoids time intensive analog modeling of inter-tile interconnection delay.

The big departure from the tile abstraction in the span abstraction is the relaxation of tiles as non-overlapping units demarcating the boundary of intra-tile communication. Instead, the spatial compiler determines the need for inter-tile communication on a communication path by communication path basis, based on the geographical extent of each communication path. If a communication path extends over too large an area, inter-tile communication infrastructure, with associated pipelining requirement, is used.

Concretely, the spatial compiler uses a span abstraction that defines the acceptable extent for intra-tile communication. For example, based on the target system clock speed and

the speed of the underlying hardware, the spatial compiler may choose to limit intra-tile communication to a *u*x*u* span. As long as a communication path stays within a *u*x*u* span, it can avoid using inter-tile communication infrastructure. On a long communication path, the span, just like the tile, determines the interval at which pipeline registers are needed.

One way to view span in relation to tiles and mini-tiles is that whereas the tiles formed from mini-tiles form non-overlapping tiles, spans are blocks of mini-tiles that are allowed to overlap. As an illustration, consider a chip with a 4x4 array of mini-tiles shown in Figure 27. If a design uses tiles composed of 2x2 mini-tiles, there are 4 non-overlapping tiles. Figure 27 also shows two signal nets A and B. Under the tile abstraction, Net B incurs inter-tile interconnection. Under the span abstraction with spans comprising of blocks of 2x2 mini-tiles, Net B fits within a span and is thus permitted as an intra-tile communication path. Similarly, Net A fits within another span, one that overlaps the span encompassing Net B.

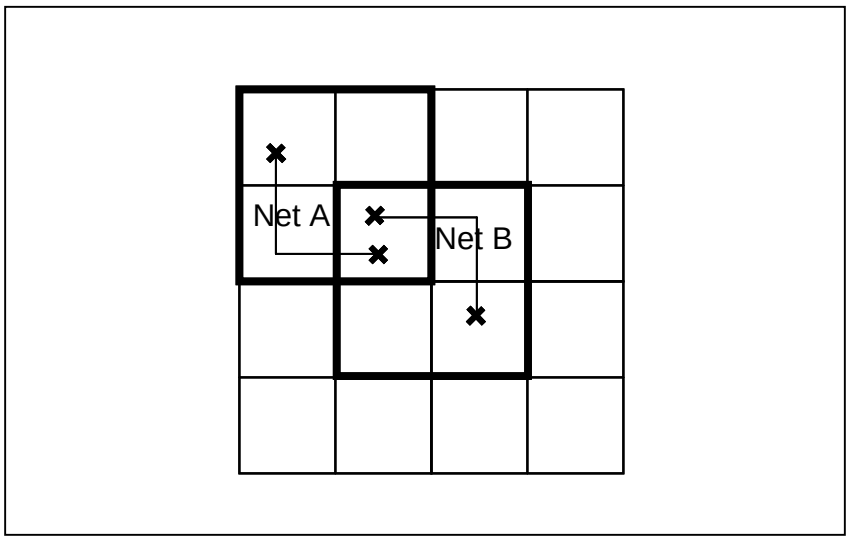


Figure 27: The span abstraction permits both Net A and Net B to be intra-tile nets, something not possible with tiles.

6 Distributed Reconfigurable Memory

Specialized accelerators commonly found in embedded systems need high memory bandwidth to provide operands at a data rate sufficient to feed their high performance compute engines. Global main memory with caches, as used for general-purpose computation, does not adequately support high-performance accelerators. In particular, when a global main memory is scaled to very high bandwidth, an expensive memory system, similar to a supercomputer memory, is a likely result.

Specialized accelerators often dedicate one or more operand storage devices to hold specific types of data. These take the form of lookup table memories, coefficient array memories, streaming buffer memories and other dedicated storage that can be used to accelerate specific computation. The use of multiple dedicated memories within a single computation allows parallel access and increases memory bandwidth. Unlike a global shared memory, these memories are local memories and need not be integrated into a single address space. They do not support general global memory access and they are much less expensive. These memories are deployed in configurations that are customized to a given application and, it is not immediately apparent how they should be deployed within an application independent programmable chip.

Traditional programs, such as those written in FORTAN and C, reference a central main memory. Techniques are therefore needed to put programs in a form that is amenable to execution on distributed memory architectures. As part of this process, the ACRES compiler and its users identify virtual RAMs that can be treated as isolated memory objects within a distributed memory system. Each virtual RAM is treated as a contiguous vector of memory words that can be referenced by load or store operations with an indexed memory address.

The virtual RAMs may differ in number, size, width, and depth from the physical RAMs that are actually provided. While a small virtual memory may be supported using only a single physical memory, as the number of words and the bit-width of a virtual memory increases, the number of physical RAMs required to support it also increases. The ACRES architecture provides techniques to utilize multiple physical RAMs to support the needs of a single virtual RAM. Reconfigurable logic and dynamically controlled switches within the ACRES fabric allows larger hardware structures to be flexibly assembled from multiple pre-fabricated RAM blocks.

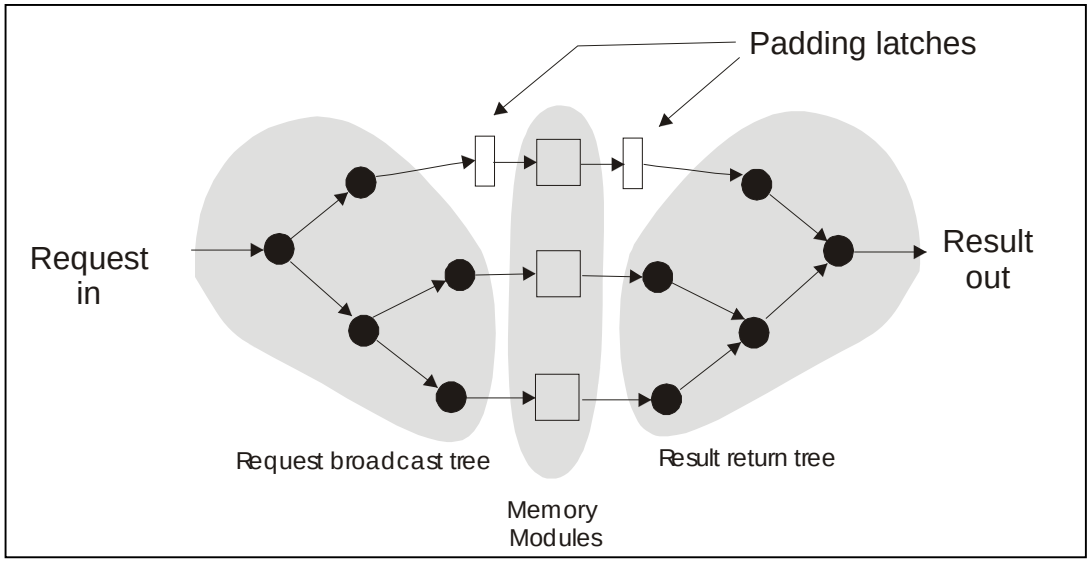


Figure 28: Virtual RAM implementation

The ACRES architecture defines a procedure to assemble a parametric family of virtual memories from physical memories along with necessary glue logic. An example virtual RAM is depicted in Figure 28. A request input port accepts a read or write command and the address of the word to be accessed. In the case of a write, the request input port carries the data operand to be written. In the case of a read, a result output port returns the correct result from a successful read operation. A pipelined spanning tree supports the broadcast of address and write-data. Similarly, a pipelined merge tree returns results from reads. Both the broadcast and return trees are assembled out of configurable network elements, represented by black circles in Figure 28, on the computing fabric. Note that in Figure 28, both memory access networks are height balanced, possibly incorporating padding latches, so as to provide a memory interface where the latency of memory operations is independent of RAM address.

6.1 Sharing Memory Resources

The ACRES architecture supports the use of a shared set of physical RAM blocks and network elements to support multiple virtual RAMs. Both bandwidth sharing and data array sharing can be supported. Bandwidth sharing allows a set of network hardware elements and RAM blocks to support broadcast and return access trees and RAM array access for multiple virtual RAM configurations. When a virtual RAM *X* is not in use, access tree and RAM block bandwidth used by *X* can be used for another virtual RAM. This is legal as long as no resource is used for two virtual RAMS on the same cycle.

Figure 29 illustrates an example of sharing access network and RAM port bandwidth. Three virtual RAMs, *X*, *Y* and *Z*, are mapped to a set of six physical RAM blocks pRAM_1 through pRAM_6. Virtual RAM *X* is spread over all six physical RAM blocks, virtual RAM *Y* is mapped only to pRAM_2 and pRAM_3, while virtual RAM *Z* is mapped to pRAM_4, pRAM_5, and pRAM_6. Figure 29 also shows the memory access nets, with solid arrowed lines for virtual RAM *X*'s request network, dashed ones for virtual RAM *Y*'s, and dotted ones for virtual RAM *Z*'s. Arrowed lines that coincide, such

as the solid line and dashed line between network element TC_10 and pRAM_2 share a physical link through statically scheduled time-multiplexing.

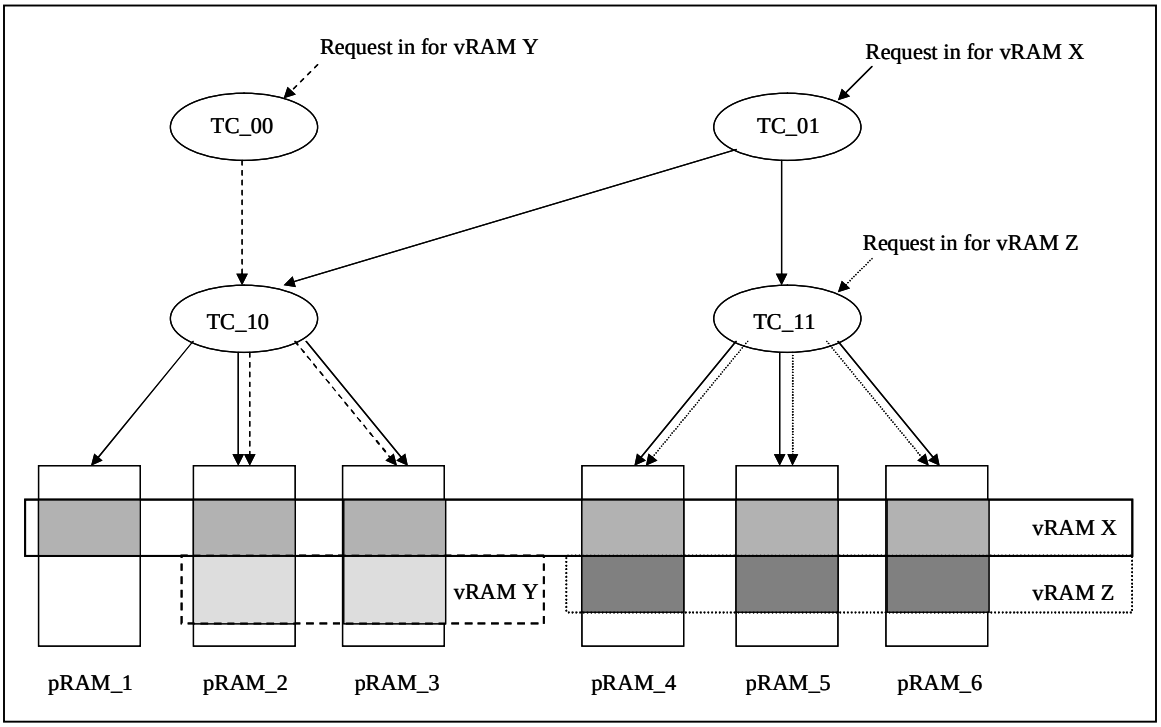


Figure 29: Memory resource sharing example.

RAM array sharing allows memory locations within a RAM block to support multiple virtual RAMs. Virtual RAMs have an associated data lifetime. A virtual RAM is initially dead. When valuable data is loaded into the RAM, it becomes alive. Data may be repeatedly manipulated within the live virtual RAM. When all useful results have been retrieved from the RAM, it returns to the dead state. A single word within a physical RAM block may represent more than one word as long as they are within multiple virtual RAMs, no two of which are simultaneously live. The reuse of memory to host different data objects is common on most computing platforms. What distinguishes ACRES is that the reuse of memory storage may involve a change in the memory access network topology.

Referring back to Figure 29, if vRAM *X* has a life time that does not overlap that of vRAM *Z*, the storage locations of vRAM *X* in pRAM_4, pRAM_5 and pRAM_6 can be reused by vRAM *Z*. As the example shows, the different between RAM array sharing and bandwidth sharing is not that large, mainly one of memory allocation.

The time evolution of virtual RAM to physical RAM mapping opens up an opportunity for very low overhead re-association of data with virtual RAM, *i.e.* values stored in memory becomes associated with different virtual RAM over time. A virtual RAM represents an abstract unit with associated access bandwidth and number of access ports. Over the course of execution, an object in the application, such as an array, may be partitioned into virtual RAM in a number of ways in order to support different levels of parallel access. For example, during one phase of execution, the array may be partitioned

into *n* parts as *n* virtual RAMs so that a loop parallelized into *n* threads can each work on one portion. During another phase of execution, another loop may be parallelized into *(2n)* threads to work on the same array, which is then more conveniently viewed as partitioned into *(2n)* virtual RAMs. Figure 30 illustrates this example.

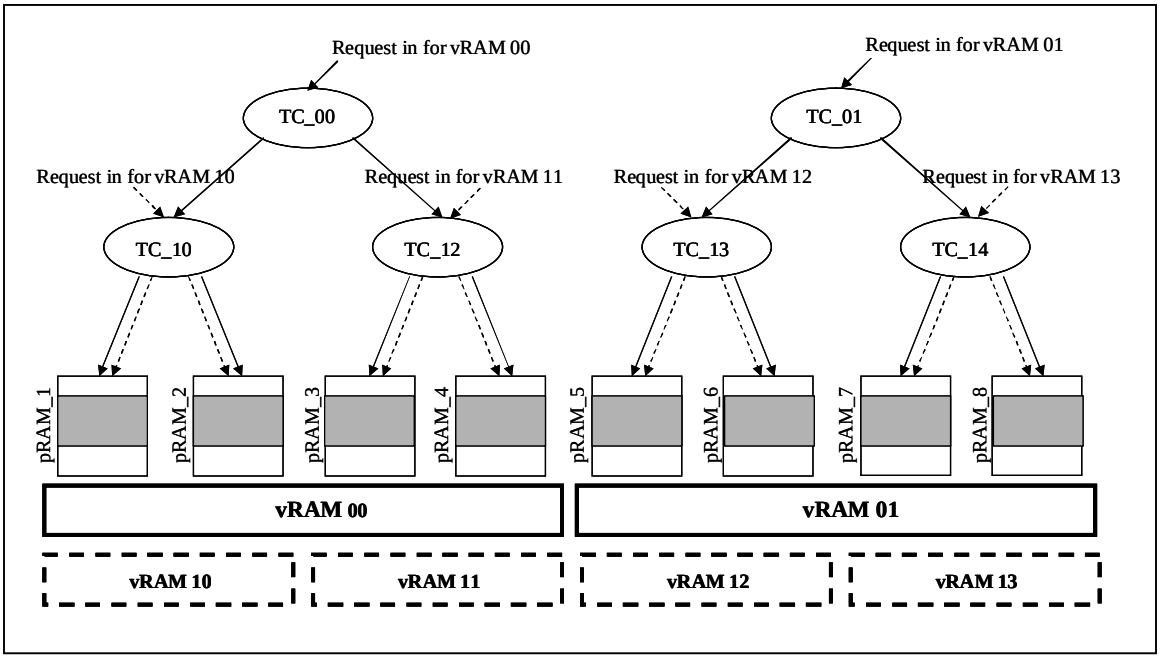


Figure 30: Example illustrating re-association of data into two different sets of virtual RAMs over time. At one time epoch, an array that spans eight physical RAM blocks is divided into two virtual RAMs: vRAM 00 and vRAM 01. At another time epoch, the same set of memory locations are organized into four virtual RAMs: vRAM 10, vRAM 11, vRAM 12, and vRAM 13. Data values may be carried from one epoch to the next epoch.

ACRES supports re-association of data into a different set of virtual RAMs by re-configuring the memory access networks. The re-configuration can either be implemented by changing configuration control bits, or it can be implemented using dynamic tag-based switching as elaborated in the next section. The significant thing to note is that with careful static planning, this re-association need not involve copying of data, but rather a re-organization of the memory access networks.

6.2 Memory Transport Cells

Dynamic RAM sharing relies on the use of tagged messages that traverse request broadcast and result merge trees. Each message takes a route through a web of shared network elements that depends upon its tag. The tag identifies the exact multicast tree that the message takes. Memory access network may be assembled out of the data interchanges described in Section 5. This section further elaborates on support for tag-based switching, and multi-cast using specialized memory transport cells shown in Figure 31.

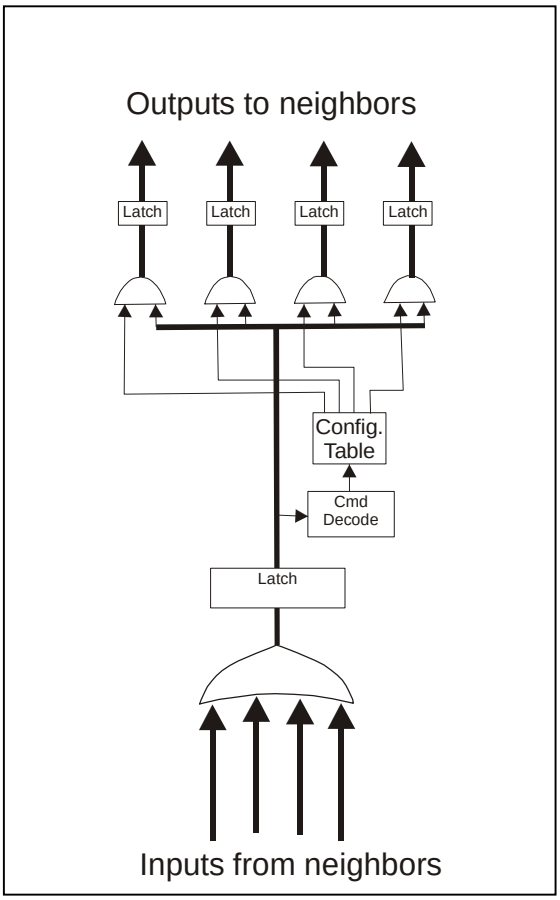


Figure 31: Memory Transport Cell

A message enters the transport cell through the input OR gate at the bottom of Figure 31. It is assumed that static scheduling ensures that no two messages enter the same cell on the same cycle. Inputs messages come from either neighboring cells, or a load/store unit issuing a memory operation. The tag in the message is decoded by the command decoder (Cmd Decode in Figure 31), and used to look up a configuration table. A successful lookup yields a bit-vector that identifies the set of neighbor cells to which the message should be forwarded. In Figure 31, the bit vector is used as an enable/disable signal to the AND gates that control propagation to the cell's neighbors.

When a transport network is used by multiple virtual RAMs, a transport cell effectively switches configuration from one virtual RAM to another when it selects a different configuration entry. This enables very fine grain sharing of a transport network. Conflicting access to shared network resource is avoided through static scheduling. A reservation table is derived for each virtual RAM's transport network usage. The scheduler uses these reservation tables to ensure that execution schedules it generates are conflict free.

6.3 Compilation Abstractions

Within the ACRES spatial compiler, memories are viewed at virtual and physical levels. Physical memories are pre-defined RAM blocks that are distributed throughout a computing fabric. A virtual RAM is a statically identified (*i.e.* at compile time) entity that behaves as a single linearly-addressed memory with a fixed number of access ports, and associated access bandwidth for each port. These two views, together with the abstraction of variables and data objects at the application source specification, form the three main compiler abstractions related to memory.

During the generation of application specific virtual architectures, ACRES programs are transformed so that they reference multiple distributed memory structures in parallel. The ACRES project has developed techniques that transform loop nests into a form that localizes memory access. The procedure defines the structure of a SIMD-like loop solver and its memory system. The resulting application-specific architecture minimizes the number of needed memory references and promotes spatial distribution and local storage and use of data. It is very important for performance and scalability that data is stored in memory that lies near its use.

The ACRES spatial compiler next maps source level variables and data objects to virtual RAMs. At the virtual architecture level, each load/store operation is annotated with the object that it accesses. The spatial compiler then goes through a mapping and transformation process that results in each load/store operation accessing only one virtual RAM. Multiple virtual RAMs are treated as separate, unrelated objects that cannot be referenced from a single memory access operation.

The process of generating virtual RAMs aggregates multiple variables into a single virtual RAM when they are all accessed using the same memory access operations. Conversely, a large source-level object may sometimes be partitioned into a number of disjoint virtual RAMs. For example a matrix might be partitioned into multiple sub-matrices that are accessed independently and in parallel.

Memory disambiguation is vital to this process. Without it, all load/store operations access the same virtual RAM. Static disambiguation is used where possible to achieve very efficient RAM access. Static disambiguation may be done strictly by compiler analysis, or may require user directives or use of specific coding styles (*e.g.* use of arrays in C instead of pointers). More dynamic techniques are also possible but will not be discussed at this time.

Each virtual RAM has associated with it one or more access ports. During the initial phase of spatial planning, just as arithmetic operations are mapped to functional units, load/store operations are mapped to memory access ports. As in the case of mapping virtual arithmetic operations to functional units, the exact phase ordering of this step is an area for interesting research. Performing the mapping early simplifies the compilation flow but may not yield the best performance.

Spatial planning is responsible for mapping virtual RAMs to physical RAMs and for assembling the memory access networks. A classic high-level synthesis approach would define the request and return trees that implement a virtual RAM prior to the start of

placement and routing. Within ACRES, the broadcast and return tree design is influenced by a tile placement floor plan. Physical RAM blocks are spatially placed into tiles prior to final interconnect design. Interconnect is then routed in the form of a spanning tree that conforms to the chosen placement. The tree is opportunistically reshaped to fit the chosen floor plan. Each virtual memory's latency is determined after assembly and depends upon RAM block placement.

The virtual to physical RAM mapping step may result in one-to-one, many-to-one, one-to-many, or many-to-many mapping. Many-to-one mapping happens when several virtual RAMs, all accessed from the same vicinity, fit into a single physical RAM in terms of total size and access bandwidth. The one-to-many mapping comes about out of necessity when a virtual RAM exceeds the size of a physical RAM block. Many-to-many mapping can arise when several one-to-many virtual RAMs share the same physical RAM blocks.

After spatial planning is completed and a pre-scheduling architecture is generated, the latencies of memory access networks, and the resource sharing between them are fixed. The scheduler can then make use of reservation tables and latencies derived for these networks to statically ensure that memory access operations are scheduled in a way that is efficient and avoids resource usage conflicts.

6.4 Memory Addressing, Access Width and Size

ACRES' distributed and reconfigurable memory introduces a number of complex addressing issues which are addressed in this section. Because of a wide disparity in performance between configured and hardwired circuitry, the following solutions each look for a common flexible mechanism that can be hardwired.

Figure 32 provides a complex usage example that illustrates several virtual RAMs that have been mapped on to several physical RAM blocks. The physical RAM block size need not be uniform. The address range of each virtual RAM is divided into contiguous chunks where each chunk is mapped to a contiguous region in a single physical RAM block.

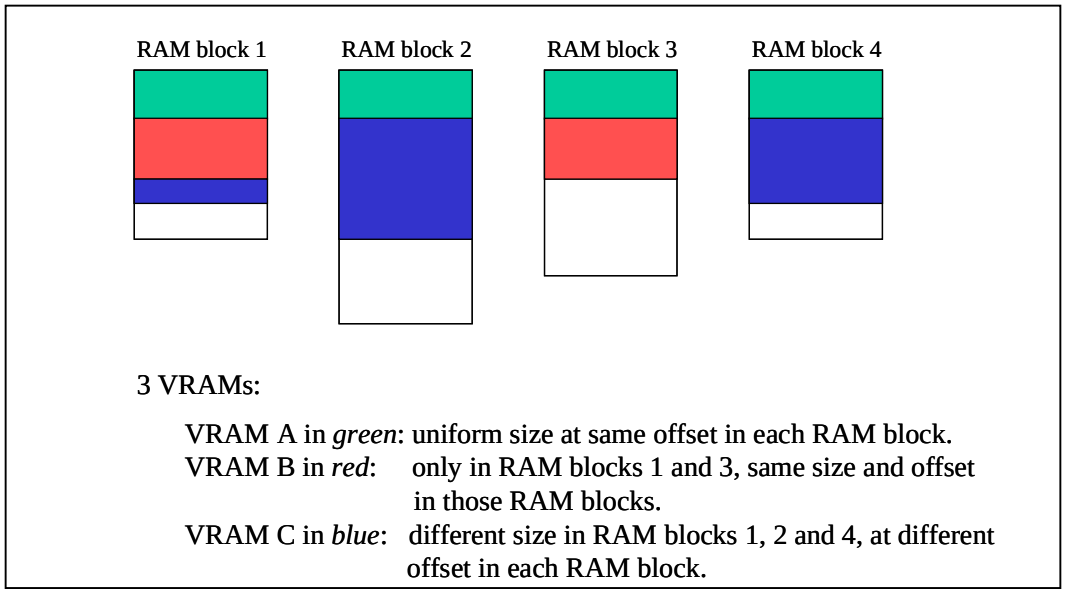


Figure 32: An example of a very general many-to-many virtual RAMs to physical RAM blocks mapping.

6.4.1 Basic Address Manipulation Mechanism

Every memory access specifies a virtual RAM and an address within the virtual RAM that is being referenced. For each access, the following mechanism identifies the chunk of memory within a physical RAM and the word within that chunk that is referenced.

For the following discussion, assume that a processor makes a reference with an address of the form:

<*virtual RAM identifier, offset within virtual RAM*>.

This address is broadcast via an access network to those RAM blocks used to implement that virtual RAM. At *each* physical RAM, the following sequence of action happens within its *address manipulation unit*.

1. The (*virtual RAM identifier*) is used to identify an entry within an *address mapping table*. If a match is found, the following information is read out of the address mapping table:

   (*local_start_offset, local_end_offset, local_base_addr*)

2. Check whether this RAM block is involved.

   (*local_start_offset* <= *offset within virtual RAM* <= *local_end_offset*)

3. If yes, the RAM block has (part of) the referenced memory, specifically at:

Local_address = local_base_addr
+ (offset within virtual RAM) – local_start_offset

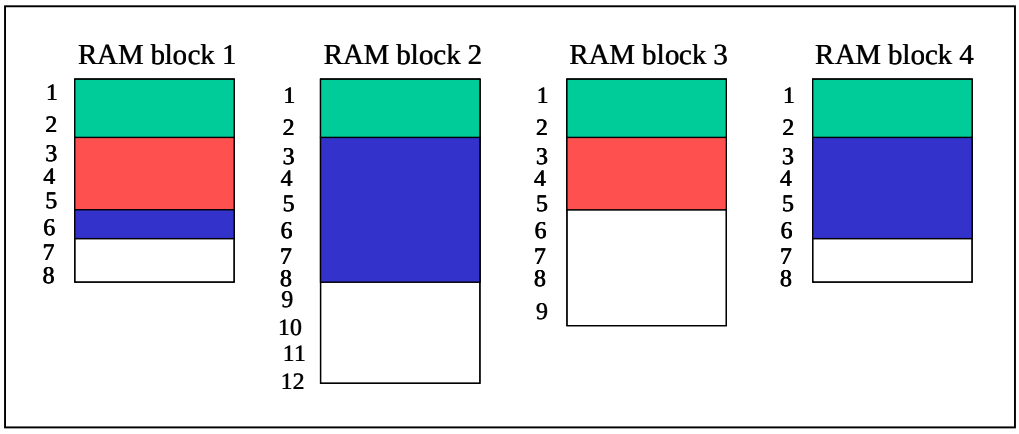


Figure 33: The example in Figure 32 with memory locations numbered.

local_base_addr is the address of the local RAM block where the virtual RAM's chunk within that local RAM block begins.

Consider the examples in Figure 33. The *local_base_addr* value for the green virtual RAM is 0 in every physical RAM block. For the red virtual RAM, this value is 3 in RAM block 1 and also 3 in RAM block 3. (Note: there is no match for the red virtual RAM's ID in RAM blocks 2 and 4's address mapping tables.) For the blue virtual RAM, this value is 6 in RAM block 1, 3 in RAM block 2, and again 3 in RAM block 4.

local_start_offset: consider each virtual RAM and (linearly) number its words, starting from zero. We use this, the offset, to identify a specific word within a virtual RAM. The *local_start_offset* of a virtual RAM for a particular RAM block is the offset of that virtual RAM's first word within that RAM block. Again consider the examples in Figure 33. The *local_start_offset* for the green virtual RAM is 0 in RAM block 1, 2 in RAM block 2, 4 in RAM block 3, 6 in RAM block 4. Similarly, this value for the red virtual RAM is 0 in RAM block 1 and 3 in RAM block 3. For the blue virtual RAM, this value is 0 in RAM block 1, 1 in RAM block 2, and 7 in RAM block 4.

local_end_offset Again consider each virtual RAM and (linearly) number its words, starting from zero. We again use this, the offset, to identify a specific word within a virtual RAM. The *local_end_offset* of that virtual RAM for a particular RAM block is the offset of that virtual RAM's last word within that RAM block. Again consider the examples in Figure 2. This value for the green virtual RAM is 1 in RAM block 1, 3 in RAM block 2, 5 in RAM block 3, 7 in RAM block 4. Similarly, this value for the red virtual RAM is 2 in RAM block 1 and 5 in RAM block 3. For the blue virtual RAM, this value is 0 in RAM block 1, 6 in RAM block 2, and 10 in RAM block 4.

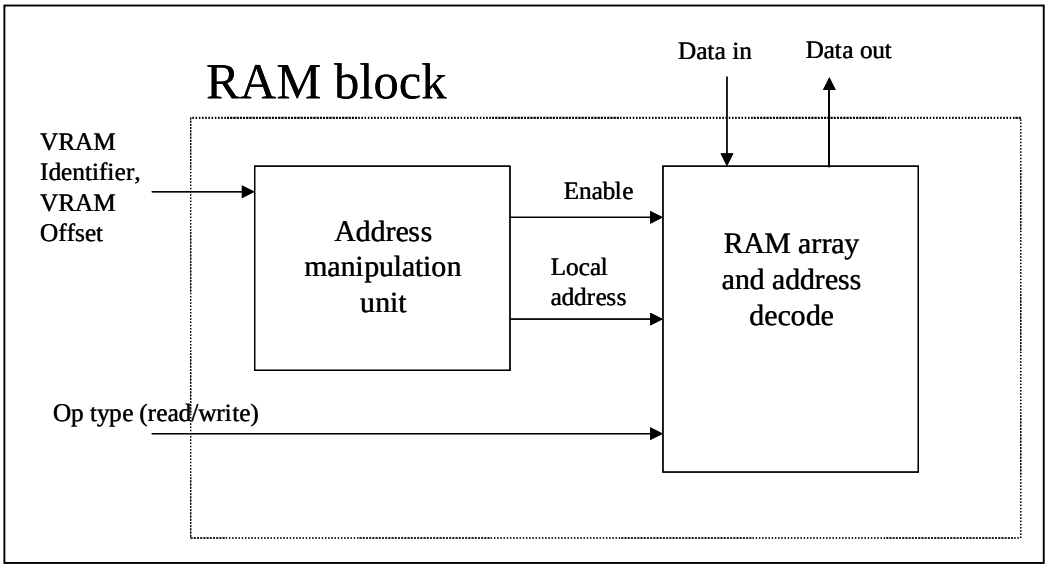


Figure 34: Physical RAM block with associated addressing support.

As shown in Figures 34 and 35, each RAM block is equipped with an address manipulation unit comprising an address mapping table and logic to determine whether the RAM block is involved, and if so, the local address. In step 1, the VRAM identifier is used as an index or as an associative tag that selects a table entry that must be present whenever the corresponding physical RAM implements a chunk within the VRAM. While address manipulation steps 2 and 3 above are specified in their general form, the logic involved can be simplified if the begin and end offsets, base_address and chunk size in each RAM block are power-of-2 sizes. An implementation may thus choose to trade flexibility for efficiency.

For a single-word load reference, only a single physical RAM returns a result. It is assumed that all other physical RAMs return zero. The natural action of the result return tree is to OR these values yielding a correct returned result.

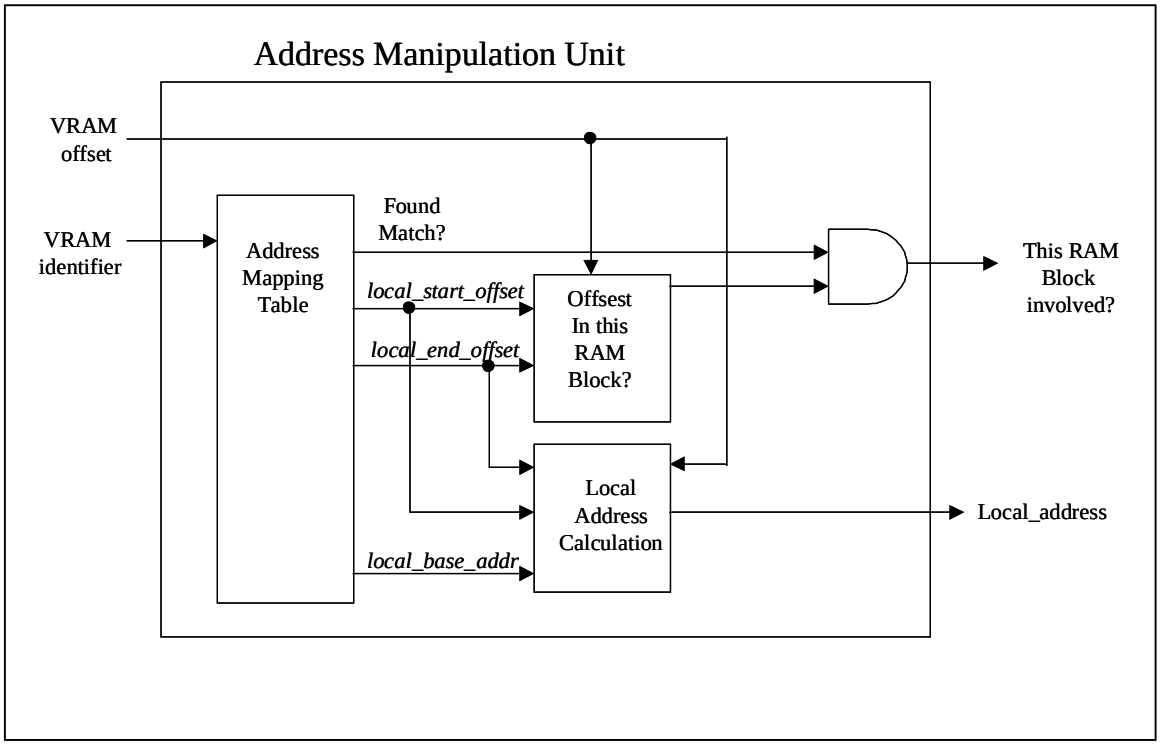


Figure 35: Address Manipulation Unit.

6.4.2 Memory Access Width and Size

The discussion up to this point assumes that each virtual RAM access a fixed-size word that move through each pipeline stage of the memory network in one cycle. The assumption has been that a memory access or result merge tree has a word-wide data-path, and each RAM block has word-wide access port or ports. Figure 36 illustrates such an example with an access network data-path for the green virtual RAM. In addition to the word-wide data, each link carries other control information such as route tag. The link-width markings in the figure only refer to the width of the data component. A design may provide additional wires for the control information, or reuse the data wires, shipping the route tag and other control information in prior cycles.

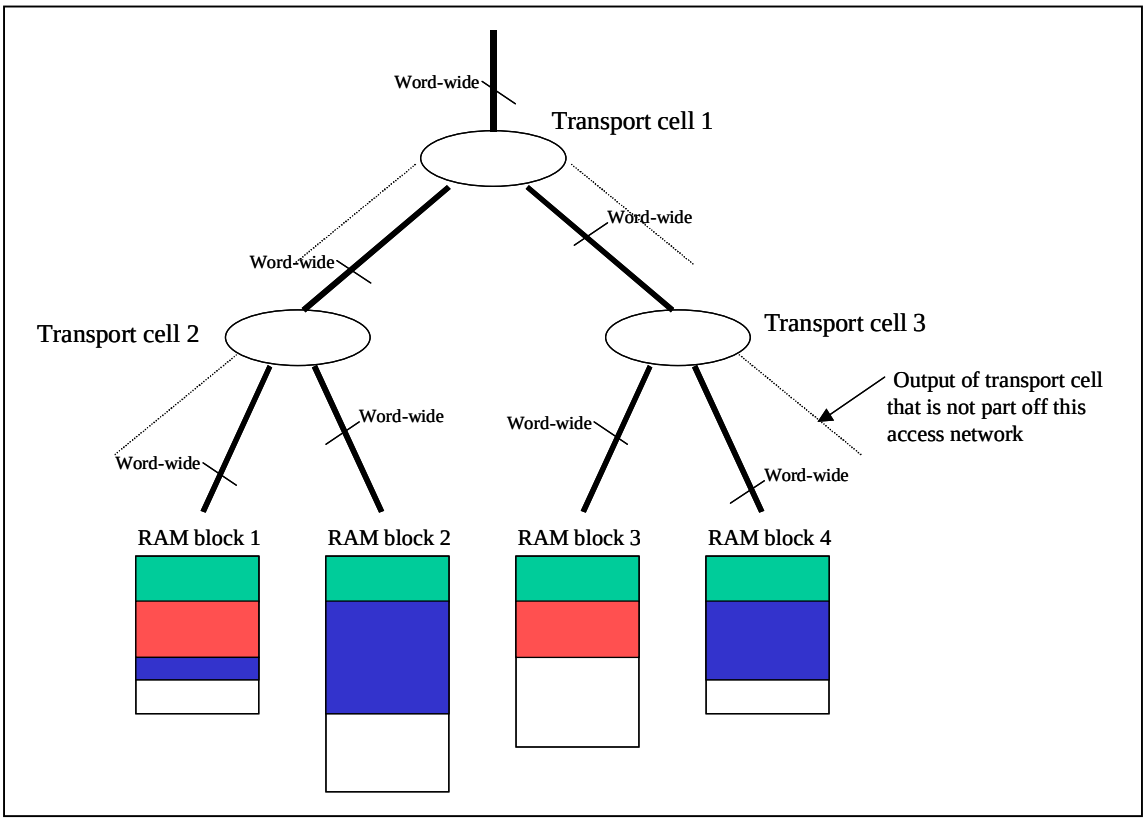


Figure 36: Simple memory access network data-path with word-wide data-paths throughout the access tree, and word-wide memory ports.

Next, consider an example where each RAM block supplies a slice of the word width, and multiple RAM block results are concatenated to provide the full word. Figure 37 illustrates the data-path of such a design for the green virtual RAM where the RAM blocks are paired, 1 with 2, and 3 with 4, to provide the full word-width. A read access will result in a pair of RAM blocks supplying results. Again as in the case of Figure 36, Figure 37 only emphasis the width of the data-path portion of each link. Each pair of half-word wide paths is latency-equalized.

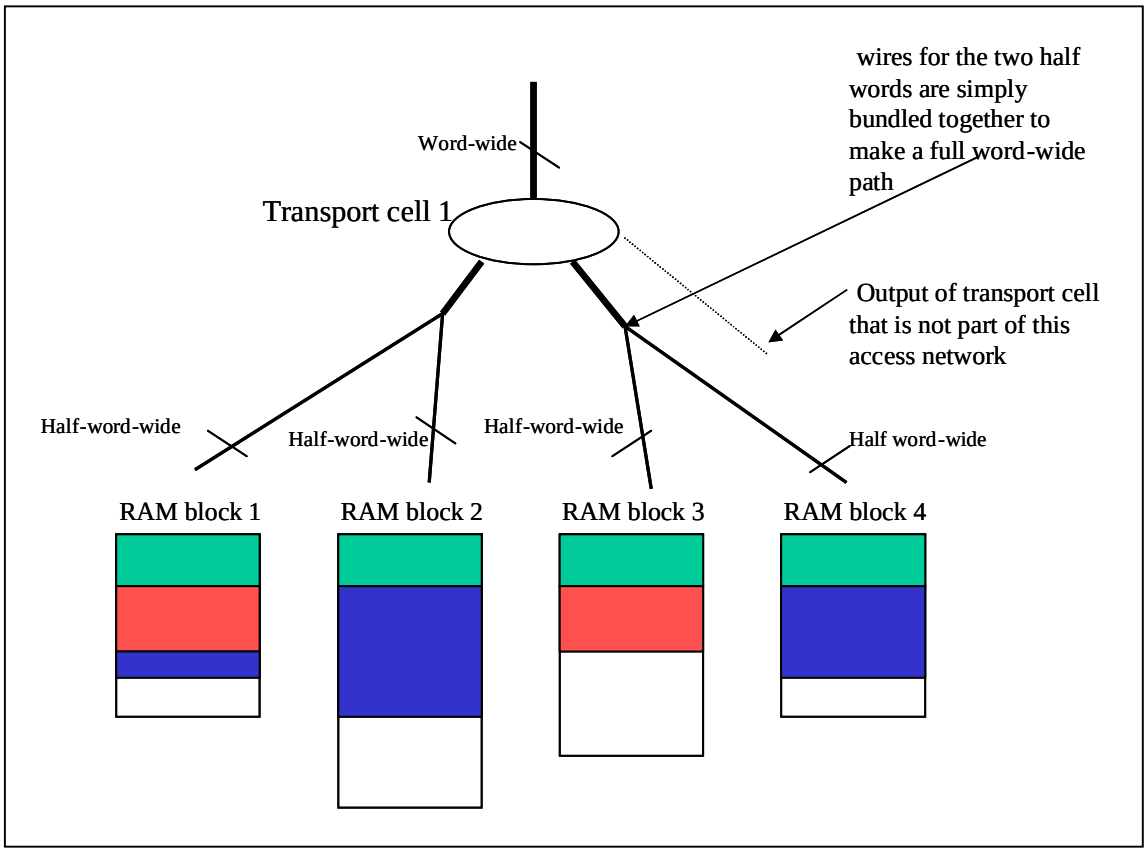


Figure 37: Memory access networks with non-uniform data-path width along parts of the access network.

6.4.3 Multi-cycle Access Address Manipulation Mechanism

This section describes memory addressing hardware that supplies a stream of data over multiple clock cycles. Consider an example similar to Figure 36, but where an access may reference a stream of data. Furthermore, the contiguous chunk of a virtual RAM referenced in one access may straddle multiple RAM blocks. More logic is needed to determine when (on which clock cycle(s) of a multiple-cycle data-stream) each RAM block is accessed.

Specifically, consider a design where a processor again makes a memory access with an address of the form <*virtual RAM identifier*, *offset within virtual RAM*> but also provides an access size parameter: *acc_size*. This address is broadcast through the virtual RAM's access network to all its RAM blocks. At *each* physical RAM, the following actions happen in its address manipulation unit.

1. The (*virtual RAM identifier*) is used to lookup an *address mapping table*. If a match is found, the following information is read out of the address mapping table:

   (*local_start_offset*, *local_end_offset*, *local_base_addr*)

2. Check whether the RAM block is involved:

   (*local_start_offset* <= (*offset within virtual RAM* + *acc_size*)

   AND

   (*offset within virtual RAM* < *local_end_offset*)

3. If the check in step two returns true, the RAM block has (part of) the referenced memory. It does the following computation to determine the local addresses of the locations/words involved, the number of locations involved, and how many words in the access precedes the portion that lies in this RAM block.

   Normalized_start = max (*local_start_offset*, *offset within virtual RAM*)

   Normalized_end = min(*local_end_offset*, (*offset within virtual RAM* + *acc_size*))

   Num_preceding_words = max(0, (*local_start_offset* - *offset within virtual RAM*))

   Num_local_words = *Normalized_end* – *Normalized_start*

   local_address = *local_base_addr*

   + *Normalized_start* – *local_start_offset*

A RAM block waits for *Num_preceeding_words* cycles after the data-streaming begins before supplying or extracting data for reads and writes respectively. The locations involved are the *Num_local_words* starting from *local_address*. This design assumes that the access tree is perfectly balanced, *i.e.* the path from each transport cell to every RAM block connected directly or indirectly to it has to have the same latency, measured in

number of clock cycles. Latency padding can be employed if necessary to ensure that this is the case.

As before, the specification provided here is structured for clarity. Many equivalent forms can be derived through algebraic manipulation. For a number of low-level hardware reasons, one form may be preferred over another. Similar to the single word/cycle access case, each RAM block is augmented with an address manipulation unit, although details of its computation vary slightly. Furthermore, with each RAM block, control is provided to decide when to supply data to or extract data from the data stream.

In general, a stream of operands (of length *acc_size*) may be return from multiple physical RAMS. Assuming that on each cycle that a physical RAM should not return an operand, it returns the value zero, then the result return tree will naturally merge streams from multiple physical RAMS into a single desired result stream using a distributed and pipelined OR.

6.4.4 Alternate Address Manipulation Mechanism

An alternate address manipulation scheme uses a unified, physical address to reduce the amount of specialized addressing hardware at each RAM block while giving the compiler greater responsibility and flexibility in generating addresses. It requires that all the RAM blocks are of the same power-of-two size, and has limited data streaming support. The physical address scheme concatenates a RAM block ID with offset within the RAM block into a linear address space. Each RAM block, when determining whether it is involved in a simple one-word access, simply extracts higher order bits of an access' physical address to form a block ID and check whether that matches its own block ID. The main hardware support associated with each physical RAM block is a programmable block ID register, and ID match logic.

Any translation from the local, virtual RAM addressing scheme into the physical address scheme is handled outside of the memory access structure, in an address manipulation step before a memory access is initiated. Typical manipulations include adding an offset address, or permuting bit-fields to achieve the effect of spreading a virtual RAM across several physical RAM blocks. The spatial compiler is responsible for inserting the address manipulation when it maps virtual RAMs to physical RAMs.

7 Related work

Research on reconfigurable systems and reconfigurable logic has been active for over a decade. Reconfigurable system research efforts include GARP [12, 31], SCORE [13, 47], PipeRench [10, 11, 25, 26, 61, 62], RaPiD [15, 16, 19], RAW [3, 6, 41, 65, 69] and SmartMemories [45]. These system-level research projects all describe high-performance computing systems that exploit statically configurable hardware to implement spatial computation. A number of research efforts have focused on the more limited use of reconfigurable hardware as programmable functional units in an otherwise conventional microprocessor. Research efforts in that area include PRISC[54], Chimaera[30] and AEPIC[64].

A number of start-ups, such as QuickSilver, Chameleon and Morphics, have been advertising break-through reconfigurable technology, particularly for telecommunication applications. Unfortunately, the paucity of publicly available technical information makes it impossible to have a clear technical picture of their (proposed) offerings.

Lower level reconfigurable logic research efforts such as those undertaken by Rose and his students [7, 8, 58-60, 70] focus on identifying ideal low-level reconfigurable logic blocks as well as efficient interconnect and switch technologies to support needed configurable wiring. Low-level research efforts are coupled to the steady evolution of FPGA products that offer greater circuit density, performance, and programming flexibility with each generation.

The XIMD architecture[49, 71] is an extended VLIW architecture that can "dynamically partition its resources to support concurrent execution of multiple instruction streams" [71]. In essence, an XIMD chip can be used as a single VLIW, or be partitioned up into multiple processors executing in the MIMD mode. At this abstract level, it shares similar capabilities to ACRES' remote branch architecture. At a low, mechanism level, however, the two approaches are different. The XIMD architecture shares conditional code and synchronization variables between its subunits, with each subunit making its own control decisions. Appropriate arrangement of code in the subunit results in either VLIW-like behavior, when they execute similar control code, or MIMD-like behavior, when they make distinct control decisions.

Software for high-level synthesis is another closely related research area. Early efforts include the Cathedral at IMEC[46, 50] and the Yorktown Silicon Compiler System[9]. The PICO project [37, 63] at HP Labs developed an automated hardware synthesis tool that generates a highly specialized parallel solver for a given application that is specified in C. PICO's specialized solvers are used within custom ASICs for applications needing high performance, low cost, and low power. PICO incorporates complex transformations that, for a given source, generate efficient hardware for that source. PICO automates a search for a best design within a space of solutions and generates and evaluates many more design options than any human design team. The SPARK project at UC Irvine represents another effort in this area[28]. Recent work on synthesizing hardware from term rewriting specification[32, 33, 53] represents another interesting approach to high-level hardware description and synthesis.

ACRES work also builds upon prior research in parallelizing compilers. Parallelizing compilers take sequential input code and represent that code in a form that exposes parallelism. Traditionally, compilers perform optimizations that increase performance by decreasing the number and cost of operations performed. In addition, parallelizing compilers enhance parallelism through code transformations. Important prior related work includes work on instruction –level parallelism [22, 34] [17] [43].

Not all forms of parallelism are easily derived from sequential code. Programs are often explicitly represented in a parallel form using a variety of approaches. Stream-C [23, 24] is a programming language as well as a compiler. The Stream-C language is a variant of C that supports the representation of explicit parallelism. The Stream-C compiler automatically maps streaming applications written in the Stream-C language onto FPGAs. StreamIT[27, 67] is another stream oriented language primarily targeted at "latency exposed architectures" such as RAW.

Software and compilation is a significant aspect of the RAW project. By baring a number of important micro-architectural features to software [69], the RAW architecture requires sophisticated compilation software to assist in dealing with issues of data and code mapping, as well as communication scheduling. The work by Barua[6] and Lee[41] within the RAW project are some early examples of compiler efforts in those areas, dealing with issues of disambiguating memory access, distributing data, and scheduling instructions in a distributed execution environment.

A number of projects have explored architectures that are better suited for distributed implementations. Clustered VLIW is employed in machines like the Multiflow[14] to partition the register file and functional units of a VLIW processor into clusters. This enables a high degree of parallel access while avoiding an expensive fully multi-ported register file. The approach has been widely used in application specific domains, such as DSP and media processors[57]. Even general purpose superscalar processors today have separate integer, floating point and sometimes predicate register files. Researchers have investigated various architectural and compilation issues related to clustered (*i.e.* partitioned and distributed) register files. Examples in this area include methods for allocating data and operations to clusters[18], managing the transfer of data between clusters[56], assignment and scheduling[21], code-generation[36] and other issues[35].

Researchers have also been looking into microprocessor architectures that run multiple threads simultaneously on clustered and decoupled processor core resources. An early work in this area is M-machine[38]. Subsequent work includes the MAJC[68] processor. Grid Processor Architectures[48] represent a more recent effort that combines limited distribution of simple control over a grid of ALUs and direct data-path connectivity between the ALUs. At a different end of the sharing spectrum, numerous projects, such as the Piranah[5], Hydra[29] and MPOC[55], have looked into various forms of SMP-on-a-chip, sometimes called chip-multiprocessor (CMP), that share chip resource at a coarser-grain level, typically starting at the caches or memory.

Conclusions

Reconfigurable systems provide a powerful new architectural approach that is maturing from a technology for implementing low-level glue logic into a technology that provides a flexible programmable high-performance processing capability for specialized applications. The explosive growth of gate count within inexpensive reconfigurable devices and improvements in software for automatically configuring such systems will continue to enhance the advantages of this technology.

The ACRES platform integrates hardware features from microprocessors, ASICs, and field programmable logic as well as software features from compilers, CAD, and from software for reconfigurable logic. From microprocessors, ACRES incorporates efficient execution of lengthy and complex program sequences. From ASICs, ACRES incorporates efficient parallel execution using distributed control and data. Software for compilation and high-level CAD synthesis couple a uniform high-level programming model to microprocessor and ASIC architectures and thus provide a high-level programming abstraction that can be used to program the technology. Field programmable hardware and software technology maps efficient custom hardware structures onto off-the-shelf chips.

While ACRES draws substantial strength from earlier work, it also incorporates several novel features. ACRES' spatial compilation flow is unique in its use of automatic early spatial planning and late scheduling. While classical approaches develop an architecture for a custom processor without knowledge of a spatial plan, ACRES uses a spatial plan to influence important design choices when specifying this architecture. This unique design flow provides a system-level approach that opens up new opportunities for dealing with the old but often still difficult problem of meeting timing closure. At the same time, ACRES design flow brings interesting challenges in identifying the best heuristics for performing the various placement and mapping steps.

The ACRES platform further incorporates a number of interesting hardware technologies that facilitate the design of distributed and scalable specialized processors and mesh well with its unique compilation flow. These include critical components such as the switched and pipelined interconnect, the reconfigurable memory and the remote branch technologies. Each of these technologies uses a combination of hardware and software to provide a higher degree of scalability without incurring negative effects arising from reduced clock cycle time.

ACRES research progress highlights unique requirements representing rich areas for future research. Spatial compilation requires the automatic design of an application specific physical architecture – how many functional units, of what types, interconnected in what ways? How does one go about choosing between different possible architectures? There are many open questions about specifics of the ACRES spatial compilation flow. For example, when should virtual operation to physical component mapping be done? Many individual spatial compilation steps represent difficult research problems. For instance, can traditional place and route heuristic be adapted for spatial planning and the routing of sharable data transport? Devising and evaluating spatial compilation strategies will provide exciting research for years to come.

The ACRES architecture is based on the premise that only a few chip types are sufficient to cover a substantial fraction of high-performance embedded applications. Final customization of these chips is based on an enhanced programmable technology that integrates conventional programming, CAD, and programmable logic configuration. Research that proposes a few specific candidate chip structures and measures their suitability for a range of applications is needed to prove this point. Such research will inevitably result in improvements to proposed chip hardware.

On-line reconfiguration represents another research opportunity. Static configuration misses opportunities to exploit information learned as execution progresses. As computation progresses, there is opportunity to redefine (*i.e.* reconfigure) the physical architecture at run time. Hardware can be reconfigured to accommodate changing application requirements. This requires both the development of on-line spatial compilation techniques as well as architectures that support on-line reconfiguration.

Reconfigurable systems will find increasing acceptance within the marketplace. If reconfigurable logic systems are to proliferate they must move beyond hand-customized designs layered on low-level components. Approaches are needed to design very complex computer systems using a high-level software approach rather than a low-level hardware approach. The ACRES platform provides an approach that both allows easy access to programmable logic through high-level language programming as well as preserves the efficiency of reconfigurable logic technology. This research is preliminary and much research remains to fulfill this vision.

References

[1] S. Abraham, et al. Efficeint Design Space Exploration in PICO. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, San Jose, CA, 2000, pp. 71-79.

[2] S. Aditya, B.R. Rau, and R. Johnson. *Automatic Design of VLIW and EPIC Instruction Formats*. HP Labs Technical Report, HPL-1999-94, 1999.

[3] A. Agarwal. Raw Computation. *Scientific American*, August 1999.

[4] J. Babb, R. Tessier, and A. Agarwal. Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, Los Alamitos, CA, 1993, pp. 142-151.

[5] L.A. Barroso, et al. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, BC, Canada, 2000.

[6] R. Barua, et al. Compiler Support for Scalable and Efficient Memory Systems. *IEEE Transactions on Computers*, Nov 2001.

[7] V. Betz and J. Rose. Effect of the Prefabricated Routing Track Distribution on FPGA Area-Efficiency. *IEEE Transactions on VLSI*, Sept. 1998. **6**(3): pp. 445-456.

[8] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. 1999: Kluwer Academic Publishers.

[9] R. Brayton, et al. The Yorktown Silicon Compiler System. *Silicon Compilation*: pp. 204-310.

[10] M. Budiu and S.C. Goldstein. Fast Compilation for Pipelined Reconfigurable Fabrics. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA 1999)*, 1999.

[11] S. Cadambi and S.C. Goldstein. Fast and Efficient Place and Route for Pipeline Reconfigurable Architectures. In *Proceedings of the ICCD '00*, 2000.

[12] T.J. Callahan, J.R. Hauser, and J. Wawrzynek. The Garp Architecture and C Compiler. *IEEE Computer*, April. **33**(4): pp. 62-69.

[13] E. Caspi, et al. A Streaming Multi-Threaded Model. In *Proceedings of the Third Workshop on Media and Stream Processors (MSP-3)*, Austin, Texas, 2001.

[14] R.P. Colwell, et al. Architecture and implementation of a VLIW supercomputer. In *Proceedings of the Supercomputing '90*, 1990, pp. 910-919.

[15] D.C. Cronquist, et al. Specifying and Compiling Applications for RaPiD. In *Proceedings of the Symposium on Field-Programmable Custom Computing Machines (FCCM'98)*, 1998.

[16] D.C. Cronquist, et al. Architecture Design of Reconfigurable Pipelined Datapaths. In *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, Atlanta, GA, 1999, pp. 23-40.

[17] J.C. Dehnert and R.A. Towle. Compiling for the Cydra 5. *The Journal of Supercomputing*, May. **7**(1/2): pp. 181-228.

[18] G. Desoli. *Instruction Assignment for Clustered VLIW DSP Compilers: A New Approach.* HP Labs Technical Report, HPL-98-13, 1998.

[19] C. Ebeling, et al. Mapping Applications to the RaPiD Configurable Architecture. In *Proceedings of the Field-Programmable Custom Computing Machines (FCCM-97)*, 1997.

[20] J.R. Ellis, *Bulldog: A Compiler for VLIW Architectures.* 1985, Cambridge, Massachussetts: The MIT Press.

[21] S.B. Emre Özer, Thomas M. Conte. Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures. In *Proceedings of the 31th Annual International Symposium on Microarchitecture (MICRO-31)*, Dallas, Texas, 1998, pp. 308-315.

[22] J.A. Fisher and B.R. Rau. Instruction-level Parallel Processing. *Science*, September. **253**(5025): pp. 1233-1241.

[23] J. Frigo, M.B. Gokhale, and D. Lavenier. Evaluation of the Stream-C C-to-FPGA Compiler: An Applications Perspective. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA 2001)*, Monterey, CA, 2001, pp. 134-140.

[24] M.B. Gokhale, et al. Stream-Oriented FPGA Computing in the Stream-C High Level Language. In *Proceedings of the Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*, 2000.

[25] S.C. Goldstein, et al. PipeRench: A Reconfigurable Architecture and Compiler. *IEEE Computer*, April 2000. **33**(4).

[26] S.C. Goldstein, et al. PipeRench: A Coprocessor for Streaming Multimedia Accleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, 1999, pp. 28-39.

[27] M.I. Gordon, et al. A Stream Compiler for Communication-Exposed Architectures. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, 2002.

[28] S. Gupta, et al. SPARK : A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations. In *Proceedings of the International Conference on VLSI Design*, 2003.

[29] L. Hammond, et al. The Stanford Hydra CMP. *IEEE MICRO*, March-April 2000.

[30] S. Hauck, et al. The Chimaera Reconfigurable Functional Unit. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, 1997, pp. 87-96.

[31] J.R. Hauser and J. Wawrzynek. A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Napa, CA, 1997.

[32] J.C. Hoe and Arvind. Hardware Synthesis from Term Rewriting Systems. In *Proceedings of the X IFIP International Conference on VLSI*, Lisbon, Portugal, 1999.

[33] J.C. Hoe and Arvind. Synthesis of Operation-Centric Hardware Descriptions. In *Proceedings of the International Conference on Computer Aided Design*, San Jose, California, 2000.

[34] W.W. Hwu, et al. The Superblock: an Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, May. **7**(1/2): pp. 229-248.

[35] M.F. Jacome, G.d. Veciana, and S. Pillai. Clustered VLIW Architectures with Predicated Switching. In *Proceedings of the Design Automation Conference*, 2001, pp. 696-701.

[36] K. Kailas, K. Ebcioglu, and A. Agrawala. CARS: A New Code Generation Framework for Clustered ILP Processors. In *Proceedings of the 7th High-Performance Computer Architecture (HPCA-7)*, Nuevo Leone, Mexico, 2001, pp. pages 133-143.

[37] V. Kathail, et al. PICO (Program In, Chip Out): Automatically Designing Custom Computers. *IEEE Computer*, September 2002. **35**(9): pp. 39-47.

[38] S.W. Keckler, et al. Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, Barcelona, Spain, 1998, pp. 306-317.

[39] K. Lalgudi and M. Papaefthymiou. Fixed-phase retiming for low power design. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 1996, pp. 259--264.

[40] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*: ACM, 1988, pp. 318-327.

[41] W. Lee, et al. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, 1998.

[42] C.E. Leiserson and J.B. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, June 1991: pp. 5--35.

[43] P.G. Lowney, et al. The Multiflow Trace Scheduling Compiler. *The Journal of Supercomputing*, May. **7**(1/2): pp. 51-142.

[44] S.A. Mahlke, et al. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 1992, pp. 45-54.

[45] K. Mai, et al. Smart Memories: A Modular Reconfigurable Architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, Vancouver, 2000.

[46] H.D. Man, et al. CATHEDRAL-II: a Silicon Compiler for Digital Signal Processing Multiprocessor VLSI Systems. *Design & Test of Computers*, 1986: pp. 13-25.

[47] Y. Markovskiy, et al. Analysis of Quasi-Static Scheduling Techniques in a Virtualized Reconfigurable Machine. In *Proceedings of the Tenth ACM International Symposium on Field-Programmable Gate Arrays (FPGA 2002)*, Monterey, CA, 2002, pp. 196-205.

[48] R. Nagarajan, et al. A Design Space Evaluation of Grid Processor Architectures. In *Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO-34)*, 2001, pp. 40-51.

[49] C.J. Newburn, A.S. Huang, and J.P. Shen. Balancing Fine- and Medium-grained Parallelism in Scheduling Loops for the XIMD architecture. In *Proceedings of the Architectures and Compilation Techniques for Fine and Medium Grain Parallelism. (IFIP WG10.3 Working Conference)*: Elsevier Science Publishers B.V. (North-Holland), 1993, pp. 39-52.

[50] S. Note, et al. Cathedral III: Architecture driven high-level synthesis for high throughput DSP applications. In *Proceedings of the 28th ACM/IEEE Design Automation Conference, DAC 91*, San Francisco, CA, 1991, pp. 597 -- 602.

[51] M.C. Papaefthymiou. Understanding Retiming Through Maximum Avarage-Delay Cycles. *Mathematical Systems Theory*, 1994. **1**(27): pp. 65-84.

[52] B.R. Rau. Iterative Modulo Scheduling. *International Journal of Parallel Processing*, February. **24**(1): pp. 3-64.

[53] J. Ray and J.C. Hoe. High-Level Modeling and FPGA Prototyping of Microprocessors. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'03)*, Monterey, California, 2003.

[54] R. Razdan and M.D. Smith. A High-Performance Microarchitecture with Hardware-Programmable Functional Units. In *Proceedings of the MICRO-27*, San Jose, CA, 1994, pp. 172-179.

[55] S. Richardson. *MPOC: A Chip Multiprocessor for Embedded Systems*. HP Labs Technical Report, HPL-2002-186, 2002.

[56] S. Rixner, et al. Memory Access Scheduling. In *Proceedings of the 27th International Symposium on Computer Architecture*, Vancouver, BC, Canada, 2000.

[57] S. Rixner, et al. Register Organization for Media Processing. In *Proceedings of the Sixth Annual International Symposium on High-Performance Computer Architecture*, 2000, pp. 375-386.

[58] J. Rose, A.E. Gamal, and A. Sangiovanni-Vincentelli. Architecture of Field-Programmable Gate Arrays. *Proceedings of the IEEE*, July 1993. **81**(7): pp. 1013-1029.

[59] J.S. Rose and S. Brown. Flexibility of Interconnection Structures for Field-Programmable Gate Arrays. *IEEE JSSC*, March 1991. **26**(3): pp. 277-282.

[60] J.S. Rose, et al. Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency. *IEEE JSSC*, October 1990. **25**(5): pp. 1217-1225.

[61] H. Schmit. Incremental Reconfiguration for Pipelined Applications. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1997, pp. 47-55.

[62] H. Schmit, et al. PipeRench: A Virtualized Programmable Datapath in 0.18 Micron Technology. In *Proceedings of the 2002 IEEE Custom Integrated Circuits Conference (CICC)*, 2002.

[63] R. Schreiber, et al. *PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators.* Technical Report, HPL-2001-249, 2001.

[64] S. Talla. *Adaptive Explicitly Parallel Instruction Computing.* New York University, Department of Computer Science, PhD Dissertation, 2000.

[65] M.B. Taylor, et al. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. *IEEE Micro*, Mar/Apr 2002.

[66] R. Tessier, et al. The Virtual Wire Emulation System: A Gate-Efficient ASIC Prototyping Environment. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA '94)*, Berkeley, CA, 1994.

[67] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the 2002 International Conference on Compiler Construction*, Grenoble, France: Springer-Verlag LNCS.

[68] M. Tremblay. MAJC: AN Architecture for the New Millennium. In *Proceedings of the Hot Chips 1999*, Stanford, CA, 1999, pp. 275-288.

[69] E. Waingold, et al. Baring it all to Software: Raw Machines. *IEEE Computer*, September 1997: pp. 86-93.

[70] S. Wilton, J. Rose, and Z. Vranesic. The Memory/Logic Interface in FPGAs with Large Embedded Memory Arrays. *IEEE Transactions on VLSI*, March 1990. **71**(1): pp. 80-91.

[71] A. Wolfe and J.P. Shen. A Variable Instruction Stream Extension to the VLIW Architecture. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, 1991, pp. 2-14.