Balancing Computation and Memory in High Capacity Reconfigurable Arrays

by

Stylianos Perissakis

Dipl.Eng. (National Technical University of Athens) 1992
M.S. (University of California, Berkeley) 1996

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor John Wawrzynek, Chair
Professor David Patterson
Professor John Strain

2000

The dissertation of Stylianos Perissakis is approved:

Chair Date

Date

Date

University of California at Berkeley

2000

Balancing Computation and Memory in High Capacity Reconfigurable Arrays

Copyright 2000
by
Stylianos Perissakis

Abstract

Balancing Computation and Memory in High Capacity Reconfigurable Arrays

by

Stylianos Perissakis
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor John Wawrzynek, Chair

Reconfigurable arrays have been used to speed up computational tasks, some times achieving orders of magnitude of improvement either in cost/performance or raw performance. An integral part of such systems is a significant amount of memory, distributed among multiple reconfigurable chips. The current scaling trend of VLSI technology, that allows packing of more and more gates on a single silicon die, tends to throw such architectures out of balance, unless significant amount of memory is integrated on the die, close to the processing elements. Embedded DRAM technology, integration of dynamic RAM with logic circuits on the same chip, offers a unique opportunity to balance the processing and memory resources in large reconfigurable chips.

In this thesis I evaluate the potential of embedded DRAM technology in the context of a reconfigurable architecture. I examine the design space for embedded DRAM macros suitable for reconfigurable architectures and present a point design, the *Configurable Memory Block* (CMB) for the *Trumpet* FPGA. With the aid of sample application kernels, I evaluate the density, performance and energy advantages of embedded DRAM technology over the conventional technology of small internal SRAM banks and larger external memory. Based on a set of typical applications, I develop a methodology for choosing a cost/performance-optimum design point for reconfigurable architectures similar to Trumpet. I find that FPGAs incorporating embedded DRAM banks can demonstrate high performance and energy efficiency, in addition to the obvious integration benefits resulting from the merging of the processing and memory elements on the same chip.

Professor John Wawrzynek
Dissertation Committee Chair

Contents

List of Figures

List of Tables

Acknowledgements

This research has been motivated by two large research efforts in U.C.Berkeley: the BRASS project, led by my advisor, John Wawrzynek, that investigates reconfigurable computing; and the IRAM project, led by professor David Patterson, that examines ways to utilize embedded DRAM technology. I have benefited greatly from my exposure to the high quality of research in both projects. The ideas for this thesis, that focuses on the synergy of reconfigurable computing and embedded DRAM technology, came out of these research efforts.

This work would not have been possible without the efforts of a large number of people. Design and implementation of the Trumpet prototype required the contributions of many colleagues over two continents: the reconfigurable portion of Trumpet was designed and implemented by William Tsu, Kip Macy, Atul Joshi and Randy Hwang. The memory portion was based on the DRAM macro developed by LG Semicon of Korea. Young Choi, Yangsung Joo and Kyungnam Park have been our daily contacts across the Pacific. Staff engineer Jim Beck was always available to contribute his system-level expertise and real-world experience towards the prototype specification, PCB development and testing strategy.

As with most hardware projects, software tools have been instrumental in the early development of "driver" applications and the verification of the hardware design. The BOOM system, developed primarily by Nick Weaver and Michael Chu, was our in-house programming environment for Trumpet. In addition, a number of programming tools were developed by Dr. André DeHon. As part of his postdoctoral work at Berkeley, André has juggled many roles, from system architect to software developer. During the development of Trumpet he was our project manager, as well as a source of information and ideas. Many thanks to all these people, whose contributions made this work possible.

Finally, I would like to thank my advisor, John Wawrzynek, for motivation, guidance and support over the several years that I have been participating in his research group.

Primary funding for this work came from DARPA Information Technology Office, contract number DAB63-96-C0048.

Chapter 1

Introduction

1.1 Reconfigurable computing

Field-Programmable Gate Arrays, or FPGAs, have been used in digital systems for about 15 years. At the time of their conception the FPGA was viewed more as a larger, better programmable logic device (PLD), than an actual computing device. FPGAs replaced the product-term logic common in programmable logic arrays (PLAs) with a lookup table (LUT), and added more register storage and a more flexible interconnect. In addition, a more elaborate design entry environment was generally provided. This generality, together with advances in process technologies and the resulting increased logic capacity of silicon dies, allowed FPGAs to replace multiple small or medium scale integration logic chips[20].

For several years, the primary application of FPGAs has been just that – collecting a number of MSI chips into a single package, to allow area and cost savings in large, complex digital system boards. In addition, the reprogrammability of FPGAs has allowed designers to introduce last-minute fixes of design flaws, by just reprogramming the FPGAs. Finally, the capability of implementing a logic design without the need to fabricate a special chip has allowed the rapid prototyping and low cost implementation of digital systems.

With the ever increasing logic capacity of silicon dies and the corresponding increase in circuit performance, it is becoming possible for FPGAs to implement not just random, low performance logic circuits, but high performance compute engines for demanding computational tasks as well. Lately, FPGAs are often viewed as general purpose programmable computing devices, not unlike microprocessors. Implementation of computational tasks on FPGAs has been termed *reconfigurable computing* and has been shown to have promising characteristics, when compared to other programmable architectures, such as microprocessors or DSPs[28].

With a large number of independently programmable logic elements, FPGAs can take advantage of parallelism in several ways. Typical inner loops in conventional processors may consist of a dozen instructions and take several cycles per iteration, even on modern superscalar processors. Of those instructions, typically one or two will perform the actual computation, while the others will be used for tasks like pointer arithmetic, memory accesses, loop counters and conditional branches. On an FPGA, all these tasks can often be executed in parallel, leading to much higher performance – often just one cycle per iteration. This effect is equivalent to a high degree of *instruction level parallelism* (although the definition of an "instruction" in FPGAs is much different than that for microprocessors). Also, when significant *data parallelism* is available in the application, an FPGA can use multiple identical functional units and employ a SIMD-style programming model. Finally, FPGAs can take advantage of *task level parallelism*, at a much coarser grain, by implementing more than one compute kernels on a single device at the same time, to the extent permitted by the logic capacity of the device.

Most conventional architectures offer a relatively small set of primitive operations, like addition, multiplication, shift, and so on, that can be executed quickly (often within a clock cycle). When the operations in the application do not fall within this limited set, they must be synthesized from the ones available in the architecture, frequently resulting in long latencies. For example, arbitrary permutation of the bits in one word, common in encryption algorithms, takes a series of shift and mask operations on most microprocessors. Such operations are trivial to implement on an FPGA. In the same way, evaluating a wide boolean expression on a microprocessor will typically require a sequence of boolean operations and conditional branches, utilizing only a single bit of the datapath for a number of cycles. In most FPGAs, boolean expressions can be evaluated with one or two LUTs in a single cycle.

Even with common operations, FPGAs can often reduce the required complexity when one of the operands is known at compile time, by customizing, or *specializing* the functional unit around the value of that operand. For example, multiplication of a variable with a compile-time constant can often be converted to a small number of additions, which is much cheaper to implement than a general multiplication.

Another form of specialization takes advantage of narrow word widths to reduce the amount of resources required for a given task. In most microprocessors the datapath width is fixed (usually 32 or 64 bits), which leads to a waste of area and power when the data are much narrower than that. FPGAs can configure the width of functional units to exactly match that of the data. Recently, most microprocessor manufacturers have introduced instruction set extensions – like Intel's MMX, or Sun's VIS – that allow a wide datapath to process multiple narrow data elements in parallel. Although such extensions go a long way towards eliminating this waste, it is not as flexible as common FPGAs, which can adapt to any odd data width[53, 78].

Mainstream processor architectures dedicate a large amount of resources to multiplex the execution of instructions on a limited number of functional units. This overhead is most pronounced in high-end processors, that require significant amounts of control logic to take advantage of instruction-level parallelism. The silicon real estate allocated to functional units in a modern microprocessor is strikingly small – typically much less area will be allocated to the functional units than to control logic[4, 67]. In FPGAs, where the computation is usually implemented as a fully "spatial" data flow, the required control logic is much more limited. With the flexibility and uniformity of FPGA fabrics, it is possible to allocate more resources to functional units for regular computations. Unlike with microprocessors, control overhead is only paid for when the application requires it. It is not uncommon for computations implemented on an FPGA to consist almost entirely of datapath and very little control logic.

FPGAs of course also have a significant overhead to pay for their generality. Implementing a gate as a lookup table is inherently slower, takes up more area and burns more energy, than an equivalent gate in a full-custom datapath. In the same way, programmable interconnect requires a large number of switches, which add to the area and energy dissipation of the design, while they further degrade the switching performance. Still, DeHon shows that the peak computational density of FPGAs tends to be between one and two orders of magnitude higher than that of microprocessors[28, 29]. Experience shows that FPGAs are able to sustain a large fraction of their peak density on regular, data-intensive computations, while they don't fare as well in more irregular, control-dominated tasks.

Application of FPGAs to computing tasks to date has been rather limited, largely because with the current state of design automation FPGAs tend to be hard to program. The user/programmer needs to be both an application expert and programmer, as well as a logic designer. In most systems to date, "programming" is performed either through direct schematic capture, or through hardware description languages, like VHDL or Verilog. Work is in progress, both at Berkeley[19, 23] and elsewhere[16, 101], on hardware design based on common high-level computer languages.

Despite these disadvantages, reconfigurable computing has been on a promising track. As early as 1989, researchers have built large computing systems based on multiple FPGA chips[8, 12, 41]. Such systems, often referred to as *custom computing machines* or CCMs, have demonstrated impressive performance in select applications[41, 132], proving that there exists at least a class of applications where the advantages of reconfigurable computing outweigh the shortcomings of the implementation constraints. More recently, commercial vendors have been offering a variety of accelerator boards based on FPGA technology[24, 31, 140]. The natural progression from there is chip-level integration of an FPGA-based coprocessor and a a host processor. Currently there is a significant interest in both industry[119, 127] and academia[47, 101, 139], including Berkeley[48], towards this direction. Such a close integration offers the possibility of eliminating performance bottlenecks, such as the FPGA to processor or FPGA to memory links, resulting in powerful single-chip computing devices that can yield good performance in a large variety of applications, since they combine the advantages of both FPGAs and conventional processors. This trend, from large systems, to single boards, to single chips, is a clear manifestation of Moore's law[83], that predicted the approximate doubling of integrated circuit capacity each year.

1.2 The need for high capacity on-chip memory

The more successful multi-FPGA systems that were built in the past had multiple memories interconnected to the FPGAs, rather than one large monolithic memory system. The original *Splash* [41] consisted of 32 Xilinx 3090 chips, interconnected in a ring topology, and an equal number of SRAM chips, connected to the ring links. *Splash II* [9], designed shortly thereafter, had a more elaborate topology, again with one SRAM per FPGA but with a dedicated memory interface on each FPGA. *Perle-0* [12], the first instance of the *PAM* architecture built at DEC PRL, was a 5×5 Xilinx array, with a single 64-bit wide interface to 4 Mbits of SRAM. However, in the later incarnation, *Perle-1* [132], this shared memory was replaced by 4 independent memory banks, with a total capacity of 4 MBytes.

Many researchers have observed that the memory system is often the limitation

System	Total LUTs	Memory/Compute	Compute/BW	Compute/port
Splash	20,480	205 Bytes/LUT	80 LUTs/bit	640 LUTs
Splash 2	12,800	655 Bytes/LUT	50 LUTs/bit	800 LUTs
Perle 0	3,200	164 Bytes/LUT	50 LUTs/bit	3,200 LUTs
Perle 1	10,240	410 Bytes/LUT	80 LUTs/bit	2,560 LUTs

Table 1.1: Metrics for existing reconfigurable systems (Note 1 Xilinx CLB $\equiv$ 2 4-LUTs)

in application performance. According to the designers of the first Splash system, "bigger, wider and faster memories would be an important objective for a follow-on design"[41]. They also point out that the host should be able to access these memories for I/O. In Splash II, both these problems have been taken care of. Rencher et.al., who used Splash II for an automated target recognition application, observe that "the distributed memory and aggregate memory bandwidth is what makes this implementation possible"[103]. However, they also point out that even better performance would be possible if the memory system supported faster back-to-back read-write cycles (a problem also observed by Abbot et.al.[1]). Finally, the designers of Perle-1, who have also developed a variety of applications for their system, mention that many of these applications utilize the full RAM bandwidth available[132], suggesting that they are memory limited, rather than compute limited.

From the experience of those researchers, it appears that the amount and flexibility of the memory resources found on their systems should be viewed as a minimum requirement for a successful custom computing machine. Table 1.1 summarizes some features of the memory architecture of the above machines. The amount of compute resources is captured as the total number of LUTs available. The ratio of memory capacity to compute resources and the ratio of compute resources to memory bandwidth are the metrics used to indicate the memory-logic balance. The amount of compute resources (or LUTs) divided by the number of memory ports characterizes the *granularity* of the distribution of memory in the system.

Unfortunately, the scaling trend dictated by Moore's law has some negative implications. As the density of FPGAs and memory chips grows, if compute elements and memory remain in separate chips, architectures like the above will tend to become more coarse-grain. Whereas it is easy to maintain a similar or even higher memory-to-compute ratio by just adding more memory chips, the compute-to-bandwidth ratio and the compute resources per memory port are bound to deteriorate. This trend is inevitable, since die periphery grows slower than die area, so larger systems tend to have less I/O per compute element[1]. This imbalance can often lead to bandwidth bottlenecks, making it hard to exploit parallelism: small total bandwidth will eventually limit the achievable data parallelism; and task-level parallelism will also be limited given only a small number of independent interfaces. In the words of the Splash II designers, "as FPGAs accommodate larger and larger designs on a single chip, the probability will grow that more than one part of a given chip's design will need to access memory in the same clock period ... making a single memory port the bottleneck."[17]

Of course, reconfigurable architectures are not the only ones affected by this separation of processing elements and memory. Historically, microprocessors have mitigated the effects of slow and narrow memory interfaces by hiding the main memory behind on-chip SRAM caches. Caches have certainly gone a long way towards keeping the processing pipelines busy, but with the current scaling trend, they become increasingly expensive in terms of the die area they consume. As a case in point, the HP PA8500 processor[11] dedicates three quarters of its die to a 0.5MB instruction cache and a 1.0MB data cache. At the same time, new applications emerge – for example, the multimedia and human interface functions that are of primary importance today tend to stream through large data structures without much temporal locality, rendering caches ineffective[30].

When it comes to reconfigurable architectures, there are additional considerations: with the high degree of data and instruction-level parallelism possible, FPGAs can often saturate a wide memory interface more easily; and to exploit task level parallelism, CCMs need multiple independent memory interfaces. Therefore, the same problems faced by more conventional architectures today are even more severe for FPGA-based machines. For this reason, current FPGAs have integrated small amounts of memory in their architecture. Some can treat the LUT configuration memory as a RAM[142], while others implement small SRAM blocks, up to a few Kbits[7]. Certain FPGAs actually employ both techniques at the same time[141]. Unfortunately, SRAM memories require a large layout area, and so they are in competition with the compute elements for the limited available space. This results in a relatively small on-chip memory available in such devices. For example, the `Virtex XCV3200E` FPGA from Xilinx, probably the largest FPGA available as of this writing, contains 65,536 4-input LUTs and 832 Kbits of memory, implemented as 208 banks

[1]Techniques like area bonding only push the problem from the IC package to the circuit board, since interconnect lines need to fan out sideways eventually.

of 4 Kbits each. Compared to the data in Table 1.1, this device is very memory-poor (or computation-rich). As we will see in Chapter 6, such small SRAM banks work quite well for fine-grain lookup and retiming tasks, but are insufficient for more demanding storage requirements.

Finally, a flexible on-chip memory system of substantial capacity may enable new modes of computation: in the *SCORE* computation model[104] compute kernels are swapped into and out of the FPGA on demand, much like memory pages in a virtual memory system. In such an environment, it is crucial to be able to load FPGA configurations and swap application state as fast as possible. In addition, data streams may need to be buffered when the consumer computation is not resident in the FPGA at the same time as the producer. High bandwidth access to large memory capacity, as well as distributed memory pages, are requirements for such systems to be practical.

Embedded DRAM technology[65] offers the opportunity to integrate a large amount of memory on the same chip as the processing elements. This integration can make it possible to implement multiple fast and wide memory interfaces, providing a very high memory bandwidth to the execution core. However, DRAM technology also has certain shortcomings compared to conventional SRAM and logic technologies – most importantly, DRAM is significantly slower than SRAM, while logic circuits implemented in a DRAM process tend to be slower than the same circuits in a pure logic process. These differences lead to a tradeoff between architectural efficiency, in terms of the number of cycles required to perform a certain task, and circuit efficiency, in terms of the minimum achievable clock cycle time. Although on-chip memory leads to efficient architectures, the shortcomings of the technology may partly offset those benefits.

Performance aside, there are also other benefits associated with integrated single-chip architectures. For small portable systems the reduced part count and overall system size can be as important as the raw performance[72]. The physical size and weight of devices such as cellular telephones or "personal digital assistants" (PDAs) are usually crucial to the success of a product. But even for larger systems, reduced part count usually leads to reduced cost. Although larger chips generally have lower manufacturing yield than smaller ones, leading to increased cost per unit area of silicon, the continuing integration trend that has been going on for decades, culminating to the current "system-on-a-chip" methodologies, indicates that higher integration is an overall win.

In addition, when memory accesses can remain on-chip, significant energy savings arise[38]. Energy efficiency is a crucial consideration for portable battery-operated devices. The energy advantages associated with on-chip memory are examined in Chapter 7.

1.3 Embedded DRAM technology

Until recently, there has been little innovation in DRAM design. With DRAMs being sold as standardized commodity parts, competition between vendors has concentrated on the cost per bit. After the introduction of the basic 1-transistor – 1-capacitor memory cell, design efforts have focused mainly on the design of an increasingly compact memory cell. Functional innovations have been rare, because of the standardization that is necessary to make DRAMs interchangeable. The introduction of page-mode and synchronous DRAM are probably among the most notable.

However, demand for high density DRAMs has been declining. It appears that DRAM capacity grows faster than the demand, roughly quadrupling every 3 years. This trend is obvious from comparing a PC from the early eighties to one from the late nineties: while the PC motherboard used to be filled with DRAM chips, now it contains only one or two SIMMs, often with no more than 8 chips [97]. For the last few years there has been an oversupply of DRAM, that has driven prices down, forcing consolidations in the DRAM industry. Now appears to be the right time for innovation.

The golden opportunity could be the recent "system-on-a-chip" trend, integration of most system components on a single chip, including digital logic, analog parts and memory. DRAMs provided as "intellectual property" (IP) cores could spur new growth. Integrating all system memory on the same die as the processing units, can provide the functional units with low latency, high bandwidth access to the data; reduce the power dissipation from avoiding wide board-level interconnects; reduce parts count and system cost, size and weight; and allow flexible customization of the memory provided [97].

Aside from architectural considerations, the main challenge for integrating DRAM on a logic chip is that neither pure DRAM nor pure logic processes appear to be quite appropriate for the purpose. Logic processes provide fast logic and dense, multilayer interconnect, but lack the process steps needed to fabricate a small memory cell with good retention characteristics. On the other hand, DRAM processes have been optimized for very dense memory at low cost (ultimately cost per bit) and large data retention time, at the expense of transistor speed and interconnect density [56, 65].

Prototype systems from academia that integrate DRAM and logic circuits have usually chosen the logic process path [13, 54, 115], probably because DRAM processes have not been widely available outside of the DRAM industry. On the other hand, industrial prototypes have often been based on DRAM processes [27, 61, 86]. Recently, *embedded DRAM* processes have appeared, in an effort to bridge the gap. These processes incorporate features of both logic and DRAM processes in order to achieve dense, high performance logic at the same time as dense DRAM. To date, there is no process that offers logic on par with a pure logic process and memory of the same density and performance as a dedicated DRAM process. It is nevertheless possible to achieve good logic performance and reasonable DRAM density, at an increased cost per wafer. Characteristic of the state of the art in 1999 is the 0.18μm logic-based process from IBM with copper interconnect, a trench capacitor module, logic-only salicide and dual oxide thicknesses [26]. This process achieves performance identical to the corresponding logic-only process at the cost of a large memory cell, about 60-70% larger than the typical DRAM cell. The cost premium per wafer is undisclosed.

Another area of recent innovation is the speed of the DRAM itself. A major limitation of DRAM-based designs is the long latency of the DRAM core, as compared to SRAM. In conventional systems this latency has generally been hidden behind SRAM caches. However, as caches get increasingly expensive for today's large data sets and less effective for emerging applications, there is a renewed interest in high speed DRAM designs. This has resulted in some recent DRAM macros with random access latency several times smaller than off-the-shelf DRAM [40, 92, 124]. This high performance comes at the expense of lower density, but this may be an acceptable tradeoff in embedded applications. As seen in Table 1.2, performance that approaches that of SRAM with density not too far from that of conventional DRAM is possible.

Although embedded processes are more expensive than their pure logic or DRAM counterparts, the additional cost may be justified by their improved performance. Instead of competing on the cost-per-bit basis, embedded DRAMs may lead to product differentiation through added functionality or higher performance, in much the same way as in microprocessors and other logic chips. With the increasing popularity of handheld electronics, where high integration and low power consumption is paramount, it is likely that embedded DRAM processes will continue to evolve towards a more performance-driven direction.

	Latency		Density		Ref
Conventional 64Mb DRAM	29 ns	(1.00)	465.7 Kb/mm^2	(0.83)	[89]
	?		660.6 Kb/mm^2	(1.17)	[123]
Mean	29 ns		563.2 Kb/mm^2		
Fast 64Mb DRAM macros	6.8 ns	(0.72)	264 Kb/mm^2	(0.89)	[124]
	12.0 ns	(1.28)	330 Kb/mm^2	(1.11)	[92]
Mean	9.4 ns		297 Kb/mm^2		
2 - 4Mb SRAM	2.9 ns	(0.45)	35.5 Kb/mm^2	(0.79)	[96]
	9.0 ns	(1.38)	37.2 Kb/mm^2	(0.83)	[62]
	$\approx$ 7 ns	(1.08)	31.8 Kb/mm^2	(0.71)	[88]
	$\approx$ 7 ns	(1.08)	74.9 Kb/mm^2	(1.67)	[133]
Mean	6.5 ns		44.9 Kb/mm^2		

Table 1.2: Latency and density of 0.25μm memories. For DRAMs, the row latency is reported. Both latency and density are given as absolute numbers and as a ratio to the mean for the corresponding technology.

1.4 Related work

Recently, designers of FPGAs, as well as more conventional architectures, have investigated ways to take advantage of embedded DRAM technology. In conventional microprocessors, the most obvious path was to replace SRAM caches with larger ones implemented with DRAM technology, at the cost of a longer access time. Both experimental prototypes and commercial products have appeared. MoSys markets external cache chips based on DRAM technology, that are intended to replace conventional caches on PCs. Although they do not offer significant performance benefits, they reduce the cost considerably, because of the much smaller die size that they require. The MoSys cache chips hide the refresh operations behind burst accesses. To make this possible, they break down the memory into very small banks, so that they achieve a very fast row cycle. In a different approach, Lee and Katz[77] describe a DRAM cache design where the refresh operation is omitted completely. Instead, a timed cache line invalidation scheme is applied, marking as empty those lines that have not been accessed for more than the DRAM retention time.

In a somewhat more aggressive design, Mitsubishi has integrated a RISC CPU with 2 MBytes of internal DRAM as the main memory[113]. An additional narrow interface to external memory is provided. However, such techniques fail to take advantage of the high bandwidth provided by the internal memory, leading to only modest performance benefits [14, 64]. The IRAM project at Berkeley[105] is developing a high performance architecture based on a vector processor and 16 MBytes of on-chip DRAM. The motivation for using a vector processor is that it can utilize a very high memory bandwidth, while it is relatively insensitive to access latency. *Vector IRAM*, or V-IRAM[73], largely addresses the needs of next generation portable devices, where integration and low power consumption is very important, while many of the applications (like human interface, multimedia, communications) tend to vectorize fairly well.

The above approaches tend to be very coarse-grain. Typically, all the compute elements are clustered in one high performance processor core, coupled to a monolithic memory system. Whereas some of these approaches have demonstrated promising results, more fine grain architectures have the potential of higher benefits. FPGAs, with the naturally distributed computation, are a good candidate for such a distributed model.

The *Active Pages* architecture[93] integrates a small amount of FPGA logic in a memory chip, to assist the CPU with some of the more data parallel tasks. Active pages are designed to resemble a conventional memory page as much as possible, with the addition of functions to load and activate compute kernels that will execute in the reconfigurable logic. The architecture is mostly suitable for data parallel tasks, with very limited resources for inter-page communication. Reconfigurable logic is not instrumental to the architecture and the Active Pages group is now turning its attention to integration of conventional processors into the memory[52, 94].

Recently, NEC has designed an FPGA with integrated DRAM[86] in a very fine-grain architecture. The highlight of this design appears to be the very fast reconfiguration, that is achieved by pitch-matching the configuration memory of each logic block to the sense amplifiers of the neighboring memory bank. This makes it possible to achieve full reconfiguration in a mere 7 cycles. Unfortunately this design was not fabricated and this direction was not pursued any further. This design probably represents the highest degree of fine-grain integration possible with today's technology. As discussed in this thesis (Chapter 6) this may be over the optimum in terms of cost-performance. It is however an impressive data point with respect to the potential of fine-grain integration of logic and memory.

1.5 Overview of this work

The focus of this thesis is the evaluation of embedded DRAM technology as applied to reconfigurable computing.

Chapter 2 is a broad overview of the features of commodity and embedded DRAM designs, the various alternatives, and their merit with respect to their application to single-chip reconfigurable devices. As part of this work we designed and fabricated a test chip, called *Trumpet*, that implements some of the ideas discussed in this chapter. Trumpet combines a small reconfigurable array and a "configurable memory block", based on DRAM technology. Trumpet has been invaluable in giving us a good idea of the design issues involved and the implementation costs of various ideas. Chapter 3 describes the design of the Trumpet memory block in detail.

A survey of existing SRAM and DRAM designs provides data regarding the density advantage of DRAM over SRAM technology. A brief overview of these density trends and a case study of the Trumpet configurable memory block are presented in Chapter 4.

In Chapters 5 to 7 I focus on the effect of the high memory density that DRAM offers to the performance and energy efficiency of reconfigurable architectures. In Chapter 6 I describe a methodology for "sizing" a design consisting of a combination of reconfigurable logic and embedded DRAM blocks. Given a set of architectural parameters, the goal is to determine the values of these parameters such that the performance on a set of applications is maximized. Applying this methodology to a design similar to Trumpet yields insight into the scaling trends of such devices. In addition, with small modifications, this methodology can be applied to conventional designs with external memory. Doing so allows a direct performance comparison between single-chip devices based on embedded DRAM and the conventional multi-chip alternatives.

I find that embedded DRAM technology works best at the low end of the cost-performance spectrum. Small, low cost devices are found to perform about 3 times better on a small set of benchmarks than multi-chip systems of comparable cost. At the high end, there is a large range of speedups that depend on the characteristics of the application at hand. Some applications actually suffer some slowdown when executed on the embedded DRAM device, up to 2 times, while others achieve large speedups, up to 7.7 times. Still, even when no significant speedups are achieved, the energy efficiency and small form factor enabled by such devices will often make them an appealing technology.

The study of applications also reveals that large embedded DRAM blocks are not appropriate for everything. Devoting large memories, sized in the Mbit range, to tasks like small FIFOs and tables that only need Kbits of memory, is a waste of their capacity. Independent of technology, SRAM or DRAM, such large memories are not the right match for very fine-grain storage and retiming needs. At the same time, the longer column access latency of such DRAMs, when compared to Kbit-sized, distributed on-chip SRAMs, some times hurts performance. On the other hand, embedded DRAMs work great for regular, wide accesses, where very high bandwidth to a significant amount of data can be utilized productively. It appears that some combination of small and fast memories with larger, slower ones may be ideal for a wide class of applications.

Finally, the same application kernels suggest that the existing standard SDRAM interface, although invaluable for the commodity DRAM industry, may not be the most appropriate choice for applications where the DRAM is directly exposed to the user. We will see that alternative interfaces can be simpler and more effective at the same time.

Chapter 2

The embedded DRAM design space

The need to make commodity DRAMs interchangeable and the importance of maximizing density have led to very limited variations among the designs of different manufacturers. When it comes to embedded DRAM blocks, or *macros*, compatibility among vendors is much less of an issue. At the same time, memory density, although still important, is not necessarily the primary consideration. Because of these differences, a much larger range of features and ideas is acceptable for embedded designs. In this chapter we look at some of these features and how they apply to embedded DRAM designs. Where appropriate, we relate these to reconfigurable architectures in particular.

In brief, the question addressed in this chapter is "how *could* and how *should* embedded DRAM macros differ from commodity DRAMs" ? From the embedded DRAM designer's point of view, choosing the right set of features should be fairly straightforward when the application is fixed, as in an ASIC design. On the other hand, on programmable general purpose components there is usually a large range of applications, often with conflicting requirements. Balancing those requirements is not an easy task – usually, a wide range of benchmarks will be required to make informed decisions regarding the features to implement.

2.1 Bandwidth

For a large class of applications, memory bandwidth translates directly to application throughput. This is particularly true for multimedia, communications and human interface functions, that perform relatively simple computations on a large amount of data. Traditionally, microprocessor-based architectures have relied on SRAM caches to provide a high memory bandwidth to the processor. This approach relies on the assumption that sufficient locality exists, so accesses to the main memory, usually DRAM, are relatively rare. However, with the limited degree of temporal locality in new applications and the high level of instruction-level parallelism in modern processors, the ability of caches to intercept accesses to the main memory is limited. Therefore the bandwidth demands placed on the main memory increase[18]. With FPGAs in particular, high bandwidth is also desired in order to minimize the configuration time. In multiprogrammed systems, context switching time also depends directly on the available memory bandwidth.

The peak bandwidth delivered by a DRAM, or any memory, is simply equal to the *I/O bus width* times the *maximum access rate* (clock frequency for synchronous memories). However, the peak bandwidth can rarely be sustained. A number of factors, relating to the behavior of the application and the efficiency of the memory access protocol, will usually interfere, to necessitate a number of idle cycles on the memory bus. We can simply model the sustained bandwidth as the product of the peak bandwidth and a *protocol overhead* factor. This factor depends primarily on the address sequence, the mix of reads and writes, the various timing parameters of the memory – most importantly, the row and column latencies – the number of memory banks (or more generally the amount of on-chip SRAM, like sense amplifiers) and details of the memory command repertory and encoding.

Bus width

Wide buses are obviously much easier to implement on-chip, where no external interconnect is needed, than off-chip. Current commodity SDRAMs are available with interfaces up to 32 bits wide. In contrast to that, embedded DRAM macros are currently available with up to 512-bit interfaces[60].

Implementation of such wide interfaces requires a certain departure from the common DRAM design, where only a small number of bits is accessed from each memory array. As shown in Figure 2.1, in the conventional design a small number of data I/O lines, usually

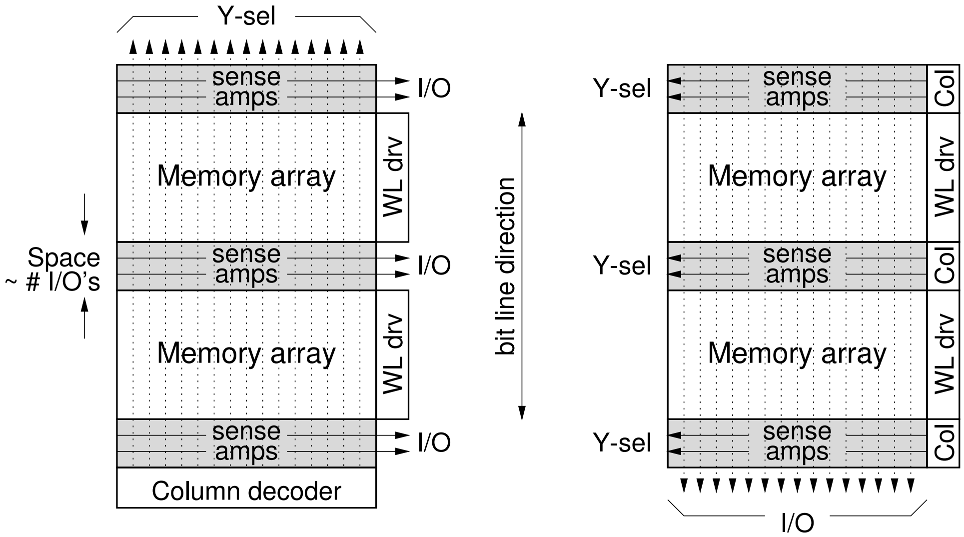

Figure 2.1: Memory core floorplans for conventional (left) and wide I/O (right)

1 to 4, run in a direction perpendicular to the bit lines. A large number of column select, or *Y-Select* lines, run parallel to the bit lines and connect the selected word to the I/O lines. A large number of I/O lines would result in a significant area overhead, because more space would be needed between memory arrays to route these I/O lines. However, the number of Y-Selects is inversely proportional to the number of I/O's, which allows the following trick when wide I/O is desired: just reverse the role of the vertical and horizontal wires and have the vertical ones carry the data and the horizontal ones carry the Y-Select bits[37]. For example, with 1024 sense amplifiers per memory array, an organization with 4 I/O's and 256 Y-selects could be easily rearranged for 256 I/O's and 4 Y-Selects. The direct relation between the number of I/O's and Y-Selects allows the two designs to be implemented in comparable area (trade off datapath area for Y decoder area) and column cycle time (trade off Y-Select capacitive load for I/O line load).

Wide I/O schemes also interfere with the common column redundancy techniques. In conventional DRAMs, when a bit line is marked as defective, the entire column (group of bitlines that correspond to one I/O word) is replaced with a spare column as a unit. This is quite easy to implement without significant overhead, by appropriate design of the column decoder. Obviously, if the column is very wide, a large number of spare bit lines will have to be available, just to replace a single defective bit line. Recent wide I/O designs have introduced different schemes. The common idea is that instead of substituting entire columns as a unit, individual bit lines are replaced within the addressed column. A number of variations exist, but either way some overhead in either the column access time[143], or area[91] is unavoidable.

It should be noted that a wide interface is not always an appropriate means to raise the bandwidth. Many applications with narrow word widths and little or no data parallelism will still prefer their "natural" word width. When the application width is much smaller than the memory interface width and the access pattern is not sequential, most of the bits in any memory access will be wasted – so the "effective" bandwidth (to data actually needed) will be much lower than the peak. In an ASIC environment, where the application is fixed, it should be easy for the designer to choose the memory width to match that of the application. On the other hand, with a general purpose programmable component, like an FPGA, the large application space may warrant a wide interface. Since FPGAs often perform well in applications with a high degree of data parallelism, a wide memory interface seems to be a natural match. For applications that require narrow memory accesses, appropriate interface logic should be available to multiplex the wide bus a narrower one. This logic adds complexity – and with that some area, power and delay. This additional cost is the price paid for the generality of programmable devices.

Access rate

Current commodity DRAMs are limited to 133MHz operating frequency, mostly because of the difficulty of driving conventional memory buses at high frequencies and the standardization across manufacturers of memories and related components. Free of system bus limitations and compatibility concerns, embedded DRAM macros are generally ahead of the curve: macros are currently available, built with comparable process technologies, that operate at up to 200 MHz[3, 60], while a 1 GHz experimental macro has been reported recently[121].

Some recent high bandwidth standalone memory interfaces, like DDR[25] and Rambus[106], try to overcome pin limitations by operating at a slow clock cycle and wide bus width internally, while the external interface runs at a multiple of the internal clock cycle and a fraction of the width. DDR DRAMs fetch 2N bits from the core in a full clock cycle, then output N bits at a time, over the two clock phases. Rambus takes this approach further: a Direct Rambus channel multiplexes a 128-bit internal bus accessed every 10ns down to a 400 MHz, 16-bit external bus, clocked at both edges of the clock. Both of the above approaches place stringent constraints on the design of the circuit boards.

Such approaches are more akin to wide I/O designs than high access rate ones. Independent accesses can only be issued at the lower internal rate, so the high bandwidth of the interface only benefits sequential bursts of data. Designs that support a high access rate to the memory core have a lot more flexibility, since they can access non-consecutive data words at a peak rate equal to the clock frequency.

Table 2.1 summarizes the current state of the art in external and embedded memory designs.

Memory type	width/device	transfer rate	peak bandwidth	Ref
SDRAM	32b	133 MHz	532 MB/sec	—
DDR SDRAM	32b	266 MHz	1064 MB/sec	—
Direct Rambus	18b	800 MHz	1600 MB/sec	[106]
Embedded	512b	200 MHz	12800 MB/sec	[60]

Table 2.1: High bandwidth DRAM statistics

Protocol overhead

In practice, the peak bandwidth can only be achieved in very few cases – most commonly, when the access pattern is sequential read-only or write-only. In less regular patterns, the sense amplifiers of each memory bank will effectively act as a cache. As long as accesses can be satisfied from this cache, the peak bandwidth can be sustained. Accesses that need to reach the DRAM core will expose the precharge and row access latency, cutting into the sustained bandwidth. Simply put, if W bits are transferred every clock cycle of T_c seconds, the peak bandwidth will be W/T_c bits/sec. With an effective miss rate of m and a miss penalty of N_{miss} cycles, the sustained bandwidth will be limited to

$$\frac{W}{T_c} \times \frac{1}{1+mN_{miss}} \text{ bits/sec}$$

For example, common 16-bit wide, 100MHz, 64Mbit SDRAMs offer a 200 MB/sec peak bandwidth. With a typical 6-cycle miss penalty, a random address stream that will miss on most accesses may experience a sustained bandwidth of only 28.6 MB/sec, or 14.3% of the peak in the worst case. Therefore, several attempts at providing a high sustained bandwidth for irregular access patterns have focused in reducing m or N_{miss}.

Banks and subbanks: The standard approach for commodity DRAMs relies on multiple banks, typically up to 4 in commercially available parts. All banks can be "open", or active, at the same time, with a row's worth of bits residing in the sense amplifiers. Essentially, the sense amplifiers are a direct-mapped, write-through cache, so more memory banks will help reduce the miss rate m. Making each bank wider has a similar effect, although the aspect ratio of the memory arrays eventually limits the width that can be attained.

Two banks is the minimum that allows a memory to achieve the peak bandwidth – and that, under suitable access patterns, most commonly sequential. Two memory banks allow read or write accesses to be performed on one bank, while the other is being precharged and activated. The advantage of the increased cache that results from more memory banks comes at the expense of a corresponding area overhead. For example, in the 2Mbit, 2-bank memory macro used in Trumpet, the sense amplifiers and related circuits take up about 13% of the memory core area. Four banks would require twice as many sense amplifiers, corresponding to another 13% area overhead with respect to the basic 2-bank design.

The number of sense amplifiers in a memory block deserves some more attention. In larger memory blocks, a large bank is typically split into smaller subarrays, with sense amplifiers between subarrays, as shown in Figure 2.2. This arrangement limits the length and resulting capacitive load of each bit line. To save area, each row of sense amplifiers is shared between the two subarrays above and below it. A switch, controlled by the row address, connects the sense amplifiers to the appropriate subarray. It is notable that although a large number of sense amplifiers exists, only a small number of them can be used at any one time. This restriction appears to be a large waste of SRAM bits, but treating each subarray as an independent memory bank would carry with it an area overhead for additional circuits like Y decoders, read/write datapath, etc. Yamauchi et.al.[144] estimate 40% and 80% overhead for 16 and 32 banks respectively, compared to a baseline 4-bank 256Mbit DRAM design.

To take advantage of as many sense amplifiers as possible with only a small area overhead, certain schemes have appeared that treat the individual subarrays as semi-independent subbanks[114, 144]. In both schemes, concurrent activation of neighboring subarrays is prohibited, because of the shared sense amplifiers, but distant subarrays (sub-banks) can be active at the same time. One of the schemes[144], with 32 subbanks in each of 4 banks, is estimated to perform up to 65% better than a conventional 4-bank design on a 4-node multiprocessor workload, with only a small area overhead.

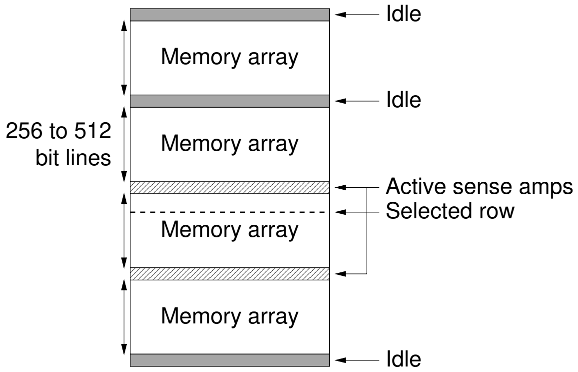

Figure 2.2: Typical memory bank floorplan

In embedded DRAM designs, where memory density may not be as critical as for commodity DRAMs, and since the sense amplifiers may be the only cache available, such techniques – either subbanks, or completely independent banks – may be appropriate. With small memory blocks in particular, the width of each bank (or memory page) may be limited by aspect ratio considerations, which makes multiple (sub)banks more desirable.

Other forms of on-chip cache: Enhanced SDRAM[120], attempts to improve on the basic bank design, by adding a row of SRAM to every bank, *in addition* to the sense amplifiers. This allows each bank to be precharged or refreshed while reads are satisfied from the SRAM row, in effect reducing the miss penalty, N_{miss}. Some times it is also possible to access two rows in the same bank at the same time: a read-only row stored in the SRAM bank and a write-only row still in the sense amplifiers, effectively doubling the amount of cache and reducing m. *Cached DRAM*[32] goes further, integrating a 16Kbit SRAM array with a 4Mbit or 16Mbit DRAM array, each with its own interface to the host, on the same chip. Finally, the *Virtual Channel Memory* architecture, or VCM, introduced by NEC[22], also integrates a 16Kbit SRAM on the same die as a 64Mbit SDRAM. This SRAM is organized as 16 independent *channels*, managed by the external system controller. The purpose of these channels is to decouple independent access streams to a computer's main memory (eg processor instruction and data accesses, I/O, graphics). There is often good locality within such streams, but not in the composite stream. By giving each of those streams its own SRAM buffer, interference should be minimized preserving each stream's locality, for an overall improvement in hit rate.

All the above techniques result in a somewhat improved sustained bandwidth, at

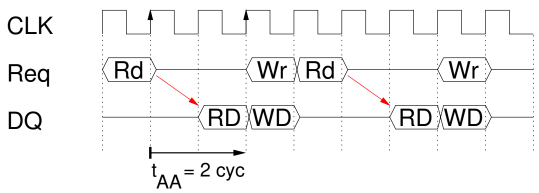

Figure 2.3: Underutilization of the interface during mixed read-write streams

the expense of a modest increase in die size. In commercial devices, where area or cost per bit is the primary concern, this area overhead allows only limited amounts of on-chip SRAM. For example, a 4Mbit Cached DRAM with 16Kbits of SRAM cache[32] has a 7% area overhead, while the overhead of a 64Mbit VCM DRAM[80], also with 16Kbits of channel SRAM, stands at 3%. In an embedded DRAM design, where area per bit is often less critical, it might be acceptable to use larger amounts of SRAM to further reduce the miss rate.

Handling mixed read-write streams: So far, the discussion has assumed that the access streams are read-only or write-only. The effective bandwidth is reduced even more when mixed reads and writes are issued. Because of the multicycle read latency of common synchronous DRAMs, idle cycles must be inserted between reads and writes. When a write follows a read, a minimum delay equal to the column read latency must be inserted, otherwise the read data and the write data will collide on the DQ bus. When a read follows a write, the requests can be issued back-to-back, but the data bus will remain idle for a duration equal to the read latency. As shown in Figure 2.3, a typical latency of 2 cycles, common in the current generation of DRAMs, reduces the effective bandwidth to 50% of the peak for alternating reads and writes, even when they all hit in the currently active bank(s). Synchronous SRAMs suffer from the same problem, although it is usually less pronounced, because of their shorter latency. A one-cycle SRAM read latency, will limit the read-write bandwidth to 66% of the peak.

To handle this situation, a number of delayed write schemes have been proposed recently, both for SRAMs[88] and DRAMs[114]. Though some variations exist, the common idea is that the write data are skewed with respect to the write command by an amount equal to the read latency. To avoid conflicts between read and write data in the memory core, the write data are temporarily placed in an on-chip write buffer. The final write into the memory array can be done when the memory is idle, when another write command is

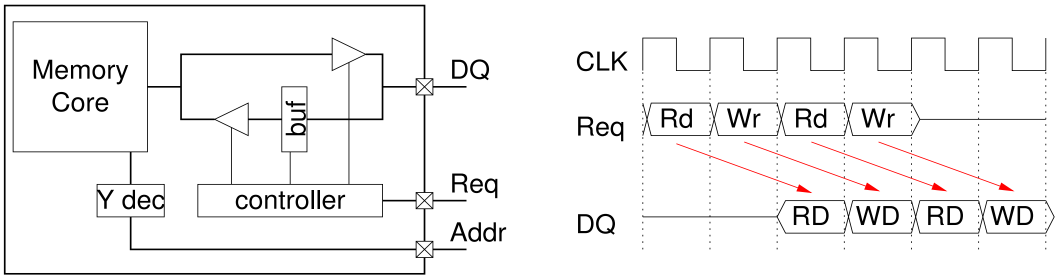

Figure 2.4: Simplified block diagram and timing of delayed write schemes

issued, or when a read is received that addresses the word in the write buffer. The key to this technique is that the write buffer is behind the interface registers and close enough to the memory core (or sense amplifiers) that writes can be completed within a cycle, without interfering with reads. In this way, as shown in Figure 2.4, the command and data buses can be utilized up to 100%, independent of the mix of reads and writes. The related overhead is quite small, mainly the write buffer and some additional logic. Although it has not been widely adopted by the DRAM industry yet[1], several commodity SRAM components relying on this technique are now available.

2.2 Memory core latency and cycle time

As seen in the previous section, unless the application access pattern is particularly well-behaved, the memory core latency will be exposed and cut into the effective memory bandwidth, ultimately limiting the application throughput. This latency manifests itself as N_{miss}, the miss penalty from the sense amplifier bank (or other type of on-chip SRAM) to the memory core. In addition, the data restore time for a memory bank usually necessitates a cycle time somewhat longer than just the core latency ($t_{RAS,min}$ specification). Consider for example an image histogram application, where an array of "buckets" is used to hold the running frequency counts. The access pattern to such an array is clearly data-dependent and random. One can expect that, unless the entire array fits in the sense amplifiers, a number of accesses will miss in the first level of the hierarchy. If the array is much larger than the sense amplifier capacity, most accesses will miss and processing will proceed at approximately the

[1]Notable exception is the Direct Rambus interface, which is somewhat similar in this respect.

memory cycle rate[2]. Therefore, to serve a broad application base efficiently, the DRAM memory core should be made as fast as possible.

So far, high speed DRAM circuit design has not been a priority of DRAM designers, because of the emphasis of the DRAM market in density. It appears that virtually all commercially available DRAMs have been optimized for the common PC/workstation environment, where accesses to the main memory are primarily cache refills or writebacks. Current commodity DRAMs and computer systems try to optimize the delay for a cache refill with higher bandwidth. The lack of interest in latency resulted in the performance of DRAM lagging that of microprocessors by an ever widening margin, known as the *processor-memory performance gap.*

Low latency research prototypes have appeared in the past, but have remained off the mainstream [68, 69, 87]. In embedded systems however, the only cache between the processing units and the memory core is usually the DRAM sense amplifiers. Because of the limited amount of equivalent cache, they are often ineffective at masking the core latency. Therefore, there is a renewed interest in high speed DRAMs.

The bulk of the row access latency consists of the time to drive the word line and sense the small signal on the bit lines. Reducing the size of the basic memory subarray in both dimensions can yield significant speed improvements, but at the cost of a considerable increase in the relative amount of peripheral circuits and corresponding reduction in density. This approach is taken by two recent high speed DRAM designs from NEC [92, 124], where the elementary memory subarray consists of only 8K or 16K cells, instead of the more common 64K or 128K.

In another approach, a larger cell capacitor provides a stronger initial signal, reducing the sense delay [68].

A number of circuit techniques have been used to speed up the peripheral circuits, with an unavoidable area overhead. Most high speed DRAM prototypes to date use some variant of a *direct sense* architecture [69, 87, 124]: a differential pair amplifier is used to sense the small signal on the bit lines and drive the output lines, while a separate regenerative amplifier drives the bit lines to full swing and restores the logic level in the memory cells. This approach isolates the large capacitive load of the bit lines from the read path. For one prototype [87] a 50% improvement in row latency (18ns) for a 14% area overhead is reported.

[2]Or, at best, at the inverse of the row access time, if a large number of banks are available and successive accesses hit different banks

Also, a *static word line driver*[124], requires more transistors than the conventional dynamic one, but needs smaller timing margins for reliable operation.

The effectiveness and cost of the above approaches are summarized in Table 2.2. The first three designs rely primarily on peripheral circuit optimizations (mainly in the sense amplifiers and word line drivers), while the last two also employ small subarrays. The design reported in [69] uses a BiCMOS process. For comparison, the mean effective memory density for the conventional designs listed in Table 4.1 (page 63) is 18.36 F^2 per bit, where F is the minimum feature size. Nagai reports a “conventional” latency of 35ns for the same process technology and memory capacity[87], while the 0.25μm 64Mbit DRAM in [89] has a latency of 29ns.

Although these fast DRAM designs are still not as fast as conventional SRAMs, they cover the middle ground of moderately high density and good performance. As summarized in Table 1.2 (page 10), such designs demonstrate latencies that are not too far from corresponding SRAM latencies, while their density is closer to that of commodity DRAM.

Process	Capacity	Latency	Improvement	Density	Ref
1.3 μm	1 Mb	23.0 ns	22%	35.1 F^2/bit	[69]
0.6 μm	4 Mb	17.0 ns	51%	34.9 F^2/bit	[87]
0.8 μm	4 Mb	14.0 ns	> 67%	31.3 F^2/bit	[68]
0.25 μm	64 Mb	12.0 ns	59% (cf [89])	47.3 F^2/bit	[92]
0.25 μm	64 Mb	6.8 ns	77% (cf [89])	59.2 F^2/bit	[124]

Table 2.2: High speed DRAM statistics. Density is expressed as F^2 per bit, where F is the minimum feature size.

2.3 Logical interface

SRAM and DRAM interfaces are currently standardized because commodity parts need to be interchangeable. This is not the case with embedded designs, since there is no need for multiple part suppliers. While designs might converge in the future, this may be a good time to reexamine the memory interface.

The first obvious approach is to expose the basic DRAM interface to the user: the common `RAS, CAS, WE` controls and the multiplexed address bus. This interface makes the memory block controller trivial, with corresponding benefits in the memory block area and possibly cycle time as well. However, the raw DRAM interface is often difficult to use, in particular when the access patterns to the memory are irregular and/or data dependent. In the general random access case the application designer has to implement the equivalent of a cache controller, a piece of logic responsible for maintaining tags, issuing row and column commands and halting the data flow when row accesses must be performed. This overhead can be significant, in particular in an FPGA environment, where logic resources are not as dense as in an ASIC.

Another option is to provide an "SRAM abstraction" to the user. The memory block controller, implemented in custom VLSI, could incorporate tag logic and row/column control, allowing the user to view the memory as a "pseudo-SRAM". In that case, simple read request, write request and non-multiplexed address bus can be used. Still, this approach cannot eliminate the variable DRAM latency, so some form of feedback is needed from the memory controller back to the user logic. In the *Trumpet* chip, we took the approach of hiding this variability from the user by a stall mechanism. This will be discussed in detail in Chapter 3. An SRAM-like interface makes design of applications easier, but at the same time it hides optimization opportunities from the user, as demonstrated in the following example.

Example 1: Basic image processing

In many basic image processing applications we have a purely sequential access pattern into both the input and output images. For this example assume that the datapath resources are enough to process one pixel per cycle. With 32 bits per pixel (R,G,B, padded up to 32 bits) and 1Kbit wide DRAM pages, we can fit 32 successive pixels per page. The SRAM abstraction will cause one stall per 32 pixels, or cycles, because the next row of DRAM must be fetched and there is no means to do a prefetch. In fact, if the stalls at the input and output buffers cannot overlap, we may actually have 2 stalls per 32 cycles. A typical stall cost of 6 cycles leads in an effective throughput of 84% of peak in the best case (overlapping

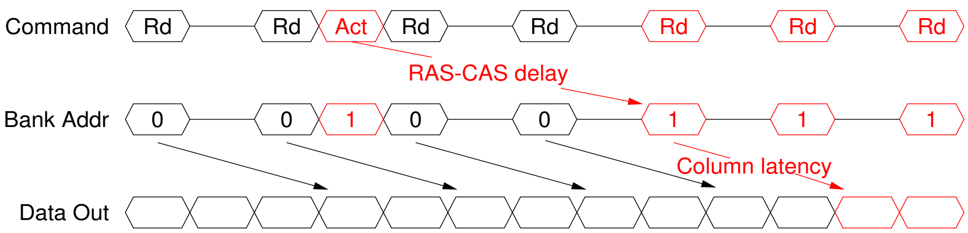

Figure 2.5: Bank ping-pong

stalls), or 73% in the worst (no overlaps).

The plain DRAM interface works better in this case: assume that we have 2 or more banks and the address bits are arranged so that successive groups of 32 pixels map to alternating banks. Also assume for now that a burst read capability is provided by the DRAM controller. This requirement will be revisited later. Two-element long burst reads allow a continuous read stream with a read command issued every other cycle. It is possible to issue a bank activate command during one of the idle cycles, prefetching the next row and avoiding stalls completely. This approach allows us to sustain 100% of the peak throughput.

The above example was very simple. It is not very difficult for the programmer to design the application control logic so that reads and bank activations are interleaved in the right order. There is a significant payoff in performance for a very modest increase in the design effort. However, in the more general case of random accesses, the additional design effort needed by the programmer may be significant, as demonstrated next.

Example 2: Entropy coding

This example examines the accesses to the Huffman table during the entropy coding step of a JPEG compression application [134]. An input stream of quantized DCT coefficients is read and zero-length encoded. Once a nonzero coefficient is encountered, the last zero-length and the current nonzero value are concatenated and used as an index to a Huffman table. The values looked up in the table are output as a sequential stream.

The accesses to the Huffman table are random and data dependent. The Huffman table can be stored compactly in 3 Kbits, which corresponds to 3 DRAM pages for a 1 Kbit wide organization. With a 2-bank memory block, it is not possible to hold the entire table in the sense amplifier rows. In cases like this, an SRAM-like interface can simplify the application logic design.

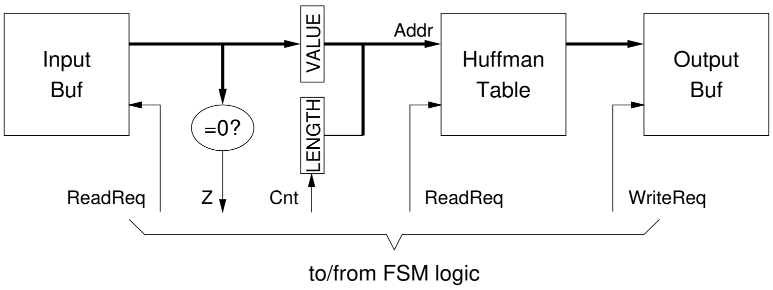

Figure 2.6: Simplified entropy coding datapath

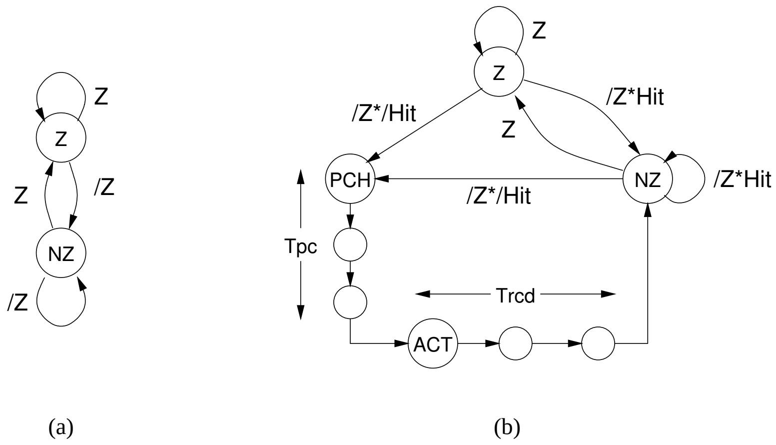

Figure 2.7: Entropy coding FSM logic for the SRAM (a) and DRAM (b) interfaces

If the controller of the memory block provides the logic to decompose a read request to a row access and a column access when necessary, the application control logic is trivial – just 2 states, `Zero` and `Nonzero`, corresponding to the last input. A read request is issued to the Huffman table on every occurrence of the nonzero state and a write request is issued to the output a fixed number of cycles after the Huffman table read. Read requests are issued continuously to the input. A controller like this can be implemented in only 2 or 3 logic blocks in most FPGA architectures. When an access to the Huffman table cannot be satisfied from the data currently loaded in the sense amplifiers, the memory block controller will issue a `stall` signal that will freeze all the logic, until the internal row access is performed. This process can be made completely transparent to the user logic.

With a raw DRAM interface exposed to the logic, the programmer has to take charge of maintaining the tags, decomposing read requests to row and column accesses when necessary and pausing the data flow whenever a row access is performed. The state machine implementing the user control logic will have to include wait states that mirror the internal delays of the DRAM (precharge latency and RAS-CAS delay in this case) and multiplex row and column addresses onto the DRAM address inputs. A possible implementation of such a controller, 8 states altogether, is shown in Figure 2.7, where it is compared to the controller for the SRAM interface case.

It is estimated that the additional logic needed to handle misses with a DRAM interface can be implemented in about 50 logic blocks in an FPGA similar to Trumpet. But more important may be the additional design effort and understanding of the DRAM specifics required of the programmer. Note that explicit handling of misses is required for *correctness* of the results, without any performance improvement possible over the pseudo-SRAM case.

The above examples demonstrate that providing an SRAM abstraction to the programmer is a mixed blessing. On one hand, it makes handling irregular reference streams a lot simpler, which relieves the programmer of the design effort and frees logic resources. In effect, the logic to handle row accesses is moved from the user space to the hardwired memory block controller, where it can be much more efficient. On the other hand, the SRAM interface hides optimization opportunities from the user. In particular, it provides no prefetching semantics, so sustaining the peak bandwidth is not possible. In addition, if the overhead to handle the general random access case is fixed in the CMB controller, it will be paid for even when it is not needed. This overhead can consist of somewhat increased CMB area, as well as possible reduction in the cycle time, because of the increased logic complexity.

For a fixed-function device, typically an ASIC, the interface choice can be driven by the application. For a programmable device, for example a microprocessor or an FPGA, it may be desirable to have both modes supported. This flexibility allows the optimizations when the access patterns are regular and at the same time relieves the programmer from the additional effort when access patterns cannot be predicted. The down side of course is that the complexity of the memory controller is increased, since two access modes have to be supported.

Dependence on DRAM timing parameters

A problem with both the SRAM and DRAM interfaces is that the DRAM latency is exposed to the programmer. This problem is more severe with the DRAM interface, where not only the column latency, but other timing parameters, like the precharge time or the row latency are also exposed. This is not an issue for most one-time designs, but for a programmable architecture that is intended to span across multiple generations, it can be a major impediment towards object code compatibility. Details such as memory latency often change from one implementation of the architecture to another. Exposing such implementation-dependent features to the programmer has been proven in the past to be a bad idea, since it prevents a single binary from running on more than one implementation of the architecture. For example, the initial MIPS processor design did not interlock on read after write data dependencies, relying instead on the compiler to schedule instructions accordingly [50]. This architecture made it impossible to execute binaries

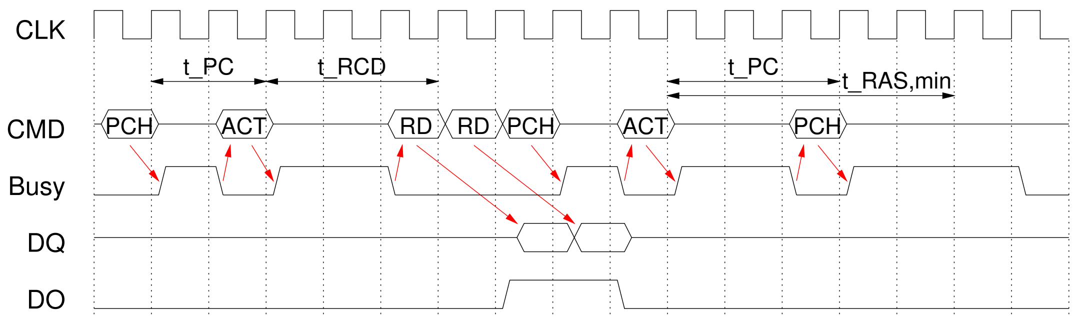

Figure 2.8: Bank busy protocol

compiled for one implementation on another, and was very soon abandoned for a fully interlocked implementation[3]. Similar problems existed for the Intel 80860 processor [79] with its "explicit" floating-point pipeline, a model that was also abandoned in favor of a fully interlocked pipeline in the 80960 [82].

A possible solution to the above problem could be to provide some kind of handshake between the memory controller and the user logic. At a minimum, a `data out strobe` should signal availability of read data to the logic. User logic could be built around this strobe to mask the column access latency. If the memory controller implements an SRAM abstraction, this will be sufficient. If the raw DRAM interface is used, the user logic is responsible for issuing precharge, bank activate and read/write commands at the right timing. In that case, additional feedback is needed from the memory controller, to convey bank availability information, so the timing parameters will not have to be hardcoded in the logic. Per bank `busy` signals should be provided in that case. Implementation details of such a protocol are somewhat involved and can vary, but one possible implementation is shown in Figure 2.8. Note how the `Busy` signal is raised after each `ACT` and `PCH` command.

Example 3: Entropy coding revisited

As seen in Figure 2.9, with the addition of data out strobe and bank busy signals to the memory controller the user FSM for the entropy coding example is simplified, since there is no need for multiple wait states. More importantly, the design can now be ported to implementations with different latencies.

Of course, most current reconfigurable devices expose several other implementation

[3] Interestingly the name, `Microprocessor without Interlocked Pipeline Stages`, survived.

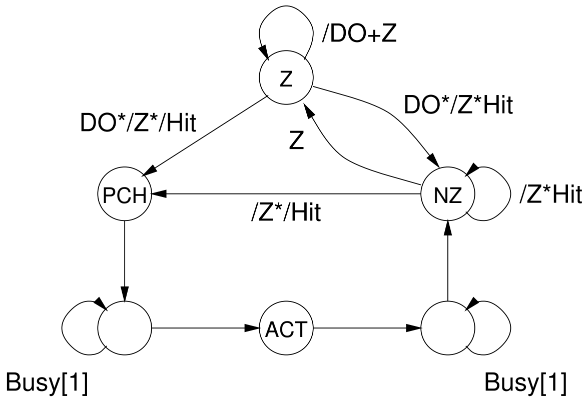

Figure 2.9: Entropy coding FSM after the addition of handshake signals

dependencies to the user. Not the least, configuration bitstreams are not portable between implementations. However, efforts are under way to define *application binary interfaces* (ABIs), that will survive across generations and make binaries portable [104, 111]. A well-defined implementation-independent memory interface should go along with such ABIs.

Alternative interfaces

As discussed earlier, the basic SDRAM interface must support bursts, to allow spare cycles for prefetch (bank activate) commands and thus attain the peak bandwidth when the access pattern allows it. The *mode register*, as defined by the SDRAM interface standard, can be programmed to set the DRAM into burst mode. Handling the mode register can be very cumbersome when different epochs of computation have different requirements, especially in ASIC or FPGA environments, where adding states in the user's state machines is more difficult than adding a few instructions in a piece of code. Furthermore, when the access pattern is not compatible with the burst sequence, single word accesses have to be used instead, leaving no space for prefetches. Such cases are for example all strided accesses, the zig-zag scan pattern of the JPEG compression, or some implementations of the DCT transform.

It is possible to make the controller simpler and at the same time more effective, by providing separate *row* and *column* command lines. In that case, a read or write and the activation of the next bank can be issued at the same time, without the need to use burst transfers. This approach leads to a significant reduction in the complexity of the memory block controller: all burst logic, as well as the mode register, can be removed. User logic is also simplified, since it does not have to handle the mode register. In addition, this allows background prefetches to be performed even when accesses are not sequential.

Row commands can be encoded with an `Activate` and a `Precharge` control line, accompanied by a `RowAddress` and a `RowBankAddress`. For the column commands, a `Read` and `Write` control line and a corresponding pair of `ColAddress` and `ColBankAddress` can be used. The above set of controls represents a simple, yet effective interface, that allows the programmer to take maximal advantage of the independence of memory banks. In a more elaborate implementation of the same idea, the latest (1999) incarnation of the Rambus memory interface provides for separate row and column buses [58]. It is notable that although the basic transfer unit is a 16-byte packet[4], no multi-packet bursts are implemented for longer transfers.

Another possibility is a *stream* read-only or write-only mode, that provides an internal address counter and prefetch logic, relieving the user from these tasks. Of course, this mode is advantageous only in environments where user-level logic is much more expensive than logic in the memory macro – for example an FPGA with embedded memory banks. In addition, sequential accesses must be the prevalent access pattern, to make this capability worthwhile. These conditions are met by the SCORE architecture [104].

2.4 Refresh logic

In an ASIC or FPGA environment, where the user logic frequently accesses the DRAM directly, refresh is somewhat more difficult to handle than in conventional systems. In a microprocessor-based design the DRAM is typically accessed only on cache misses. Then the DRAM can be refreshed in the background, while the CPU is accessing the cache. If a refresh cycle and a cache miss coincide, the refresh latency may appear as an occasional increase in the miss penalty. But when the compute logic accesses the memory directly, refresh cycles and regular accesses will collide more often, so the matter of arbitration deserves some more attention.

Because refresh cycles are rare compared to memory accesses (one every few thousand cycles as opposed to one per cycle or so), a reasonable policy is to give priority to refresh over logic accesses – otherwise, a continuous stream of memory accesses from the logic could lead the refresh process to starvation. The problem with this approach is that

[4] Both data and control (row and column) packets occupy the bus for 4 cycles

support must be provided for the operation of the logic to be preempted. Furthermore, the refresh cycles cut into the application throughput, since the memory is unavailable during those times. In some designs (for example *Trumpet*), a handshake mechanism is implemented between the memory and the logic for other reasons: frequently, to hide the variable latency of the DRAM from the user logic, or to make the application logic insensitive to the actual timing parameters of the memory, as discussed in Section 2.3. In those cases it is not very difficult to incorporate the arbitration mechanism in this handshake. In other cases, where the logic is custom designed around the specific timing parameters of the DRAM and it expects to know “by design” when data are returned, a handshake mechanism may have to be designed specifically for the needs of refresh.

In certain cases, it may be possible (and actually beneficial to performance) to deactivate the refresh mechanism, or not to implement one at all. When a memory macro is being designed for a specific application with known access patterns, it may be possible to take advantage of those patterns. For example, in many cases the lifetime of data in the memory is known to be shorter than the retention time, which generally falls in the millions of cycles. In many network switch designs, if data packets are buffered for too long, it is a sign of congestion in the network and packets have to be dropped. In a DRAM-based cache design[77] a timed invalidation scheme has been designed instead of a refresh mechanism: lines that have not been touched for a certain amount of time are marked as invalid. In other cases, applications sweep through their data set in a short time, making refresh unnecessary. Finally, in some cases with simple, regular patterns, it may be possible for the logic controller to execute refresh cycles in otherwise idle time slots.

MoSys uses an interesting hidden refresh scheme in its 1T-SRAM design[40]: A macro consists of several small DRAM banks. In addition, a small SRAM cache equal in size to one DRAM bank is provided, but this is transparent to the user logic. Every access to the DRAM will also transfer the addressed row to the SRAM cache. Each DRAM bank can be refreshed while the logic is idle or accessing other banks. When the access pattern hits one bank continuously, the SRAM cache is used to prevent starvation of the refresh process: at least one access in every refresh period is guaranteed to hit in the cache. Then an idle cycle is available at the “hot” bank, which provides the needed opportunity for refresh. In this way, user logic is completely relieved from the refresh responsibility and for this reason the design is marketed as “1-transistor SRAM”.

2.5 Soft error immunity

Cosmic rays (protons, neutrons, alpha particles, etc) can upset the contents of dynamic memory cells, by injecting minority carriers into the storage node as they pass through the memory array. This effect is the fundamental limitation in the storage capacitor scaling trend, since the *soft error rate* (SER) depends directly on the stored charge. For 16Mbit DRAMs (typically in a 0.30 to 0.35 μm technology and cell capacitance around 35fF) Ziegler et.al.[148] have observed experimentally a SER roughly on the order of 1 soft error per 32 MBytes per year[5]. With the current scaling trend of DRAM cells, this does not change considerably across technology generations. However, with rapidly increasing amount of memory on computer systems, soft error rates are becoming more of a problem. For example, a server actively using 1 GByte of memory would expect to see a SER on the order of 30 errors a year, or an error every few days.

Soft errors are generally handled adequately in conventional computer systems with *parity*. A “9th bit” is provided with every byte of data to encode parity information and in the case of a parity mismatch the operating system is notified. Soft errors pose a serious problem for certain FPGA architectures however, when DRAM is used to hold configuration information. In many architectures multiple tri-state drivers are used to connect the logic blocks to interconnect lines. Normally, at most one driver will be enabled at a time, usually under configuration control. If configuration bits are corrupted, it is possible that more than one of those drivers will be enabled at the same time, likely causing a large current to flow across the fighting drivers. If the configuration memory consists of SRAM cells and the configuration bitstream is loaded through a parity-protected memory system, the FPGA is generally safe. Otherwise, the potential of catastrophic failure exists.

With current technology, it appears that the frequency of such failures is quite low. With configuration bitstreams rarely exceeding a few Mbits, even in the largest systems, one may expect soft error rates affecting the configuration memory of one per several years of continuous operation. Of those errors, most are likely to be harmless – for example, changes in the LUT contents, connections broken, or spurious connections to otherwise unused wires. So, the expected time to failure seems to exceed by far the useful life of the device. On the other hand, for mission-critical applications, as in avionics, military or medical electronics, that might not be acceptable. Furthermore, if in the future the capacity of FPGAs scales to

[5] The central point on their research being that this varies significantly by memory cell type.

several MBytes of configuration memory, the above assumption might not hold any more.

A number of ways to handle soft errors exists: at the level of FPGA circuit design, architectures that rely on multiplexers rather than tri-state drivers are safer. When such an architecture is not desirable, it may be necessary to implement some hardware support in the memory system. The primary choice in this case is between error detection and correction. Parity can be used with relative efficiency to detect single errors. Alternatively, a number of error correction codes, most commonly a Hamming code, can be used to actively correct one error and detect more. However, since a soft error in the configuration memory appears to be a rather rare event, the additional overhead may not be worthwhile. For example, Kalter et.al[66] report an 11% area overhead and 5ns out of a 16ns column access latency for an ECC DRAM design. Parity overhead on the other hand can be made quite small, depending on a likelihood of double errors[6]. For example, a 1 bit in 32 parity scheme corresponds to under 4% area overhead.

A simple protocol for a "safe" configuration load based on parity would require parity match circuits in the configuration path of the FPGA, responding with an `Ack` signal to the agent that controls the configuration loading (for example a host processor). In the case of a negative `Ack` the current transfer should be aborted and the configuration should be reloaded from safer storage, such as disk or EPROM. When a DRAM macro is specifically designed for use in an FPGA environment, one should consider inclusion of an extra bit for parity, while it may also be practical to integrate the parity check circuits in the memory datapath.

2.6 Sub-word write enable

When a narrow logical interface needs to be implemented on top of a wide hard-wired memory interface, fine-grain control of memory writes is necessary. For example, in a 64-bit wide memory interface, the memory macro designer might want to export byte-wide write enable controls to the user logic. This is particularly important in programmable systems, where different applications will have different requirements. In the case of an ASIC, where the memory macro will support a single application, omitting the write enables and implementing a memory interface of the desired width directly will lead to a more

[6]Even in the event of a multiple error, about one out of twenty events observed in [148], the likelihood of more than one error in the same word can be minimized by appropriate interleaving of the bit lines.

streamlined design.

For certain applications, such as graphics, even more fine-grain control may be necessary – for example, the write-per-bit function, common in SGRAMs. Again, this is a special feature that one should consider on graphics-heavy applications.

2.7 Dual port capabilities

Although not a common DRAM feature, some dual port embedded DRAM designs have appeared. In [145] the sense amplifier bank is dual-ported, with one read and one write port. This organization does not allow arbitrary memory locations to be accessed concurrently, but supports the read-modify-write operation common in 3D graphics. The same mode could be used to implement FIFOs very efficiently.

2.8 Interconnect

For most applications the physical interface of a memory macro will simply consist of a set of data, address and control terminals. However, special considerations are needed in the case of FPGAs. In this case, all interconnect is programmable and it should be possible for the data, address and control signals to be mapped to a number of physical wires on the configurable interconnect network. Depending on the FPGA architecture, an all-to-all mapping or a subset of that may be appropriate. While not strictly part of the memory macro, it is often convenient from a high-level design point of view to include this interconnect structure as part of the memory macro.

With a plethora of existing configurable interconnect structures, it is difficult to make any general statements regarding the design of such an interconnect block. Wilton[137, 138] has studied the interconnect requirements for the case of a 2-dimensional mesh with a custom-designed switchbox and proposed a depopulated crossbar structure for the memory interconnect block. In Chapter 3 of this thesis, the interconnect requirements and resulting network design for the configurable memory block of *Trumpet* are discussed.

An additional function of this interconnect block may be the "folding" of a wide hardwired interface down to a limited set of network wires, in order to implement a narrower interface, based on the requirements of the executing application. For a narrow read, the controller of the memory block must decode the appropriate address bits provided by the application and select the appropriate subset of the raw memory word to be forwarded to the FPGA network. For a narrow write, based again on the same address bits, the controller should drive the appropriate write mask bits of the memory, in order to limit the write operation to the addressed bits.

It should be pointed out that in an FPGA-based design, both the I/O width and the mapping to the network will be customized under control of appropriate configuration bits. This results in the memory block being *configurable*, with its own configuration bit stream.

2.9 Test support

Embedded memory blocks (of any technology) pose additional testing problems with respect to their standalone counterparts: because their interface is usually not accessible from the chip pins, their controllability and observability are limited – in fact, they depend on the design of the piece of logic that uses the memory block.

It is easy to reclaim the needed controllability and observability by implementing an external test interface, multiplexed with the regular logic connections onto the memory block interface. This interface may be narrower than the regular memory interface, in order to save chip pins. Still, this approach raises the pin count and packaging cost of the device.

A better option may be implementation of *Built-In Self Test* (BIST)[131], especially when a scan path architecture[2] is used for testing the rest of the chip: a small test engine could be inserted in the scan chain, run all the necessary memory tests and output the fuse repair information again via the scan chain. Because of the algorithmic nature of most memory test sequences, memory test engines can be implemented with relatively low complexity, almost independent of the memory capacity. The overhead of such approaches is reduced as process technologies advance and memories get larger. With small memory macros however, this overhead can be significant, unless the test engine can be shared between multiple macros on a chip.

Programmable architectures provide the opportunity of “free” memory BIST: just implement any test algorithm on the programmable device. In that case of course, it will be necessary to test the logic before the memory tests.

Chapter 3

Point design: Trumpet

The architecture studied by the BRASS project is outlined in Figure 3.1. It combines a conventional microprocessor with a reconfigurable computing array. This array consists of a number of *compute pages* and an equal number of *memory pages*, interconnected with a novel interconnection network. All three elements of the array – compute page, memory page, and network – are reconfigurable, under the control of the microprocessor.

This architecture has been designed to support a novel computation model, called *SCORE* (Stream Computation Organized for Reconfigurable Execution)[21]. The objective of the SCORE model is to allow software to exploit a wide range of hardware resources, by providing binary compatibility among reconfigurable devices of the same family. In this way, as new and larger, or smaller and more economical devices become available, the same software can run on them without modification, or even recompilation. To achieve this goal,

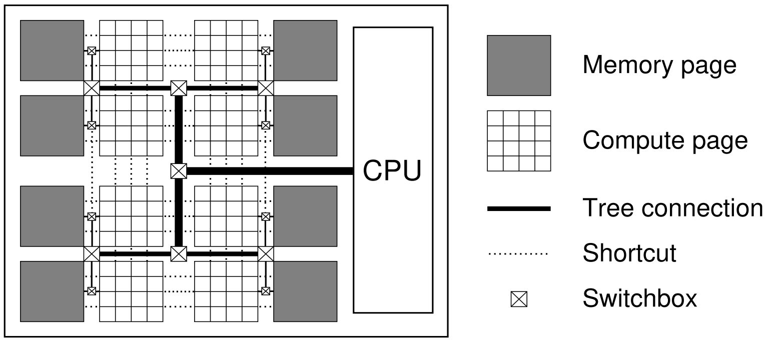

Figure 3.1: BRASS architecture diagram

the SCORE programming system decomposes each computation into page-sized operators; then, individual operators are scheduled onto the array by a scheduler process running on the processor. With the page size and binary configuration format being invariant between instances of the architecture, the number of pages can be scaled transparently to the executing application.

Operators communicate with *streams*. When the producer and the consumer of a stream are both resident in the array, a stream is just a direct connection over the network. When only the producer is scheduled, its output is redirected into a *stream buffer*, implemented as one of the on-chip memories. When the consumer is resident and some buffered input exists, it is read out of the stream buffer. When an operator runs out of output buffer space or out of input data, it may be swapped out and replaced by another operator. Therefore, in general larger buffers contribute to longer computation epochs between reconfigurations. When an operator is swapped in, the corresponding configuration and some initial (or intermediate) state can be loaded either from the shared external memory, or from one of the on-chip memory pages. Configuration from the on-chip memory is much faster, so it is desirable for the memory pages to have enough capacity to hold several configuration and state bitstreams. Finally, on-chip memory pages can be explicitly used as RAMs or ROMs by the application logic, providing high bandwidth and relatively low latency access to user data.

With the above three uses of memory, stream buffering, configuration and state storage and user data storage, heavy demands are placed on the memory system. At a minimum, the architecture should support a number of independent memory banks, each with its own address port and control, to allow for multiple streams, user data accesses and possibly reconfiguration events to overlap. The total memory capacity should be large enough to hold a number of configuration and state bitstreams, significant stream buffer space and a certain amount of user data. On-chip SRAM simply cannot provide enough capacity for all the above uses in reasonable area. External bulk memory will necessarily suffer from a small number of independent interfaces – realistically just one or two, subject to pin limitations. In addition, wide external buses are much more expensive than internal ones. Embedded DRAM is therefore a key technology, enabling this type of architecture.

In our architecture, compute and memory pages are arranged in pairs. Although in principle it is possible to store the configuration for a compute page in any memory page, reconfiguration is most efficient when it occurs between the pages of a pair: the interconnect is minimized, reducing wire pressure and energy dissipation. With this policy it is practical to use a dedicated bus to carry the configuration bits from the memory page to the compute page. Then, it is possible for one compute page to be reconfiguring from its neighbor, while at the same time other pages are executing. It is also possible to reconfigure all compute pages from their neighboring memories at the same time, taking advantage of the combined bandwidth of all the distributed memory interfaces. Total chip reconfiguration is then possible in the same amount of time it takes to reconfigure just one page. In contrast, for user data access and stream buffering, it is possible to establish links over the network between any pair of pages. When the SCORE run time system is fully implemented, the switchbox configuration for these links will be synthesized on the fly as pages are scheduled in and out of the array, and they will be static between scheduling events.

The *Trumpet* test chip is a first implementation of a small instance of the reconfigurable array: a single compute page, coupled to a single memory page[1]. It is laid out in a 10 mm $\times$ 5 mm chip, with a 0.4μm design rule for the logic and a 0.25μm rule for the DRAM. It has been designed with an operating frequency target of 250 MHz. The DRAM has been designed by LG Semicon (now Huyndai Microelectronics) and provided to us in the form of a hard macro. The rest of the design was carried out at Berkeley and consisted mainly of two parts: the compute page design, and the interface between the DRAM macro and the compute page, described in detail in Section 3.3.

Trumpet's interface to the external world consists mainly of a relatively low speed 32-bit wide bus and a pair of fast unidirectional 8-bit buses, one for input and one for output. The low speed bus runs at an eighth of the internal clock frequency, or 31.25 MHz, and is designed so that an external host (for example a PC) can easily interface to it over a standard I/O bus, such as PCI. The high speed I/O's run at the internal clock frequency of 250 MHz and are primarily intended for custom interconnect to devices such as video DACs etc. In addition, a number of control and status signals are used for the communication between the host and the compute and memory pages. The functions of these signals will be described in the appropriate sections on the compute and memory page. Finally, a "back door" interface to the DRAM has been implemented by the LGS engineers, to allow for test and repair of the DRAM independently from our logic.

There were several motivating factors behind the design and implementation of

[1]The network is an integral part of the compute page, as will be made clear in Section 3.1.

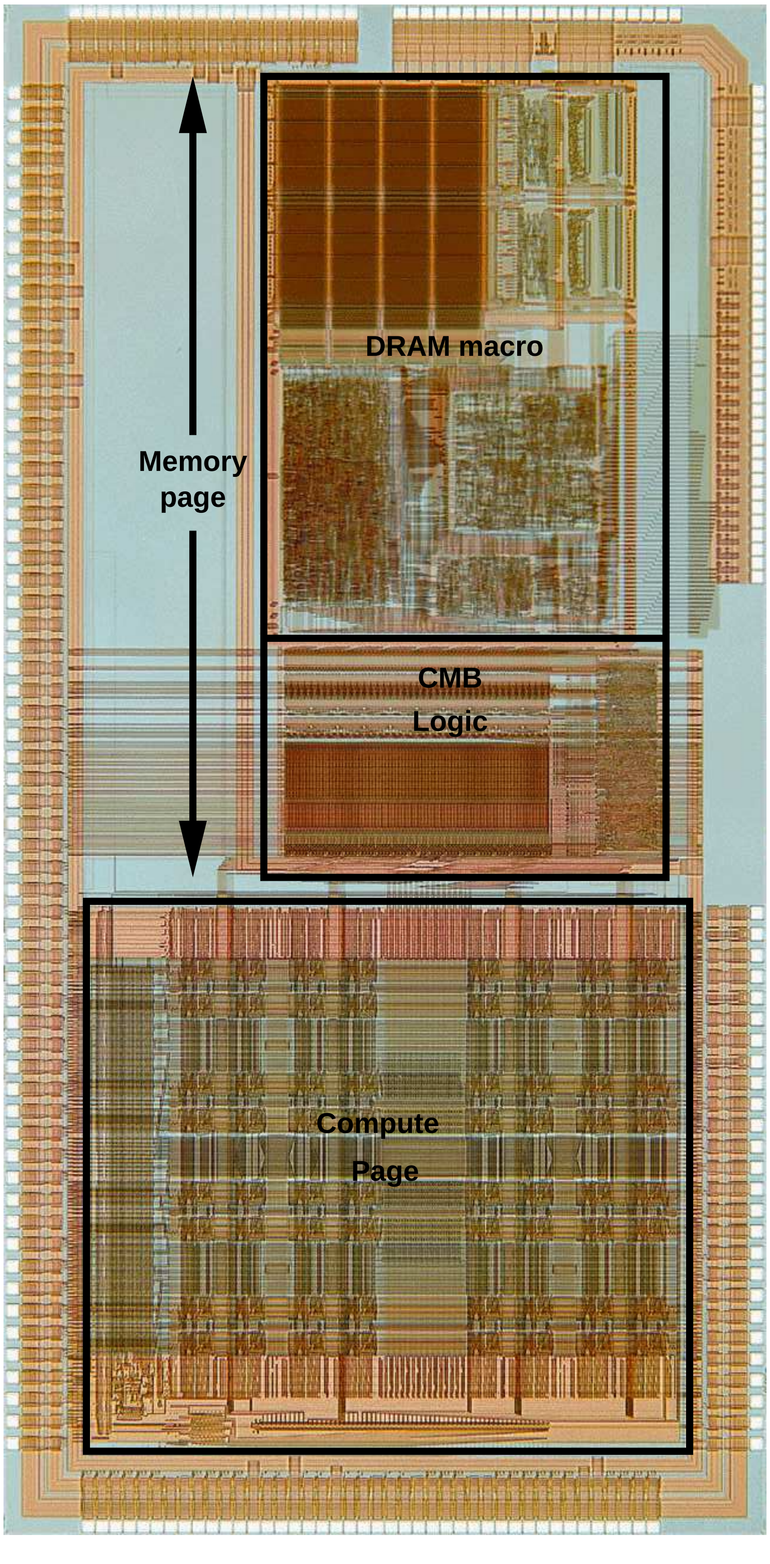

Figure 3.2: Annotated die photograph of Trumpet

this test chip: first, to verify the integrity of the architecture of the compute and memory pages; second, to give us realistic estimates of the costs associated with the various design options, in terms of area, complexity, cycle time and power; finally, the logistics of working with a DRAM vendor, integrating an existing "IP core" (the actual DRAM) into our design, matching our tool flows and other such issues had to be sorted out, before we embarked on a larger scale project.

The test chip was fabricated on LG Semicon's 0.25μm "embedded DRAM" process during 1999. Unfortunately, due to fabrication problems, the DRAM was not functional. This still allowed testing of the compute page, which does not directly depend on functional DRAM, but testing of the memory page was for the most part not possible. Results presented in this chapter are based on simulation.

In the following I will briefly introduce the design of the compute page and the DRAM macro and then focus on the design and evaluation of the memory page. A more concise description of the memory page design can be found in [98]. The discussion in this chapter does not cover implementation details in very much depth – instead, a more detailed description of the memory page implementation is given in Appendix A.

3.1 The compute page

The compute page is a lookup table-based reconfigurable array, consisting of 64 *Basic Logic Blocks*, or BLBs. Each BLB contains two 4-input lookup tables (4-LUTs), with shared inputs, as shown in Figure 3.3. A fifth input is provided to select between the outputs of the two LUTs, linking them in this way into a 5-LUT. A configuration bit enables either the two separate 4-LUT outputs, or the single 5-LUT output. This fifth input is hardwired to the 5-LUT output of a neighboring BLB. In this way, a fast path, termed the *cascade*, is formed between BLBs, that bypasses the general interconnect network. This path can be used to efficiently implement carry chains and wide functions.

BLBs are interconnected via a reconfigurable *fat pyramid* network[44]. A fat pyramid is based on a fat tree, a binary tree with the capacity of each branch increasing as we go from the leaves to the root. BLBs occupy the leaves of the tree, while the root can be used for I/O; this would provide the connection to the CPU in the architecture depicted in Figure 3.1. The tree topology was chosen to support fast placement. Instead of the common simulated annealing, on a tree one can perform placement by partitioning[46], a

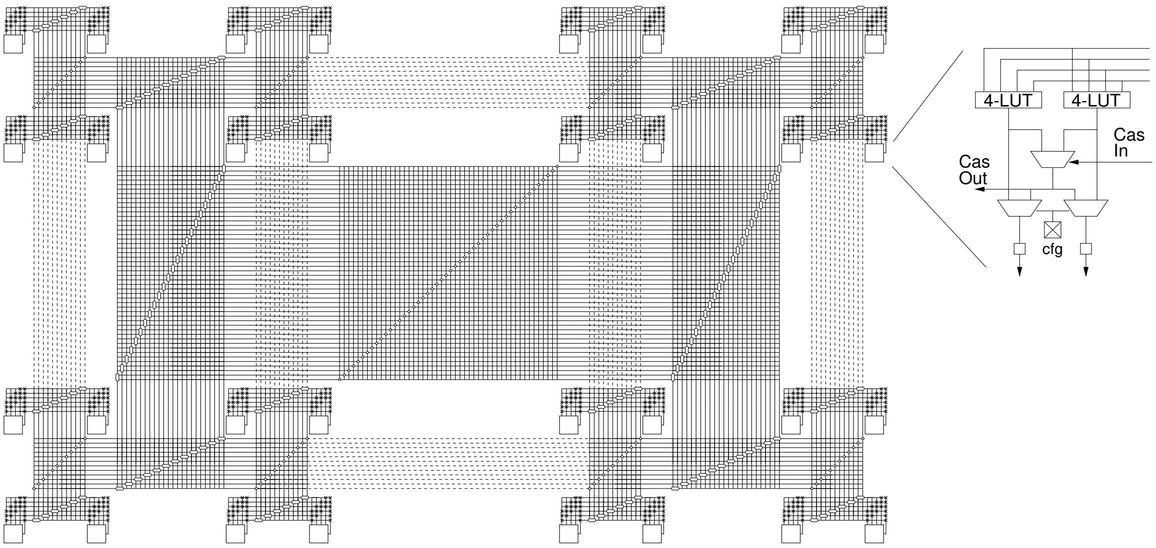

Figure 3.3: Compute page architecture. The shortcuts are shown as dashed lines. For clarity only 32 instead of 64 BLBs are shown, and the base channel capacity has been reduced to 6 wires, from 10 in the actual design.

much easier problem. The rate of capacity growth towards the root is selected to accommodate the expected wiring growth of typical applications, as captured by Rent's Rule[75], the exponential relation between the number of terminals of a circuit block and the number of gates in it. At every level in the tree, different types of switches can either double the branch capacity, or not. By choosing an appropriate sequence of switch types, any reasonable growth rate can be approximated. What distinguishes the fat pyramid from the fat tree is an additional set of wires, called *shortcuts*, that enable more direct connections between BLBs that are physically close but logically distant.

Extending the network above the level of a single compute page provides for the network that is used to connect different compute and memory pages together, as well as the main I/O to the CPU, as shown in Figure 3.1.

The array was designed with a fixed clock rate target of 250 MHz. To achieve such a high clock rate, the interconnect, which typically contributes most of the delay in FPGAs, is heavily pipelined. Because of that, the delay of a signal, in cycles, depends on the distance it travels along the network. This creates the need for special *retiming banks* at each BLB input, to realign signals that have suffered different delays along the network. The retiming depth is part of the BLB configuration and can range from 1 to 8 register stages.

The fat pyramid network and the fixed clock rate, supported by the pipelined interconnect, give the name HSRA, for *Hierarchical Synchronous Reconfigurable Array*, to this architecture. More details on the HSRA architecture can be found in [130], while the circuit implementation for this test chip is described in detail in [129].

3.2 The DRAM macro

The design of the memory page is based on a 2 Mbit synchronous DRAM macro, designed by LG Semicon. The design is targeted to LG's 0.25/0.4 μm "EDL" (Embedded DRAM/Logic) process. This process is for the most part a 64 Mbit stack capacitor DRAM technology with a fourth metal layer and a salicide layer selectively applied to the periphery. The design of the DRAM core follows the Bidirectional Matched Global Bitline (BMGB) architecture[5], aiming to increase the density of the DRAM core, while maintaining good performance.

The capacity of the DRAM macro can range from 1 Mbit to 8 Mbits, with 1 Mbit banks, 1K bits wide with 1K rows tall. In our test chip we have selected the 2 Mbit, 2-bank configuration. An *enhanced* synchronous DRAM interface is provided, for the most part compliant with the PC-100 SDRAM standard. In addition to the common SDRAM functionality, the memory macro provides an additional row buffer in each bank. Each row buffer can hold one memory row (1K bits) and is loaded in parallel from the sense amplifiers on every bank activate command. Row buffers are managed with a "write through" policy, so no explicit writeback from the row buffer to the array is required. In this configuration, during read-only streams the row buffer allows the bank to be precharged while column

Feature size	0.25 μm
Capacity	2 Mbits
	(2 banks x 1K rows x 1K bits)
I/O width	128 bits
Operating frequency	125 MHz
CAS latency	2 cycles
Row cycle time	56 ns (7 cycles)
Dimensions	2.4 x 3.6 mm^2

Table 3.1: DRAM macro characteristics

reads are in progress. This technique can be used to hide the precharge latency behind a continuous read stream, even when just one bank is used. In the same way, it makes it possible to hide refresh cycles behind read streams.

The interface is 128 bits wide and provides the capability of masking writes at a byte granularity. It operates at a maximum operating frequency of 125 MHz, so it can achieve a peak bandwidth of 2 GBytes/sec.

Table 3.1 summarizes the DRAM macro characteristics.

3.3 Design of the memory page

In addition to the given functionality of the DRAM macro, the memory page has to provide a number of functions specific to our architecture. First of all, it was decided that the memory page will provide a pseudo-SRAM interface to the user logic, as described in Section 2.3. To provide this functionality, a controller was implemented that maintains tags for the 2 memory banks and provides a handshake signal to the user logic. In addition, the compute page is designed to run at twice the clock frequency of the DRAM macro and appropriate rate matching is required at the interface. Furthermore, an interconnect network is required to allow a flexible mapping of logical signals (address, data and command signals) to the Trumpet network wires. Also, as with the BLB inputs in the compute page, the memory page inputs need some degree of built-in retiming. Transferring a configuration or state bit stream between the compute and memory pages requires some control and address sequencing. The compute page clearly cannot provide this kind of control during those times. In addition, largely because of the need for retiming and configurable interconnect, the memory page has some configuration and internal state of its own. These configuration and state bitstreams will have to be dumped and reloaded on every context switch. Block I/O can be performed directly between the memory and the external chip interface and some addressing and control is required for that. Finally, it was decided that some support for DRAM refresh will be provided by the memory page.

It is evident from the above that a fair amount of logic is needed aside from the raw DRAM macro. Ideally, we would like to have the opportunity to build a memory controller directly on top of the raw memory core. However, the LG memory macro provides an almost complete enhanced SDRAM interface that cannot be bypassed. The reasoning behind this approach was that the DRAM macro could be used in a number of different projects, by

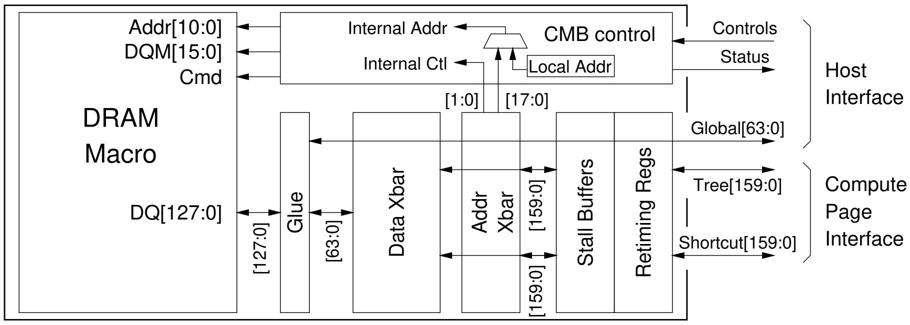

Figure 3.4: CMB block diagram (not to scale)

different organizations, so some kind of standardized, general purpose interface should be provided. It was decided by the LGS engineers that the existing PC-100 standard would be a good choice, since it is an interface with which most designers are familiar. Although this choice may be good for the long term, it was not ideal for our needs. Much of the functionality of the SDRAM interface is unused in our application, while accessing the memory core through the limited repertory of SDRAM commands sacrifices some flexibility.

We have called the combination of the DRAM macro and external logic a *Configurable Memory Block*, or CMB, a term more or less equivalent to memory page. A high level block diagram of the CMB is shown in Figure 3.4. The interface logic consists of a full-custom datapath that provides the interconnect, retiming, and circuits used to hide the variable memory latency from the user logic, and a standard cell-based controller that generates the required timing and issues commands to the DRAM macro. This controller has been designed to interface to some external agent, receive commands, execute them and return status signals. In the complete architecture of Figure 3.1, that external agent would be the on-chip CPU, while in the test chip design that role is assumed by a host PC.

Physical interfaces

The CMB provides two separate interfaces, one to the user logic executing on the compute page and one to the external controller, or host.

The host interface is implemented on dedicated wires between the host and the CMB controller. Through this interface the host issues one of 15 possible commands to the CMB controller and monitors its state. These commands are listed in table A.2, on page 146. In addition, user I/O and transfer of configuration bitstreams is performed through this interface.

The interface to the user logic consists of a set of 160 tree wires and 160 shortcut wires. All the logical signals of the memory interface – address, data, and control – are mapped on a subset of these wires, under the control of appropriate configuration bits. In addition to that, a dedicated wire carries a `Stall` signal, generated by the CMB controller. This signal is used to implement a stall mechanism that masks the variable latency of the DRAM, as explained in the following.

Application execution

In response to the `RUN` and `STOP` commands from the host, the CMB controller enters the `RUN` and `IDLE` states respectively. These states correspond to application execution and suspension. After chip reset the controller wakes up in the `RESET` state, where the DRAM initialization sequence[2] is run. Then, the CMB controller moves into the `IDLE` state. Once a configuration and initial state has been loaded to both the compute and memory pages, the `RUN` command from the host, issued to both pages at the same time, will start the application execution. As long as the CMB controller is in the `RUN` state, the appropriate tree and/or shortcut I/O's function as address, control and data terminals, as determined by the CMB configuration.

The compute page controller contains a cycle counter, a free-running down-counter that the host can load with an initial value. This is used when execution for a fixed number of cycles is required. The state of the cycle counter is visible by the CMB controller. The CMB controller will remain in the `RUN` state until either a `STOP` command is issued by the host, or the cycle counter expires. Then it returns to the `IDLE` state and is ready again to accept commands from the host.

Clock rate matching

Since the compute page is designed to operate at 250MHz, read/write requests may arrive at the CMB interface at a maximum rate of one per 4ns. On the other hand, the memory can only process requests at half that rate. Therefore, in the general case, the CMB will not be able to support a request per cycle and stall cycles will be unavoidable. In certain cases however, the CMB controller can take advantage of locality to avoid unnecessary stalls,

[2] A number of refresh cycles, as in commodity DRAMs.

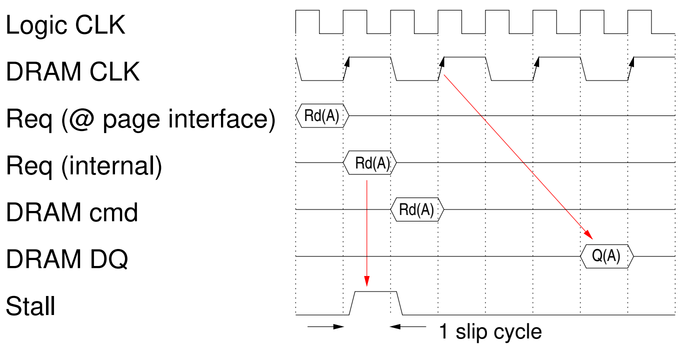

Figure 3.5: Single read request that slips. Normally the DRAM request would be issued immediately after the request is received, but because of the “wrong” phase of the DRAM clock, it is delayed by one cycle. `Stall` is raised for 1 cycle, causing the compute page to stall for a cycle and remain in lockstep with the CMB. Note that there is one pipeline stage at the CMB interface.

increasing the sustained memory bandwidth.

The CMB operates on the same clock as the compute page. This clock is divided by 2 to generate the DRAM macro clock. The CMB controller looks at the phase of the DRAM clock and aligns incoming requests to its rising edge. When a request arrives during the high phase of the DRAM clock, it will “slip” by one 4ns cycle, causing a single-cycle logic stall, waiting for the low phase of the DRAM clock. This situation is illustrated in Figure 3.5.

A continuous sequence of requests from the logic may result in every other logic clock cycle being a slip cycle. This situation can be avoided when there is sufficient locality of references. When a read request is received by the CMB controller, a full 128-bit word is read from the memory and placed in a register, while the appropriate bits are returned to the compute page. When a request is received in the “wrong” phase and it addresses part of the same 128-bit word as the last request, the data is returned directly from this register. Essentially this register amounts to an additional level in the memory hierarchy. In this way, a sequential stream can sustain a request per logic cycle with no slip cycles, except possibly for the first request. Streams that consist of sequential segments will also be mostly slip-free, with only a possible slip cycle between segments.

In any case, the maximum bandwidth delivered by the DRAM macro is 128 bits times 125 MHz, or 2 GBytes/sec. This corresponds to 64 bits at 250 MHz, so the data bus width at the interface between the compute and memory pages is limited to 64 bits.

Managing the variable latency

The relative advantages of DRAM-like and SRAM-like interfaces were discussed in Section 2.3. Both types of interfaces were considered for Trumpet, including the possibility of providing both, selected by a configuration bit. Eventually, because of scheduling constraints, it was decided to implement only one. Between SRAM and DRAM, the SRAM was deemed more interesting because it exposes all the latency management issues. In addition, it would make initial application development easier.

As is common with SRAMs, the application logic does not distinguish between row and column addresses, but sees a flat address space instead. A request consists of a `Req` strobe, bundled with a read/write control (`RWB`) and the data address. The CMB controller examines the address provided to determine if the DRAM row that contains the selected word is currently active. This is accomplished by comparing the incoming address to a pair of tags, one for each DRAM bank. When the addressed row is already active, only a column access, read or write, needs to be issued to the DRAM macro. In the case of reads, the requested word will be returned to the user logic within a minimum latency of *6 logic cycles*. This is determined by the minimum latency of the DRAM (2 memory cycles or 4 logic cycles) plus 2 cycles needed for the request and returned data to traverse the CMB interface, which is registered in both directions.

Common SDRAMs support pipelining of column requests and the CMB interface exports this capability to the user. As long as requests from the logic "hit" in the 2 rows present in the row buffers, up to one request per cycle can be satisfied, subject to the slip conditions mentioned earlier as well as the mix of read and write requests. This second limitation arises from the fact that the DRAM column latency is longer than one cycle; when a write follows a read, it has to wait for the read data to be returned and clear the DRAM DQ bus, to avoid collision. This is a common limitation of the standard SDRAM interface, that can be avoided by some recent techniques[88, 114]. The CMB handles this situation by stalling the compute page for the appropriate number of cycles, when a write follows a read too closely. This is a *write-after-read*, or *WAR* stall. In the worst case, when read and write requests are issued back to back, a 4-cycle WAR stall will result, followed by a single cycle slip, as illustrated in Figure 3.6.

When the DRAM row addressed by an incoming request is not active, the CMB controller raises the `Stall` signal and proceeds to precharge the appropriate bank, activate

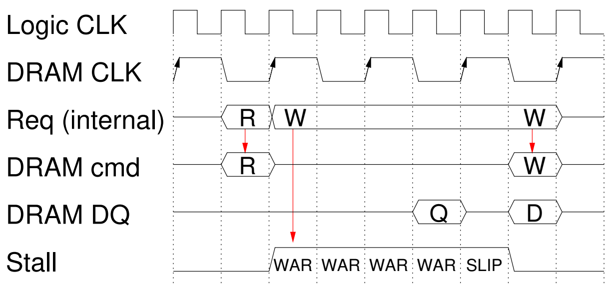

Figure 3.6: Worst-case WAR stall. After 4 WAR cycles, the write request slips for an additional cycle.

the addressed row and wait for the DRAM row access latency. After that, `Stall` is deactivated and the column command, read or write, is issued to the DRAM macro. In the compute page the `Stall` signal is used to gate the logic clock. In this way, during the time that the row access is in progress, the application logic is idle and the *perceived latency* is just the DRAM column latency. With the DRAM macro's 2-cycle precharge latency (t_{RP}) and a 3-cycle RAS-CAS delay (t_{RCD}), the equivalent miss penalty is 5 DRAM cycles, or 10 logic cycles, as shown in Figure 3.7.

Dealing with stall distribution and memory latencies

The above stall policies, slip, WAR and row miss stalls are conceptually quite simple. An important complication arises though, because of two facts: First, there is a certain latency associated with the distribution of the stall signal to the compute page and the actual stalling of the logic clock. This latency would be more pronounced in large devices, where distributing the stall signal to all compute pages across a large die would necessarily take a few clock cycles. In Trumpet, with a single compute and memory page, one clock cycle might be sufficient, but we chose to emulate a longer latency, in order to

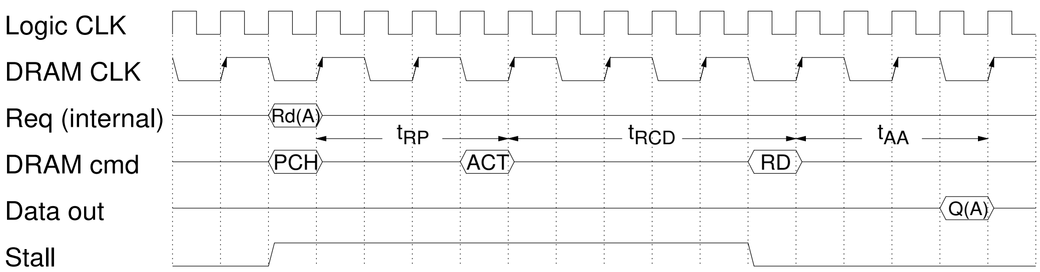

Figure 3.7: Timing diagram for an isolated row miss

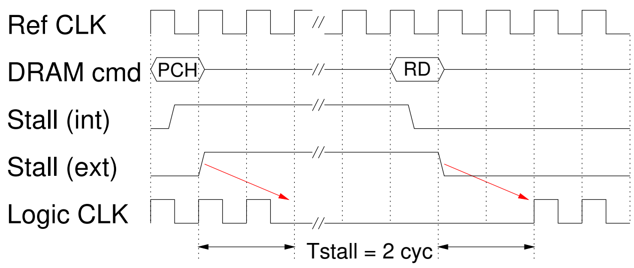

Figure 3.8: Definition of stall distribution latency

expose the related control and timing issues. The internal `Stall` signal is registered at the CMB output (as are the read data) and then registered again at the compute page input. The compute page controller will shift this by another half cycle and use this delayed signal to gate the logic clock. This corresponds to a stall distribution latency of *2 cycles*, as measured from the external `Stall` to the first missing pulse of the compute page clock, as shown in Figure 3.8. The result is that once the stall signal is raised by the CMB, some more requests may arrive from the compute page, before the logic clock is stalled. Second, the DRAM read latency is also more than a cycle, namely 4 logic cycles. Because of that, read requests that were issued to the DRAM macro shortly *before* a stall was detected will return their data *after* the stall. At that point, the compute page may be stalled and unable to receive that data.

The solution to the above problems is a set of *stall buffers* at the inputs and outputs of the CMB. During normal execution of the application, when no stalls are in effect, requests are transmitted to the DRAM immediately and read data are returned to the compute page directly. Once a stall condition is detected, `Stall` is raised and the DRAM I/O's are temporarily redirected to the stall buffers. Figure 3.9 shows a simplified block diagram of the stall buffer implementation and related timing.

Five distinct periods can be distinguished with respect to the stall buffer operation. During period (1), equal to the stall distribution latency, requests keep arriving from the compute page, because the logic clock is still active. These requests are buffered at the input stall buffer. At the same time, read data that come out of the DRAM macro can still be delivered to the compute page. However, during period (2) read data may still arrive at the DRAM macro outputs (because of the DRAM read latency), but the compute page is unable to receive them, because the logic clock is now stopped. These data will be buffered in the output stall buffers. Period (3) is quiet, without any requests or read data moved.

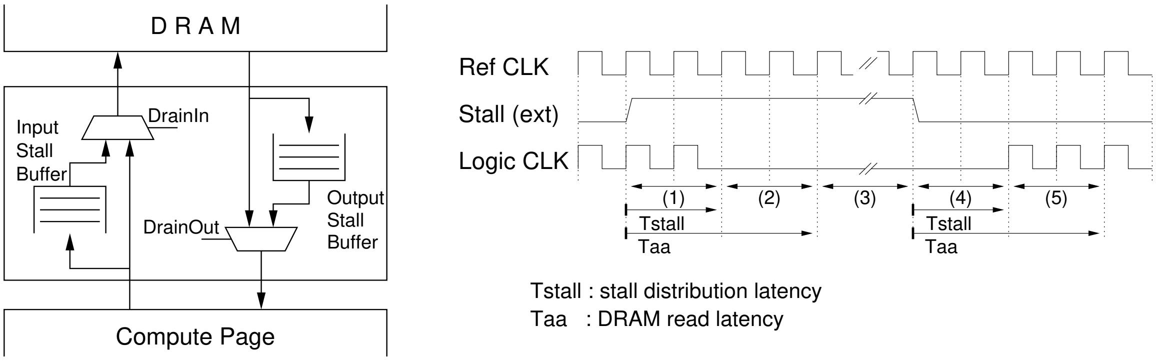

Figure 3.9: Stall buffer block diagram and timing

The last read command issued to the DRAM could have been just before period (1) and the result would have already been returned by the end of period (2). During period (3) the CMB is working on the stall condition, for example performing a row access. During period (4) the CMB is again available, but because of the stall distribution latency the compute page has not resumed yet. This is the time to empty the input stall buffer from the requests that arrived during period (1). Finally, by period (5) the compute page wakes up, but no new read data is returned by the DRAM yet. During this time, the output stall buffer is emptied and the data stored there are delivered to the compute page. In a sense, periods (4) and (5) are the duals of (1) and (2), with exactly the same duration (2 cycles each), allowing both input and output buffers to be drained. After the end of period (5), everything returns to "steady state", where the stall buffers are idle and empty. At that time the direct input and output paths are reestablished. Note that these periods can overlap – for example when the duration of the stall is short (as in a slip stall), or when back-to-back misses occur.

Interesting scenaria arise when multiple stalls occur back-to-back – for example slip, followed by miss, followed by write-after-read. This required careful design and verification of the CMB control logic and turned out to be the most time consuming part of the controller design. A large number of test cases, both directed and random, were generated and simulated at the Verilog level, to ensure proper operation of the stall logic in all possible combinations of stalling requests, as well as potential unseen situations.

The depth of the stall buffers is determined by two factors: The input stall buffer must be as deep as the stall distribution latency, so that it can absorb all requests issued by the compute page after a stall condition has been detected. The output stall buffer must be as deep as the memory latency, as measured from outside the CMB (including the CMB input and output registers), for the following reason: when the host issues a `STOP` command, both the compute and memory page receive it at the same time. The compute page clock is stalled immediately and the CMB does not issue any more requests to the DRAM macro. But for a number of cycles equal to the memory latency, data will keep coming out of the DRAM macro. These need to be buffered in the output stall buffers and be delivered to the compute page once the next `RUN` command is received. In Trumpet, the minimum memory latency of 6 cycles sets the stall buffer depth to 6 slots.

This scheme is flexible enough to allow operation of multiple memory pages on larger devices. In that case the stalls generated by individual memories would have to be combined in some way. The simplest approach would be to OR all the individual `Stall` signals, to generate a `Global Stall`, that is distributed to all compute and memory pages. This approach is simple, but would cause the entire device to stall when any of the memory pages encounters a stall condition. Alternatively, one could distribute the stall signal from each memory only to the pages that actually depend on it. Such dependencies could be extracted from the application source code and encoded in the configuration. This technique would allow forward progress to be made by pages that do not depend on the stalling memory, but requires much more complex control, since pages should be allowed to get out of lockstep.

DRAM refresh

There is little hardware support provided in the CMB for refresh. Most of the related functionality was left for the external controller (the host), with the reasoning that in an environment with multiple pages the common functionality should not be replicated in every memory page.

The standard *Autorefresh* command of SDRAMs can be used to refresh one memory row. This operation is triggered by the host with the `REF` command, when the CMB is in the `IDLE` state. The host is programmed with the refresh interval and the number of rows to refresh each time, and performs a countdown. When the refresh timer expires, the host must examine the state of the memory page. If it is in the `RUN` state, the host will interrupt execution with a `STOP` command and then issue one or more `REF` commands. Upon receipt of each `REF` command, the CMB controller will perform an autorefresh operation, refreshing one row of memory, as in conventional SDRAM. Finally, if an application has been running, it will be restarted with a `RUN` command. Alternatively, instead of using the autorefresh command, the host can issue a dummy I/O command every time a memory row needs to be refreshed. When the CMB enters the I/O operating mode, the DRAM row containing the initial address will be activated, and thus refreshed. Although this approach is slightly more complicated, it has the advantage that the refresh range may be bounded, when only a small part of the memory is in use.

With both of the above approaches, refresh is transparent to the user. It does however interfere with application execution, since the user logic accesses the DRAM directly. With no cache or any similar structure to isolate the DRAM from the memory reference stream, the application has to be interrupted every time a refresh operation must be performed. This interference impacts application performance slightly. For Trumpet, the resulting overhead is always under 5%.

Configuration and state transfers

The process of loading and unloading configuration and state bitstreams is very similar for the compute and memory page. Reconfiguration is triggered by an appropriate command issued by the host. Configuration bitstreams can be loaded into the active configuration store of the compute or memory page either from the external interface, or from the internal memory. Configuration from the external interface takes place at the limited external I/O rate of 125 MBytes/sec (32 bits at 31.25 MHz). When configuration bitstreams are preloaded into the CMB memory, configuration load can proceed at the peak memory bandwidth of 128 bits at 125 MHz, or 2 GBytes/sec. The same peak transfer rates are applicable for the transfer of state bitstreams between the compute or memory page internal registers and the external or internal memory.

A *context switch* consists of a state unload, configuration load, and state load, for a number of compute and memory pages. When the relevant bitstreams are stored in external memory, all transfers have to be serialized and the total time depends on the number of pages involved. If the internal memory is used instead, the configuration and state transfers for each pair of compute and memory pages can happen in parallel. In that case, the total time for a complete context switch of the entire device is equal to two state unloads, two configuration loads, and two state loads (one for the compute page and one for the memory page).

Based on the above rates, Table 3.2 summarizes the latencies for the various bitstream transfers. The time for context switch from the host is given for a single page pair. We observe a factor of 16 difference between the external and memory transfers, which corresponds to the 4X difference in clock rate and 4X difference in width, between the internal and external interfaces.

	Length [Kb]	Host xfer	Internal xfer
Memory state	3.000	3.07	0.19
Memory config	3.750	3.84	0.24
Compute state	17.125	17.54	1.10
Compute config	68.250	69.89	4.37
Context switch	112.250	114.95	7.19

Table 3.2: Configuration and state bitstream transfer latencies. Times in μsec.

Interconnect

In terms of area, a significant part of the CMB interface is taken up by the interconnect structure. This consists of a pair of circuits, the *address and data crossbars*, designed to allow a flexible mapping of the logical CMB I/O's (address, data and control bits) to the internal wires and eventually to the DRAM macro terminals. Despite the name, these networks are less than complete crossbars. The design of the address and data crossbars is described in detail in Appendix A.

Retiming

As in the compute page, all CMB inputs include a configurable retiming bank. Each bit can be skewed independently for a number of cycles, in order to compensate for delays sustained through the pipelined interconnect network of the compute page. At the CMB interface all inputs and outputs are registered once. In addition, inputs can be retimed for up to 4 cycles in the retiming bank.

3.4 Programming Trumpet

In this section a brief description of the programming environment for Trumpet is given. All related tools were developed in parallel with the hardware design and were used for functional verification of the hardware. Figure 3.10 illustrates the tool flow for verification with `timemill` and `Verilog-XL` and for execution on the actual hardware. Note that for the most part, the tool flow is independent of the target, with the only variation in the last one or two steps.

Prototype testing framework

The testing framework designed for the Trumpet prototypes is based on a PC host that communicates with Trumpet over its PCI bus. The path from the host processor to Trumpet consists of a PCI interface board, built around the `PLX 9050` PCI bridge chip from PLX Technology, and a Xilinx FPGA.

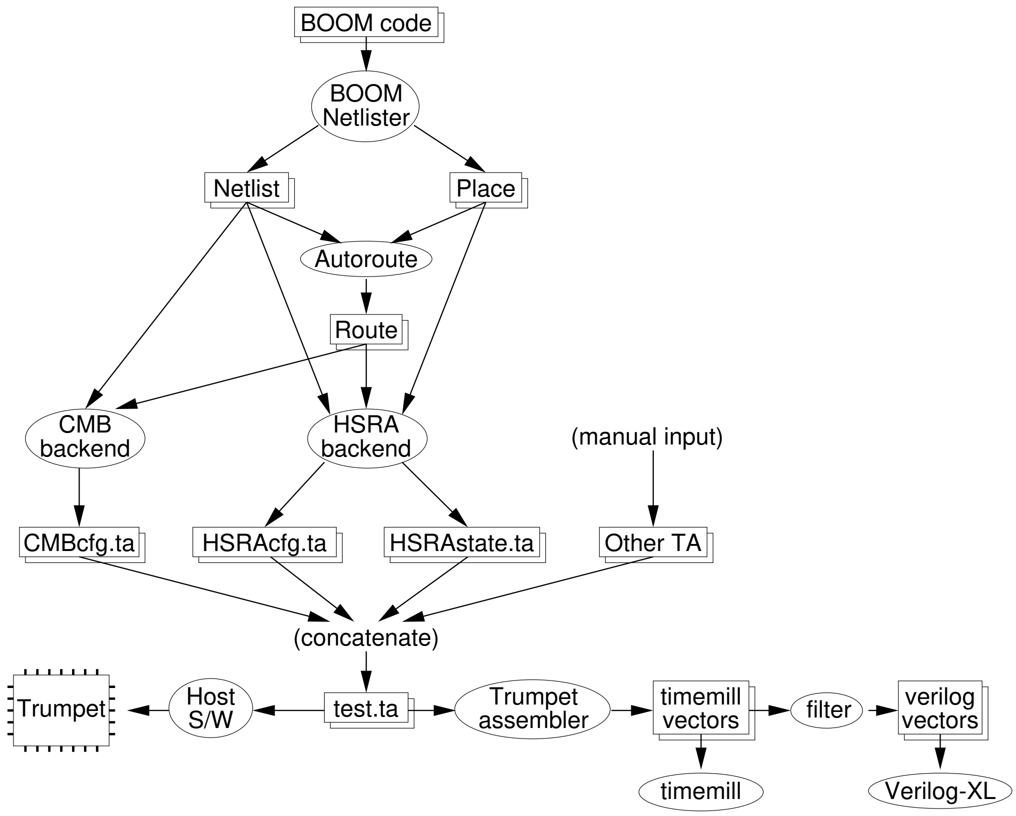

Figure 3.10: Trumpet programming tool flow

The FPGA contains a number of registers, mapped into the PCI address space, that mirror internal Trumpet registers and the Trumpet I/O's. A write from the host into one of these registers will trigger the appropriate strobes at the Trumpet interface, so that the value written will eventually be transferred into the corresponding Trumpet register. A read from an FPGA register will return to the host the appropriate value from the Trumpet registers or I/O's. In addition, the FPGA handles refresh. Because of the real time requirement, we cannot rely on the host for this function. Instead, the host will program the FPGA with the refresh interval and the number of rows to refresh each time, the FPGA will run a countdown timer and perform the refresh sequence each time the counter expires. Finally, the FPGA contains a number of registers that control various system parameters, such as the value of the various programmable power supplies and the clock frequency.

Trumpet assembly

All communication between the PC host and Trumpet happens in the form of reads and writes to and from the FPGA register space. For example, to set the CMB into the Configuration-Load-From-DRAM state, the host will write the `LCD` op-code to the corresponding FPGA register. During the `GREAD` state (global output), the host repeatedly reads the FPGA `data` register, that simply reflects the value of the Trumpet 32-bit I/O bus. The appropriate sequence of register reads and writes is specified in the so called *Trumpet assembly* language, a very simple language, consisting of just two instructions: read and write. All tests are coded as a sequence of Trumpet assembly instructions. During the hardware development, for the purpose of functional verification, a simple tool, the trumpet assembler, was used to translate these tests into either Timemill or Verilog test vectors. During actual testing of the prototype a piece of software on the PC will read in the test code and perform the appropriate reads and writes.

BOOM

Development of applications for Trumpet begins with the `BOOM` (*Berkeley Object Oriented Module generators*) system[23]. `BOOM` is a Java-based programming environment that supports simulation and netlisting, among other features. Design entry is performed in Java, with logic modules represented as Java objects and composed in a hierarchical way. The design can be simulated and mapped to 4-LUTs and 5-LUTs and a BLIF netlist can be generated. Placement is performed manually: the programmer embeds placement directives in the source code and a placement file is generated during netlisting.

A sequence of in-house tools performs routing and generates configuration bitstreams for both the compute and memory pages, based on the netlist and placement files. The final configuration bitstream is just a sequence of trumpet assembly write instructions, that write the appropriate values into the external FPGA `data` register and eventually the configuration memory of Trumpet.

3.5 Design methodology and verification

Trumpet was implemented with a mix of full-custom design and standard cell methodology. The design of the HSRA was entirely full-custom. Development of the CMB started with a behavioral Verilog model, that was used to refine and fully specify the implementation details and verify functionality at the logic level. Care was taken to code the controller part in a synthesizable way. Then, the detailed circuits for the datapath were designed and custom layout was performed. The controller model was compiled to the `Compass` 0.4μm logic library, placed and routed. The DRAM macro was designed by engineers at LGS and delivered to us as a behavioral Verilog model and a layout outline. In addition, a datasheet was provided with information such as a complete functional specification, detailed timing parameters, terminal capacitive loads, and so on.

Functional verification was initially performed for each major component (HSRA, CMB, DRAM macro) in isolation. The verification of the CMB was based on the Verilog model of the DRAM macro provided by LGS and our model of the CMB. For the transactions with the host (e.g., I/O), appropriate code segments in Trumpet Assembly were coded by hand. A simple filter sequence was used to translate those code segments to Verilog stimuli for the host interface of the memory page. More involved was the verification of the interface to the compute page, with a large number of combinations of stalling requests, and combinations of read, write and address widths, as well as variations in the routing and retiming. A combination of directed and random tests were used for this purpose. Initially, a simple configuration was generated for the CMB, (8-bit input, 8-bit output, mapped to a fixed set of CMB terminals) to be used for all the directed tests. Short test sequences were developed for all the envisioned combinations of stalling requests: slip cycles, row misses, WAR stalls and halt-restart commands from the host. In addition to that, a set of tools was created to generate random CMB configurations (in terms of I/O width, terminal mapping and retiming depth), and random sequences of requests. Another tool was developed to "replay" the same request sequences in software (where the DRAM macro is emulated by a simple array), in order to generate the correct output. These outputs were then compared to the outputs of the Verilog simulation. Several thousands of cycles were run and certain very obscure bugs were uncovered this way.

Timing analysis was performed with `Hspice` for the critical paths in the datapath and `Design Compiler` for the controller. Unfortunately with a proliferation of new tools in our design flow we were not able to setup a full wiring parasitic extraction flow in time. Our timing simulations were based on estimates of wiring parasitics, both capacitive and resistive for long wires. These parasitics were explicitly annotated in the schematics.

Once the whole chip was assembled, full-chip functional verification was performed with `Timemill`, with an appropriate netlist for the DRAM macro provided by LGS. For the interactions between the memory page and the host, the same test cases as before were used. For the interface between the compute page and the memory page, a simple application was coded in `BOOM` and the appropriate configurations were generated for the compute and memory pages. In the resulting test code, these configurations are downloaded into Trumpet and about 200 execution cycles are simulated. The testing at this stage is much less rigorous than the standalone CMB testing, mainly because long test sequences would require developing and coding appropriate applications for the compute page. On the other hand, the standalone testing of the CMB has verified the stall logic and the only untested component at this point is the physical connection between compute and memory pages – whether requests and results can be transmitted across the interface.

3.6 Evaluation

Trumpet was fabricated by LG Semicon in 1999. Unfortunately, the DRAM was not functional and could not be repaired with the built-in redundancy. Therefore, Trumpet has not been tested and all the results discussed here are based on simulation.

A major problem with the CMB design turned out to be the performance of the controller. In the design that was fabricated, there is a very long critical path, that was not possible to restructure in the limited time available before our tapeout date. This path starts with the address input, registered at the inputs of the address crossbar. After the appropriate input bits have been selected and forwarded to the core, they pass through a multiplexer that inserts the appropriate bits from the *local address*, an internal register used for sequential addressing. Then, the bit corresponding to the bank address selects one of the two tags and the appropriate tag comparison takes place, to generate the `Hit` signal. This signal is then used in the controller for the next state logic, as well as the current output logic. This path is estimated by the logic synthesis software (Synopsys `Design Compiler`) to run at a 8.68ns cycle, or 115 MHz, in worst-case conditions.

Another issue is the size of the CMB. As seen in the die photograph (Figure 3.2) and summarized in Table 3.3, the actual DRAM, including the memory core, read/write datapaths, decoders and redundancy fuse circuits, occupies slightly over a third of the total CMB area. Another third is taken by the standard cell based SDRAM interface controller, the central area in the memory page layout. The remaining area, also close to a third of the total, is the CMB interface logic. As discussed earlier, the SDRAM interface was not needed for our application, but was part of the macro as provided to us by LGS. Without it, the CMB implementation could have been about 30% smaller.

Block	Area (mm^2)	Fraction
Raw DRAM	4.53	32.7%
SDRAM controller	4.32	31.1%
CMB datapath	3.21	23.1%
CMB controller	0.69	5.0%
Miscellaneous	1.12	8.1%
TOTAL	13.87	100.0%

Table 3.3: CMB area breakdown

A follow up design has been developed, to deal with both the above problems. This design is based on a Verilog model of the DRAM core without the SDRAM interface controller. The design of the CMB controller has been retargeted to work with the simplified DRAM model. Interestingly, there was no increase in the complexity of the CMB controller – instead, a small reduction in the number of gates was observed (both versions are about 2000 cells total). In addition, the critical path in the Trumpet design has been spilt into two cycles: in the first, the address bits are routed through the address crossbar, the local address bits are inserted, and the tag comparison is performed. At that point, the `Hit` signal and the address bits are registered. In the second cycle, the remaining control logic for the next state and the controller outputs is performed. Some small additional optimizations have been performed. These changes have led to a significant improvement in performance, at the expense of an additional cycle of latency for reads. However, the initial target of 250 MHz was still not achieved. With other critical paths surfacing, the maximum operating frequency achieved (in worst-case conditions) is now 160 MHz. In the last design the most critical paths involve controller outputs driving the datapath. These outputs generally drive a large load and need to be buffered up accordingly, while a significant setup time in the datapath must be met. It is difficult to isolate one or two specific "problem" paths, because a number of them are of comparable criticality. It appears that only small improvements can be made beyond this point without a major redesign effort. Because of the difficulty in achieving a high clock rate, a more conservative clock rate is used in chapters 5 and 6, where embedded memory systems are compared to conventional external memory ones. A 200 MHz clock is assumed there, with a 0.18/0.25 μm design rule for memory and logic respectively.

In terms of functionality, at some point it became apparent that a "stream" mode, where accesses are assumed to be sequential and the CMB only listens to a "next element" strobe, would be very useful. Such a mode would be ideal for supporting SCORE streams. In SCORE, the user logic does not know if its I/O's are connected to another piece of logic (another *operator* in SCORE terminology), or to a stream buffer. Buffering of streams is completely transparent to user logic. In that case, the address and read/write controls cannot be provided by the logic. Instead, the memory should maintain an internal address counter, while the type of each access, read or write, should be implied by the current operating mode, since streams are unidirectional. These ideas were developed at the same time frame as the Trumpet design and it was too late for them to be integrated into the current CMB specification. It is however estimated that implementation of such modes would be quite easy. Sequential addressing is already implemented for most transactions (global I/O, configuration load/unload, etc). Making this capability available to the logic in the compute page would only add a small amount of logic in the controller and would require no additional datapath resources.

The CMB consumes about 315 mW and 255 mW of power at 125 MHz and nominal conditions for continuous 8-bit wide read and write streams respectively, with half of the bits changing value every cycle. Of that, 195 mW are consumed in the CMB logic. More than half of this, or about 120 mW, are consumed on the clock load and internally in the clock buffer. This is a consequence of the register-rich design, that provides retiming banks and stall buffers for every address and data bit, for a total clock load of 58pF. Although gated clocks have been used in certain cases, the power breakdown suggests that more extensive use of such techniques might have been appropriate. For wider data streams, power dissipation scales linearly with the word width, to 420 mW and 380 mW total for 64-bit wide read and write streams respectively.

Chapter 4

Memory density trends

The high density of DRAM is the primary motivation for the development of embedded DRAM technologies. Depending on the situation, there are two ways that systems may benefit from embedded DRAM: On one hand, with a given silicon budget, DRAM allows a larger amount of memory to be implemented on-chip, with potential performance and/or energy benefits. What these benefits are depends largely on the architecture. For the FPGA-based reconfigurable architecture that is the focus of this work, the performance benefits resulting for high on-chip memory capacity are discussed in detail in Chapters 5 and 6. On the other hand, when a fixed memory capacity is required, the higher density results in a smaller die, reducing the cost. Usually the small die more than compensates for the somewhat higher fabrication cost of embedded DRAM wafers. These area savings have already led to low cost embedded DRAM based products, such as the cache chips marketed by MoSys.

This chapter starts with a survey of existing commodity DRAMs and SRAMs and embedded DRAMs. Density trends for all three technologies are examined. Then, the Trumpet CMB design is examined and evaluated in terms of its area efficiency. Then, we look at an alternative CMB design based on DRAM, as well as to an SRAM-based design, for the purpose of comparison between pure logic and embedded DRAM technologies.

4.1 General trends

With F being the minimum feature size of a given process technology, a DRAM cell typically falls in the $12F^2$ to $15F^2$ range. For the common folded bit line architecture

	DRAM				SRAM			
	Capacity	Kb/mm^2	F^2/bit	Ref	Capacity	Kb/mm^2	F^2/bit	Ref
0.3μm	256 Mb	741.4	(14.6)	[146]	4 Mb	28.9	(375.5)	[15]
					4 Mb	51.5	(210.7)	[74]
	Mean	741.4	(14.6)		*Mean*	40.2	(269.9)	
0.25μm	64 Mb	465.7	(33.6)	[89]	4 Mb	31.8	(491.4)	[88]
	72 Mb	660.6	(23.7)	[123]	4 Mb	37.2	(420.0)	[62]
	72 Mb	747.7	(20.9)	[95]	4 Mb	35.5	(440.1)	[96]
	256 Mb	882.4	(17.7)	[10]	4.5 Mb	36.0	(434.0)	[90]
	256 Mb	652.2	(24.0)	[71]	2.25 Mb	74.9	(208.6)	[133]
	256 Mb	785.2	(19.9)	[125]	4.5 Mb	32.5	(480.8)	[43]
	256 Mb	1,092.5	(14.3)	[107]				
	256 Mb	786.7	(19.9)	[117]				
	1 Gb	1,147.2	(13.6)	[116]				
	Mean	802.2	(19.5)		*Mean*	41.3	(378.3)	
0.18μm	1 Gb	2,076.4	(14.5)	[81]	18 Mb	88.3	(341.3)	[147]
	1 Gb	1,917.0	(15.7)	[122]				
	Mean	1,996.7	(15.1)		*Mean*	88.3	(341.3)	

Table 4.1: Density of typical commodity DRAM and SRAM designs. SRAM designs are based on the 6-T cell, except for the ones in [74] and [133] that are based on the HRL cell. Note that the *effective* F^2 per bit is given, as these data come from measurements of complete chips. Data for the 0.25μm designs also appear in table 1.2.

there is a hard lower bound of 8F^2, resulting from the need to route one bit line in one dimension and two word lines in the other in every cell. SRAM cells on the other hand, come in three different flavors: the common 6-Transistor cell, the HRL, or highly resistive load cell and the TFT, or thin film transistor cell. Starting with the 6-T design, the other two aim at reducing the number of transistors by replacing the load PMOS devices by more area efficient structures: a highly resistive lightly doped polysilicon layer in the case of HRL, or thin-film devices, fabricated on top of the other four transistors, in the TFT case. In either case, a special *local interconnect* layer may be used to connect diffusion to polysilicon directly, allowing for a somewhat tighter layout. 6-T cells most commonly fall in the 150F^2 to 200F^2 range, while HRL or TFT cells come at or below 100F^2. However, these cells are becoming less popular in high capacity designs, because they suffer from larger leakage than the 6-T cell, while at the same time they require special process steps.

Based on the difference in cell sizes between commodity SRAM and DRAM, a common rule of thumb is that DRAM is about 10 to 15 times denser than SRAM. However the total area of a memory macro or chip design also depends largely on the size of the peripheral circuits. A survey of existing designs, summarized in Table 4.1, reveals an overall density ratio of about 20[1]. This could be attributed to a trend to make peripheral circuits much larger in common SRAMs than in DRAMs, since SRAMs are usually optimized for performance, while DRAMs are designed for maximum density.

The area trend is somewhat less clear-cut when it comes to embedded DRAMs. Such designs show a much larger variation in both cell size and yielded memory density, than their commodity counterparts. This could be in part because embedded DRAM is still a relatively immature technology, and in part because there is a larger variation in design features: generally, higher performance and wider I/O leads to a lower yielded density. For example, the two macros described in [60] are based on the same DRAM core design, but come with 64-bit and 512-bit I/O's, resulting in a large density difference, as summarized in Table 4.2.

Lithography	Technology	Capacity	Density		Ref
1.00μm	2-poly planar	1 Mb	13	(75.1)	[110]
0.45μm	Stack DRAM	16 Mb	169	(28.5)	[113]
0.25μm	Stack eDRAM	64 Mb	264	(59.2)	[124]
	Stack eDRAM	64 Mb	330	(47.3)	[92]
	Stack eDRAM	16 Mb	697	(22.4)	[3]
0.20μm	Trench eDRAM	16 Mb	745	(32.8)	[60]
	Trench eDRAM	16 Mb	1,365	(17.9)	[60]
0.18μm	Trench eDRAM	8 Mb	422	(71.4)	[121]
	Stack eDRAM	32 Mb	1,406	(21.4)	[145]
0.15μm	Trench eDRAM	16 Mb	814	(53.3)	[57]

Table 4.2: Density, in Kbit/mm^2 and F^2/bit, of current embedded DRAM designs.

The designs listed in Table 4.2 are based either on pure DRAM processes, or on "mixed" processes, that have a true DRAM cell, usually trench or stack. It is also possible to design a DRAM cell for a pure logic process. A variety of cell designs have been introduced. The simplest such cell is a variant of the true 1-transistor 1-capacitor, or 1T-1C DRAM cell, where the capacitor is implemented either as a transistor gate[36, 39] or drain/source diffusion and channel[45, 115]. A 3-transistor *gain cell* has also been used[54, 102], with

[1] Note the data in this table correspond to the total chip area, including the pad ring and other support circuits, like charge pumps, interface circuits, and so on.

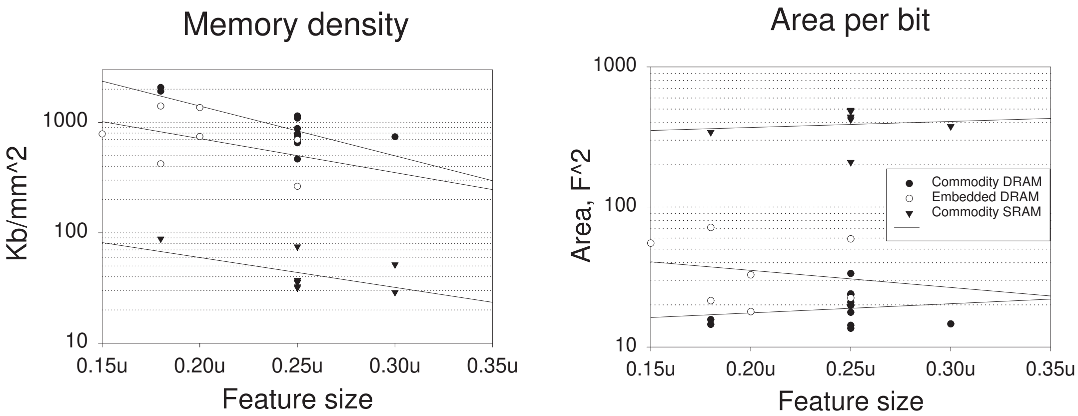

Figure 4.1: Effective memory density for different types of memories

a 2-transistor variation proposed as well[13]. Logic-based implementations of the 1T-1C design tend to be comparable with the HRL/TFT SRAM cells, with cell area around 100F^2 or less, while for gain cells [54] reports a 275 F^2 cell size.

Figure 4.1 compares the area trends of commodity DRAM and SRAM and embedded DRAM technologies. In both graphs, the effective, or yielded density is plotted. This includes the overhead for peripheral circuits and I/O. Note that in the case of embedded DRAM there is no pad ring, which gives those designs a slight advantage (about 5% on average). The area trend is shown as an average memory density, in terms of Kbits/mm^2 (left) as well as an effective area per bit, in multiples of the unit area, F^2. It is seen that while commodity DRAMs and SRAMs appear to follow fairly well defined and very similar trends, this is not the case with embedded DRAM, with a large variation in cell types and sizes, array designs and interfaces. Overall however, embedded DRAM tends to be much closer to its commodity counterpart, than to SRAM: the above data points show an average difference in yielded density of about *2X* between commodity and embedded DRAM and about *10X* between embedded DRAM and commodity SRAM.

4.2 Case study: the Trumpet CMB

The overall effect of the memory technology on an IC design depends directly on the amount of memory present in the design. Therefore, to gauge the potential of embedded

DRAM technology, one must consider the entire chip. In this section we revisit the Trumpet CMB design and look at the effective memory density resulting from the different memory technology options. Trumpet was designed with an approximate 50-50 area split between the memory and compute pages. Once all the peripheral circuits were included in the CMB design, it became apparent that the amount of actual memory was comparatively small. Still, it is easy to see that the choice of memory technology still makes a significant difference, under both the fixed area and the fixed memory capacity scenario. In designs with lower periphery overhead or coarser granularity, the impact of technology should be greater.

Original design

As discussed in Chapter 3 and summarized in Table 3.3 (repeated here for reference), the actual memory occupies only about a third of the CMB area. The SDRAM interface controller occupies another third and the CMB logic takes up the remaining third.

Block	Area (mm^2)	Fraction
Raw DRAM	4.53	32.7%
SDRAM controller	4.32	31.1%
CMB datapath	3.21	23.1%
CMB controller	0.69	5.0%
Miscellaneous	1.12	8.1%
TOTAL	13.87	100.0%

Table 4.3: CMB area breakdown

The memory macro design, developed for a 0.25μm DRAM technology, is based on an approximately 0.6μm^2 stack cell. To maintain a high memory density and good performance at the same time, the array design is based on the Bidirectional Matched Global Bitline (BMGB) scheme[5], a hierarchical bit line scheme that minimizes the number of sense amplifiers while maintaining a relatively low bit line capacitance. This results in an effective memory core density[2] of about 930 Kbits/mm^2, which is typical for DRAMs of this generation. However, the *small capacity* and *wide I/O* of the memory macro results in a comparatively large area overhead for the peripheral circuits. Once the basic memory design is considered, including row and column decoders, datapath, redundancy fuses, and so on,

[2]Cell array, sense amplifiers and word line drivers

the effective density drops to 450 Kbits/mm^2. In other words, the peripheral circuits in our macro take about as much area as the memory core itself. This is about half the effective density of commodity DRAM parts, as listed in Table 4.1. Furthermore, this number does not include the SDRAM interface controller, implemented as a large standard cell block. With the controller included, the effective density drops down to 230 Kbits/mm^2.

If to that we add the CMB logic and view the entire CMB as the basic memory block, the overall density is further reduced to 150 Kbits/mm^2. Of course, we may argue that the CMB is more than a simple memory block, since it also implements routing, retiming and stall management functions, which are normally absent from a conventional memory. Still, this is a useful metric when comparing the effect of different technologies on the CMB design.

Retargeted CMB design

To factor out the effect of the unwanted SDRAM controller, the CMB design was retargeted to work with the bare memory design. In this design, the CMB control logic is connected directly to the memory decoder, datapath and timing circuits via a very simple interface: separate row, column and bank addresses and individual wires that trigger each of the basic operations: bank precharge and activate, read and write. This simple interface makes a mode register or burst transfer logic unnecessary.

After taking the new design through logic synthesis, it was found that the complexity of the CMB controller did not increase – instead, it decreased slightly. This can be attributed to the fact that the interface is now much simpler and certain states having to do with the mode register management were removed. While no new layout was done, it is easy to estimate the new area and density numbers by just removing the SDRAM interface logic circuits from the layout. The resulting CMB design has an area of 8.45 mm^2, down from 13.87 mm^2, achieving an overall density of 240 Kb/mm^2, which is more representative of the real needs of our application.

Projection to an SRAM-based CMB

Projecting to a similar design based on SRAM allows us to gauge the benefit of embedded DRAM technology for our application, purely in terms of area.

From the data in Table 4.1, the average density of commodity SRAMs in a 0.25μm technology is about 41 Kb/mm^2. This number has been derived for complete chips, including pad ring and other peripheral circuitry. For the raw SRAM only (no pad ring or external interface circuitry, but including decoders, datapath and timing circuits), it is easy to arrive to a density of about 45-50 Kb/mm^2 based on published die photographs. The CMB logic will also be smaller in an SRAM-based CMB, primarily because there is no need for stall buffers and related circuits, since the latency of SRAM is constant. Removing these circuits would bring the area of the CMB logic (datapath plus control) down to 3.25 mm^2.

There are two relevant design points here:

1. The equal area design: within the 8.45 mm^2 of the "retargeted" DRAM-based design, one could fit 256 Kbits of SRAM, together with the simplified CMB logic block. This results in an average density of 30.3 Kbits/mm^2. In other words, there is an **8X** difference between the two designs. This difference is clearly smaller than the density ratio of the pure memories, but still quite significant.

2. The equal capacity design: the area required for 2 Mbits of SRAM and the CMB logic is about 46 mm^2, more than 5 times the size of the DRAM-based design. This is clearly unacceptable in most realistic cases.

Discussion

The following table summarizes the densities of the various implementation options. The density advantage of embedded DRAM over SRAM is quite obvious. What is interesting is the relatively low density of the DRAM in our design. Note how the DRAM density drops from 930 Kbits/mm^2 for the core to only 450 Kbits/mm^2 for the complete "raw" DRAM circuit (no interface controller). This drop is much more severe than for conventional DRAM designs. This can be attributed primarily to the small capacity (2 Mbits) of the memory macro and possibly, to some degree, the wide I/O (128 bits).

In the current memory macro design, the row and column decoders, fuse banks, datapath and some remaining unused area take up slightly more space than the DRAM core. Increasing the capacity of the macro in either dimension would amortize the area for the datapath and/or decoders over a larger number of bits. A 16 Mbit macro, consisting of 8 2-Mbit banks, is estimated to occupy an area of 25.1 mm^2 (still without the interface controller), yielding a density of about 650 Kbits/mm^2, much closer to commodity DRAM densities.

	initial DRAM	improved DRAM	SRAM
$\sim$ cell size, μm^2	0.6	0.6	11.0
Core density	930	930	—
Raw memory density	450	450	48
Macro density	230	450	48
CMB density	150	240	30

Table 4.4: Memory density, in Kbits/mm^2, of the different memory macro and CMB designs. SRAM data are for the equal area case.

Reducing the I/O width of the macro would obviously reduce the overhead for the I/O datapath – however, this would increase the required overhead for the column decoders. With a sufficiently narrow I/O, it would be required to switch back from the "wide I/O" floorplan to the conventional one (illustrated in Figure 2.1, page 16). That would occupy about the same area as the current floorplan, with the placement of the column decoders and the datapath circuits swapped. Therefore, the interface width of the memory macro has a rather minor effect in the overall area.

Still, as seen in Tables 4.1 and 4.4, the memory density of the Trumpet DRAM is much closer to that of commodity DRAM, with its narrow I/O and large capacity, than to the density of SRAMs. This can yield significant performance and energy benefits, as will be seen in the following chapters.

Chapter 5

Preliminary performance evaluation

In this chapter we examine how the large on-chip memory capacity made possible by embedded DRAM translates to performance. To this end, the total application execution time is decomposed to a number of contributing components and the effect of embedded DRAM technology to each of these is examined. The relative contribution of each component to the total execution time depends on the balance between logic and memory resources. A methodology for balancing such resources is the subject of Chapter 6. In this chapter we look at the contribution of each individual component, while in Chapter 6 the individual elements are combined.

We start by introducing the embedded and conventional architectures that will be the basis of the study in this chapter and Chapter 6.

5.1 Architecture definition

The architecture assumed for this study is very similar to the one introduced in Chapter 3, illustrated in more detail in Figure 5.1. An array consisting of compute and memory pages is coupled to an on-chip CPU and acts as a coprocessor, handling the more repetitive parts of a computation.

The CPU is assumed to be similar to the M32R/D microprocessor[113]: a simple processor core, with a modest amount of SRAM cache and its DRAM main memory inte-

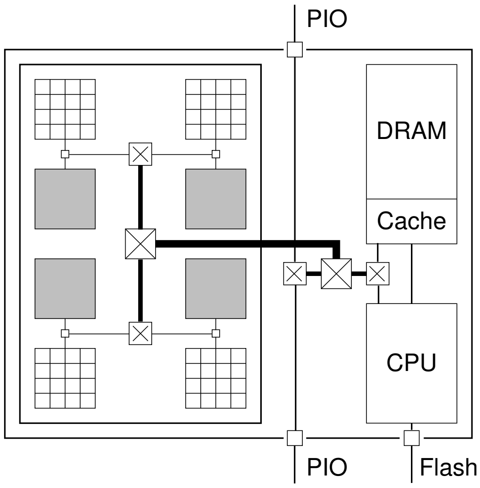

Figure 5.1: Single-chip architecture

grated on-chip. A narrow off-chip interface connects the processor to some external Flash or PROM memory, where the operating system and application software is resident. For small devices the processor will be a single-issue RISC core, while for larger ones it could be something more powerful, like a dual-issue processor. The CPU is used primarily to execute the run time system, the part of the operating system that manages the array: initiate page configurations and state swaps, setup streams between compute pages or between compute and memory pages, and handle I/O. In addition, it may be used to handle parts of the computation that do not map well to the array, or when it has spare cycles to contribute.

The memory page is similar to the Trumpet CMB. The clock rate is assumed to be 200 MHz, in line with the state of the art in modern DRAM macros[3]. Based on the experience from the Trumpet chip, such a clock frequency with a 0.25 μm logic design rule should be adequate for the controller logic. Unlike Trumpet, the user logic, memory pages and CPU are assumed to run at the same 200 MHz clock rate. The minimum latency of the memory page is then 4 clock cycles, corresponding to a 2-cycle latency of the actual DRAM macro and a register stage at the memory page interface, in both directions. The "raw" DRAM and pseudo-SRAM interface, as well as the input or output stream mode, are supported. The DRAM interface is assumed to support independent row and column command and address inputs, as discussed in Section 2.3, because of the simplicity and flexibility of this interface. DRAM macros with up to 512 bits of I/O are available[60]. For the study in this chapter we use a more conservative value of 256 bits, because very wide I/O requires either a large memory macro or a very high aspect ratio. In the next chapter the width of the memory interface will be a variable.

A system interface resides at the root of the array network. To support simple computations that can be performed in a fully spatial manner, a number of interfaces are supported, sharing the wide bus at the tree root. In this way, an input stream, some computation and an output stream can run concurrently. These interfaces can connect the array core to the processor memory, dedicated package pins for programmable high bandwidth I/O to and from peripherals, or directly to the CPU pipeline through a coprocessor-like interface. For programming convenience and consistency, the semantics of these interfaces are identical to those of the memory page: DRAM or SRAM interface, as well as the stream modes can be implemented. The run time system executing on the host processor is responsible for configuring each interface appropriately and for providing an initial address for the stream mode, when necessary.

Conventional reconfigurable systems today usually combine limited, fine-grain on-chip memory with higher capacity, “bulk” external memory. Most FPGAs include a number of SRAM banks, typically in the 2 Kbit to 4 Kbit range, often dual-ported. These small memories can be used for small lookup tables, FIFOs, register files and other small data structures. SRAM has been customarily employed as the external storage for bulk application data, probably as much for its ease of use, as for its high performance when compared to DRAM. However, external DRAM is also an acceptable option, offering higher capacity at lower cost. This on-chip/off-chip combination, with either external SRAM or DRAM as appropriate, will be the baseline “conventional” technology for the discussion in this chapter.

The overall structure of the external memory system is outlined in Figure 5.2. A logic chip contains the array, the CPU and its SRAM cache. Two external memories are connected to the logic chip via independent interfaces, so that stream-like computations that require concurrent read and write streams can be implemented efficiently. These memories are shared between the array and the CPU. The blocks labeled “CMB” arbitrate between CPU and array accesses, in addition to implementing the other memory page functionality. The available memory capacity has to accommodate all the CPU data, as well as the array configuration bitstreams, data and stream buffers. In the array the CMBs have been replaced by small 4 Kbit SRAM banks, with up to 32-bit wide interfaces. The interface logic for these SRAMs is much simpler than the CMB logic, implementing only the retiming and interconnect functions, with almost no control logic.

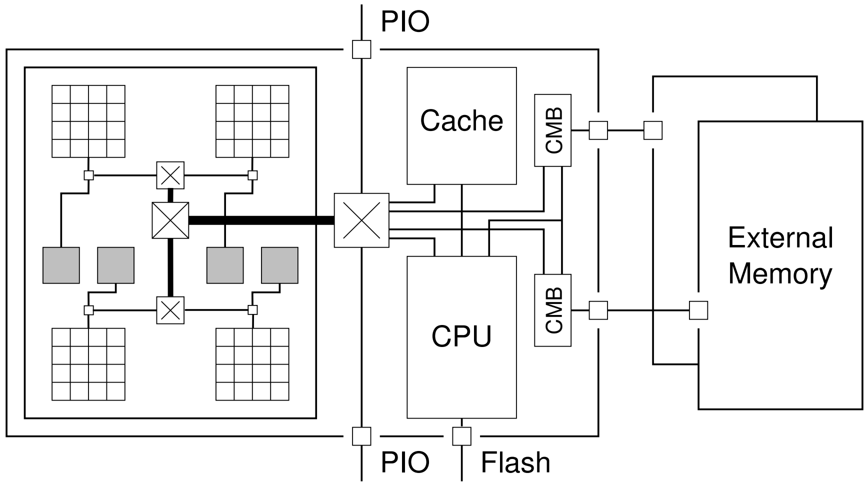

Figure 5.2: Architecture with external memory

The cycle time of the conventional device will be somewhat higher than that of the embedded memory one, reflecting the difference in logic performance between current embedded and logic processes. External memory and logic clocks will have to be related by some simple ratio. Luckily, a 2:1 relation between logic and DRAM clocks and 1:1 logic and external SRAM clocks lead to memory cycle times that are typical for current memories: 200 MHz or faster SRAMs and 100 MHz or faster DRAMs.

Table 5.1 summarizes the features of the embedded and conventional systems that are relevant to this discussion. Note that conventional systems are a combination of the second column (internal SRAM) and either the third or fourth (external DRAM/SRAM), while only the first column is relevant to embedded systems.

	Emb.DRAM	Int.SRAM	Ext.DRAM	Ext.SRAM
Memory pages	variable	variable	2	2
Logic clock	200 MHz	200+ MHz	200+ MHz	200+ MHz
Memory col.cycle	200 MHz	200+ MHz	100+ MHz	200+ MHz
Width	256b	1-32b	32b	32b
Precharge latency	3 cycles	—	2 cycles	—
Min. bank active time	4 cycles	—	5 cycles	—
RAS–CAS delay	4 cycles	—	3 cycles	—
Read latency (memory)	2 cycles	0 (flow through)	2 cycles	2 cycles
Read latency (CMB)	4 cycles	—	4 cycles	4 cycles

Table 5.1: Embedded and conventional memory system parameters

5.2 Breakdown of execution time

When a device is large enough to accommodate the logic circuits for the entire application, the configuration is static. When the available logic resources are not sufficient, we can use a time-multiplexed model: an application is broken down into a number of kernels that time share the available compute resources. Each kernel executes for a certain amount of time, processing a fraction of the input stream and buffering its outputs into a stream buffer, on-chip or off-chip. Kernels are scheduled according to a simple schedule, often round-robin. After all kernels have executed for a certain amount of time, the process repeats, processing the next chunk of input. In this way the computation is broken down into a number of epochs, with a number of kernels alternating within each epoch, as illustrated in Figure 5.3. The frequency at which kernels are swapped may depend on the available buffer space, the characteristics of the algorithm, or, in some real-time applications, on fixed timing requirements.

With the above model, the time to completion of an application, which is the figure of merit for this discussion, is equal to the total execution time of the various kernels, plus the total time for reconfiguration:

$$T_{app} = \sum_{1}^{N} T_{kernel} + \sum_{1}^{N} T_{cfg}$$

where N is the total number of kernel executions, or reconfigurations, equal to the number of kernels per epoch, times the number of epochs.

Unlike the traditional uses of FPGAs where the clock cycle time is set by the application, in this architecture the cycle time is fixed, determined by the designer of the FPGA. The same clock is used for reconfiguration. Then, each of the above time components, T_{kernel} and T_{cfg}, can be computed as the corresponding number of cycles, times the clock cycle time: $T_i = C_i \times T_{clk}$. Therefore,

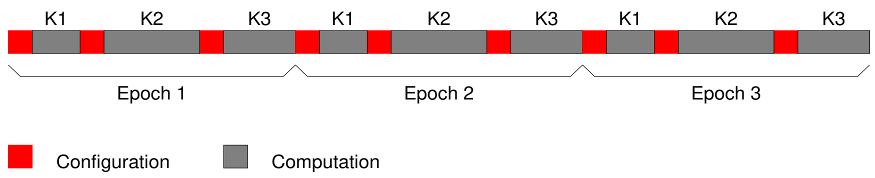

Figure 5.3: Computation epochs and kernels

$$T_{app} = [\sum_{1}^{N} C_{kernel} + \sum_{1}^{N} C_{cfg}] \times T_{clk}$$

The use of embedded DRAM technology can affect all of the above parameters. In the following we review the effect on each parameter individually: the clock cycle time, the number of cycles for kernel execution and configuration, and the number of reconfigurations.

5.3 Clock cycle time

A disadvantage of embedded DRAM technology is the somewhat inferior transistor performance in DRAM processes, which can impact the cycle time of logic circuits. DRAM processes have been optimized for dense memory at low cost and large data retention time, at the expense of transistor speed and interconnect density [56, 65]. Embedded DRAM processes that are derived from their pure DRAM counterparts usually inherit this low performance transistor. On the other hand, logic-based processes may have inferior DRAM in terms of density, performance and/or retention time.

This performance handicap of most current embedded DRAM processes can be attributed to a number of technical and cost issues. In the standard DRAM process gate oxides are made thicker, because of the boosted word line level. Logic transistors outside of the DRAM array can be fabricated with a thinner gate oxide, but this dual oxide structure incurs a significant cost premium, so it has been most commonly avoided to date. The self-aligned silicide (or salicide) layer commonly used in logic processes to reduce the source/drain resistance can cause node leakage in the memory cell and so it is usually left out. Again, it is possible to apply the salicide process to the logic circuits only, but this also adds to the fabrication cost. Channel length control is made difficult by the “capped” gate and borderless contact process used in most DRAMs (for example [70]) to achieve a smaller cell, because it increases the height of the gate stack. To maintain an acceptable $\mathrm{L}_{eff,min}$, DRAM processes typically use a longer *nominal* L_{eff} than logic processes of the same lithography. All the above factors contribute to a reduced performance.

Stack capacitor technology further impacts logic performance and density because the capacitor is fabricated after the transistors. Common processes require significant heat cycles for the fabrication of the capacitor, which makes it difficult to maintain shallow source/drain junctions. Deeper junctions result in higher parasitic capacitance and corresponding delay and energy penalties. In addition, the tall capacitor stack results in a nonplanar chip surface, which makes dense wiring difficult. Finally, multiple layers of interconnect are usually avoided for cost reasons.

Recently, embedded DRAM processes have been developed, in an effort to bridge the gap between existing logic and DRAM. These processes incorporate features of both their logic and DRAM counterparts to achieve dense, high performance logic at the same time as dense DRAM. In the simplest form, an embedded DRAM process is based on a standard DRAM recipe, with a number of metal layers added. These additional layers improve logic density, but do not help very much with logic performance, aside from a slight reduction in wiring parasitics. In more advanced processes, steps are added that aim at improving the transistor speed: multiple transistor thresholds, dual gate oxide thicknesses, a salicide layer for the logic transistors and shallow junctions have appeared in some of the most advanced processes. Trench capacitor based processes are generally more appropriate for embedded DRAM, because of the planar chip surface and the reduced thermal budget after fabrication of the transistors.

Modern embedded DRAM processes support logic with only a minor performance degradation. Typical commercial products available today include ASIC lines from IBM[57] and Toshiba[126] that support embedded DRAM macros. As seen in Figure 5.4, the logic performance of Toshiba's embedded DRAM ASIC line is less than a generation behind the logic-only line: embedded DRAM logic performance lags by about 15%. IBM's embedded DRAM line, the SA-27E, has the same performance as the standard logic process, SA-27, at 33 ps for a NAND2 gate with fanout=2. Note however that the embedded process is based on slightly finer lithography, 0.15μm as opposed to 0.16μm, which should give it an approximate 10% performance advantage. Finally, TSMC is offering a logic-based embedded DRAM process, currently at 0.25μm lithography, which is claimed to offer performance identical to the corresponding logic-only process[128].

The disadvantage of these processes is certainly the increased processing cost. Unfortunately, no pricing data are widely available, so no reliable conclusion can be drawn. Estimates for the cost overhead over conventional logic or DRAM processes fall in the range of 20-40% per wafer[49, 65]. In addition, the logic base of the embedded process may penalize the DRAM density. This is the case for example in IBM's process, where the

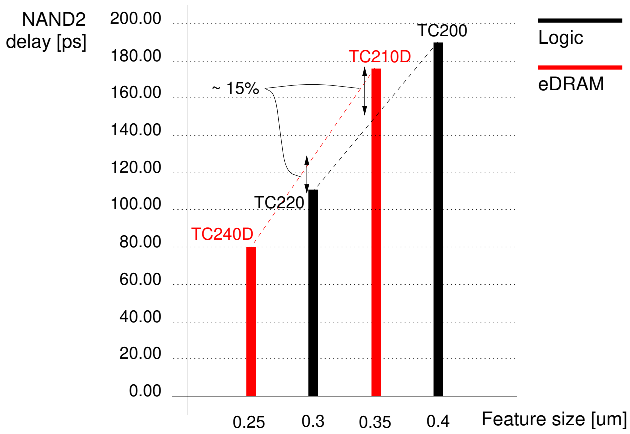

Figure 5.4: Toshiba ASIC logic performance (nominal NAND-2 fanout=5 propagation delay)

memory density lags by about a generation and a half.

These performance data suggest that in a modern embedded DRAM process, because of the somewhat inferior logic performance, the clock cycle may be inflated by a factor of up to 15%. This partly offsets the benefits arising from reducing the number of cycles required to execute a compute kernel or a reconfiguration. However, this cycle penalty is rather modest.

5.4 Kernel execution time

When external memory is combined with small internal SRAM banks, like the ones available in most FPGAs, the result is a 2-level memory hierarchy. Embedded DRAM, with the faster sense amplifiers and the slower DRAM core, is a very similar hierarchy – the main difference being that the link to the second level (the core) is very wide but rather slow. The usage patterns for these two levels of memory hierarchy determine the relative performance of these two technologies.

The first level in the "conventional" hierarchy, small internal SRAM banks, is most commonly used for small data structures, like lookup tables and FIFOs. With embedded DRAM, the sense amplifiers can be used for the same purpose. Still, their latency will be longer than that of the conventional SRAM banks. This latency is not always an issue, as many algorithms have a feedforward structure, where accesses to the memory can be pipelined, hiding the latency in this way. However, when there are dependencies between successive accesses, for example when traversing a pointer-based data structure, or when accesses to the memory are bidirectional (mixed reads and writes), this latency is exposed. In addition, SRAM banks in modern FPGAs are often dual-ported, a feature that is difficult to implement with DRAM designs.

For sequential accesses to larger data structures, the first level of the memory hierarchy is not very effective. In that case, the application throughput depends on the bandwidth into the second level of hierarchy – then embedded DRAM has a clear advantage.

Finally, for random accesses to large data sets, the low latency of external SRAM is a clear win. The wide interface of the DRAM core is usually not of any use: the random nature of accesses makes it unlikely that successive accesses will fall within the same memory word. Then, memory accesses will have to proceed at the natural data width, often as low as 8 bits. In this case, the fast narrow external SRAM interface is much better than the wide but slow interface of the DRAM core.

When a number of DRAM banks are available on chip, another important advantage arises: multiple independent address streams can be supported without contention. This can be very useful in compute kernels that manipulate a number of data streams. Distributed internal SRAM can support a similar number of concurrent accesses, but only to limited amounts of data, such as small tables for filter or transform coefficients. Larger data sets will have to be stored off-chip, in which case contention on the external interface will often result. Because of pin limitations, it is difficult to support a significant number of independent external memory interfaces. Existing systems commonly support one or two external memory interfaces, which will be insufficient for many compute kernels.

Because of the slow DRAM core, embedded DRAM is not always a good choice. A number of cases that demonstrate the effects discussed above are presented in the following. For a comparative evaluation, typical timing parameters for SRAM and DRAM are assumed, as listed in Table 5.1.

Best and worst case for a single page

The best case for a single embedded DRAM page is a sequential unidirectional (read-only or write-only) access pattern, where data parallelism is possible. Such examples are basic image processing operations, like chroma keying or gamma correction, when the source and result images are stored in different memories. Also, a simple search within a data set comes down to a single read-only access stream. Even though data types may be narrow (e.g., 8-bit pixels) the data parallel nature of such algorithms allows the entire width of the interface to be utilized, assuming that enough logic resources are also available to handle the data stream. In addition, the sequential nature of the stream allows prefetching to hide the row access time. In such a situation, embedded DRAM offers a throughput advantage equal to the bandwidth ratio between internal and external memory. *8X and 16X speedups* are achieved against external SRAM and DRAM, respectively, by our example embedded DRAM system with the parameters of Table 5.1.

In the worst case, a completely random, narrow, read-only access stream into an on-chip DRAM will proceed at the memory row cycle rate, 7 clock cycles long with the parameters of Table 5.1. The random accesses do not allow any data parallelism, so the large width of the interface is wasted. An external DRAM suffers from the same problem, with the additional handicap of a cycle rate half that of the logic clock. An external SRAM could support the same access stream at an access per cycle at the same cycle as the logic clock. Therefore, a *2X speedup* is achieved against external DRAM, but a *7X slowdown* is sustained against external SRAM.

Table lookup: Quantization

In many quantization kernels, as in the JPEG image compression standard[134], small lookup tables are used to hold a number of quantization coefficients – 64 in the case of JPEG. With 16 bits per coefficient, a table can easily fit in a DRAM row or a small on-chip SRAM. Even though the access pattern into the table may be random, the locality of reference is such that no row accesses need to be performed in the DRAM case. On the down side, however, the randomness does not allow the wide interface of the DRAM to be fully utilized. Also, because of the feedforward character of the computation involved, the increased latency of the DRAM will not be an issue. In that case, the throughput of the kernel, in term of clock cycles per datum, will be the same, independent of the memory architecture.

When multiple memory pages or distributed SRAM banks are available, parallel implementations of the kernel may be possible. In that case, most devices are likely to be limited by compute resources before they run out of distributed memory bandwidth.

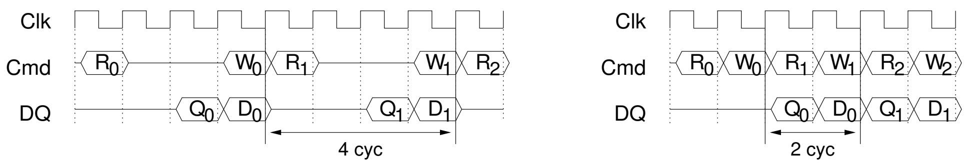

Figure 5.5: Embedded DRAM FIFO timing. Standard (left) and delayed-write (right) implementations.

Read/write streams: Small FIFO buffer

A FIFO buffer is a very common data structure, that can be implemented in the first level of memory hierarchy (DRAM sense amplifiers or distributed SRAMs), assuming that it is not too deep. With the alternating read/write access stream, the low latency of internal SRAM and the possibility of dual ports give it a definite advantage.

Because of the common 2-cycle column latency of DRAM, a basic embedded DRAM implementation will be limited to a 4-cycle period: Read – Wait – Receive – Write, as shown in Figure 5.5. Significant throughput gains can be made if a delayed write scheme is implemented, as described in Section 2.1. With such a design, the period can be reduced to two cycles. A simple single-ported on-chip SRAM, with a sub-cycle latency, will easily achieve a 2-cycle period. Finally, a dual-ported on-chip SRAM will allow a single-cycle repeat rate. An average *2X slowdown* is sustained in the embedded DRAM case.

It should be noted that since the access pattern is sequential, the large width of embedded DRAM can be used to compensate for the limited performance. Data words can be grouped in pairs, so two words are read and written in every period. This scheme however requires some additional logic resources to skew and multiplex the appropriate data words. Then, the throughput of the FIFO can be doubled and reach internal SRAM levels. This approach is appropriate when there is a surplus of logic resources.

Parallel accesses to large data

It is often necessary to sequentially scan a large amount of data that may span multiple memory pages. If the associated computation is simple and there are no sequential dependencies in the computation, there is a very large potential for parallelism, enabled by the multiple wide memory interfaces and the modest requirements on logic resources. Such cases include searching, simple filters, chroma-keying of images, and other basic operations. External memory implementations, both SRAM and DRAM, will be limited by the small total memory bandwidth. The speedup achieved in this case is proportional to the number of memory pages.

Random accesses to large data: graph manipulation

Many CAD tools manipulate large graphs, used to represent a logic circuit in various levels of abstraction: boolean logic, schematics, layout, and so on. Because in the general case such graphs may be very irregular, the locality in the reference stream is often very poor. In addition, even when multiple pages are available and the algorithm can be parallelized, the randomness of the access stream will often result in memory access conflicts. Then, the potential speedup due to parallelism is limited. In such cases, external SRAM allows the best possible performance, since one access per cycle (or one read-write per 4 cycles) can be supported. DRAM implementations, both external and embedded, will be limited by the DRAM row cycle time, while the existence of multiple memory pages may not help performance.

Summary

Table 5.2 summarizes the relative performance of the kernels discussed for the two architectures. Although the choice of kernels is rather arbitrary, it spans the entire range from worst to best case. Compared to the case of external DRAM, the embedded technology offers a speedup in almost all cases. The only slowdown is in the case of the FIFO, which in the conventional technology takes advantage of the distributed internal SRAMs. Compared to external SRAM, there is a very large variation of individual kernel results, from 7 times slowdown to $4 \times N$ times speedup, which makes generalization rather unreliable. Performance projections against external SRAM will be more meaningful when it is possible to narrow down the range of applications to a set with common characteristics – for example video processing, speech coding, or networking. It should be noted that the increased performance often possible with external SRAM comes at the cost of much more expensive memory, about 15 times more expensive per bit than DRAM, as discussed in Appendix B.

Another factor that affects the kernel performance somewhat is the refresh overhead in the case of embedded DRAM. Whereas reconfiguration events are fast enough to be completed without refresh interruptions, in the order of a few microseconds, many compute

	eDRAM	Ext.Dram	Speedup	Ext.SRAM	Speedup
Sequential (1 pg)	1/8	2	16	1	8
Random (1 pg)	7	14	2	1	0.143
Quantization	1	1	1	1	1
Small FIFO	2-4	1-2	0.5	1-2	0.5
Parallel (N pg)	1/(8xN)	1	8 x N	1/2	4 x N
Graph (R/O)	7	14	2	1	0.143
Graph (R/W)	8	16	2	4	0.5
Geom.Mean (N=4 pg)			2.97		0.94

Table 5.2: Summary of kernel performance results for embedded DRAM and conventional technologies. Performance is given as number of cycles per input datum. The middle column gives the speedup of embedded DRAM technology over external DRAM; the right column gives the speedup over external SRAM, a much more expensive conventional technology. The external DRAM implementation suffers an additional slowdown of 2X, because of the difference in logic and DRAM clocks. For the N-page sequential case both external memories are used. Also note the factor of 8 in some cases, that results from the different width of the internal and external interfaces.

kernels will have to be interrupted frequently, to allow DRAM refresh. However, this overhead is quite small: even with a conservative 8 ms retention time estimate, an embedded DRAM running at 200 MHz, a minimum row cycle of 35ns (7 clock cycles) and 2K refresh organization, will suffer less than 1% of refresh overhead. Therefore, this will be neglected from now on.

The bottom line is that the value of embedded DRAM for speedup of individual kernels is very application dependent and somewhat questionable for a general purpose device. Impressive speedups can be achieved in some cases, but significant slowdown is also possible. We see in the following that we can expect more consistent benefits when it comes to reconfiguration.

5.5 Reconfiguration time

During reconfiguration, accesses to the configuration bitstream are sequential, so prefetching is possible and misses can be completely avoided. Furthermore, the active configuration store can usually be organized as a fairly wide memory. Then the maximum width of the memory interface can be utilized. These properties allow the peak bandwidth of embedded DRAM macros to be achieved. In addition, when the reconfigurable fabric is organized in independent subarrays, or pages, it is often possible to reconfigure many or all of these subarrays in parallel, assuming enough on-chip memories are available. Such is the case for example in the Trumpet architecture, where the reconfigurable fabric is organized in pairs of compute and memory pages. In this architecture, all compute pages can be reconfigured from their nearby memories concurrently, so the total reconfiguration time is just the size of the configuration bitstream of a single compute page[1] divided by the bandwidth of a memory page. The corresponding aggregate memory bandwidth is the combined bandwidth of all memory pages and scales as the number of pages on the device.

When the configuration bitstream comes from external memory, the rate is limited by the much lower external memory bandwidth. But more importantly, the configuration of multiple pages has to be serialized. Then, the total chip reconfiguration time increases for larger devices.

In the Trumpet design, with a 125 MHz, 128-bit wide memory macro, a 2,000 MByte/sec configuration bandwidth is available[2]. With a total configuration size of 9 KBytes for a compute and a memory page, Trumpet can be reconfigured from its internal memory in 4.5 μsec. Configuration is also possible over the general purpose external interface, which is 32 bits wide and runs at 31.25 MHz, an eighth of the internal clock frequency. In this case, reconfiguration will take 16 times as long, or 72 μsec. For larger Trumpet-like devices, the difference is even more striking: an 8-page version of Trumpet would still achieve the same 4.5 μsec internal reconfiguration time, while the time for off-chip reconfiguration would increase by another factor of 8, to 576 μsec, 128 times longer than the internal reconfiguration.

Projecting to the parameters defined in Section 5.1, more typical of today's state of the art, on-chip and off-chip configuration bandwidths differ by a factor of about 4 to 6. Note that it is assumed that both the external interfaces can be used in parallel for reconfiguration. This difference in bandwidth is identical to the difference in configuration time for single-page devices, while for larger devices the same parallelism effect will apply as for Trumpet. Table 5.3 summarizes the differences in configuration times for Trumpet and the systems defined in Section 5.1.

[1] Or, as in the case of Trumpet, a pair of compute and memory pages

[2] Because of the longer than anticipated clock cycle, half of that was achieved in the implementation.

	Int.DRAM	Ext.SRAM		Ext.DRAM	
Trumpet (1 pg)	4.50	72.00	(16X)	72.00	(16X)
Trumpet (8 pg)	4.50	576.00	(128X)	576.00	(128X)
Projection (1 pg)	1.28	5.12	(4X)	7.52	(5.9X)
Projection (8 pg)	1.28	40.96	(32X)	60.16	(47.1X)

Table 5.3: Reconfiguration times, in microseconds

5.6 Number of reconfigurations

Referring to Figure 5.3 (page 74), the total number of reconfigurations is simply the number of computation epochs times the number of kernels that alternate within each epoch. So we can rewrite the execution time as

$$T_{app} = \sum_{1}^{E} \sum_{1}^{K} C_{kernel} \times T_{clk} + \sum_{1}^{E} \sum_{1}^{K} C_{cfg} \times T_{clk}$$

where E and K are the number of computation epochs and the number of kernels within each epoch respectively. The first term in this equation, the total kernel run time, is independent of the number of epochs, E, as fewer epochs mean fewer but longer kernel runs. It does depend, however, on the number of kernels within each epoch, K – simply, if with more available logic resources we can combine small kernels into fewer, larger ones, the same work will be done in fewer cycles. The second term, the total reconfiguration time, depends on both E and K, since we have to reconfigure $K \times E$ times total.

The number of kernels, K, depends primarily on the amount of logic resources, since more available gates make it possible to implement a larger part of the computation in one piece. The number of epochs, E, usually depends on the available buffer space, since more space allows each kernel to process a larger fraction of the input before it is swapped out. However, for some applications, sequential dependencies or real-time latency constraints may require kernels to be swapped before the stream buffer is full, in which case the available memory capacity may remain underutilized.

Single chip and multichip systems of comparable cost will have different amounts of logic and memory resources available, so the number of reconfigurations required will vary accordingly. Because large dies have lower yields, it may be more economical to split memory and logic resources into different chips. In that case, the silicon cost savings for the multichip systems may make more memory or more logic resources available, leading to fewer reconfigurations. In the following I make a simple calculation that will provide an upper bound on this additional amount of resources. It should be pointed out that the following calculations are made under the assumption that the silicon cost dominates, so the cost of external parts can be neglected. As we will see in Chapter 6, for low cost systems the cost of external parts plays a very important role and usually tips the balance in favor of the single-chip, embedded memory devices.

Consider an embedded DRAM device with an approximate 50-50 split of the available area into memory and logic resources. Assuming an organization similar to Trumpet, the memory resources will consist of a number of memory pages. Within each page, about half the area will be consumed by the interface functions. The conventional system will consist of a logic-only chip and *two* external memories. This system will include just two memory page interfaces, which are neglected for simplicity. Also, we will assume a 40% additional wafer cost for the embedded process[49], 33% denser logic in the logic-only process, and a cubic dependence on die cost to area[51]. We will assume large chips, so that the silicon cost dominates and costs for packaging, circuit board, etc will be neglected. All these approximations give a benefit to the multichip system. On one extreme, the multichip system will have the same amount of logic (in 75% of the area) as the embedded chip and additional memory. On the other, it will have the same amount of memory and additional logic. Applying the equal cost constraint we can easily calculate:

- Logic rich: $L_x \approx 6 \times L_e$, $M_x = M_e$
- Memory rich: $L_x = L_e$, $M_x \approx 7 \times M_e$

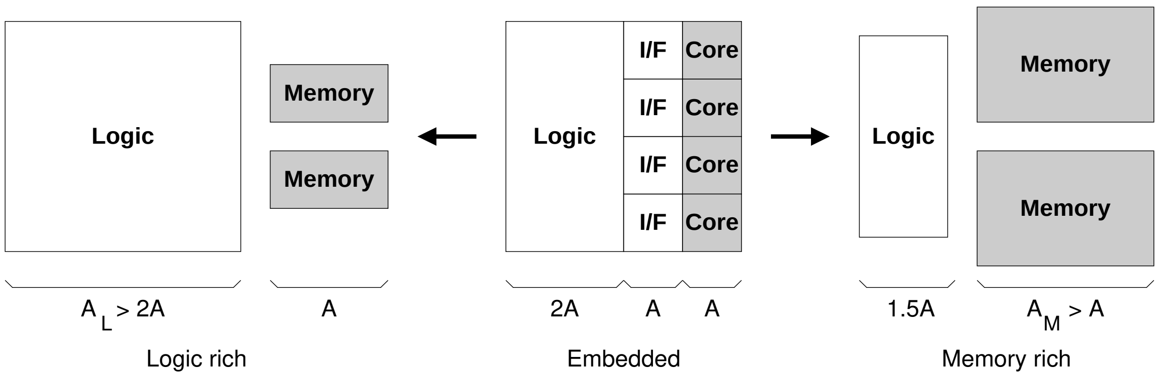

Figure 5.6: Area comparison of embedded and multichip systems

where M and L represent the total memory and logic capacity and the subscripts x and e refer to the external and embedded memory cases respectively. Furthermore, part of the memory capacity will be taken by configuration and user data storage. In Chapter 6 we find that on average about a third of the memory of the embedded device is taken up by these tasks, leaving $2/3 \times M_e$ available for stream buffers. Neglecting this configuration and data storage in front of the much larger external memory capacity, we conclude that in the memory-rich case

- Memory rich: $L_x = L_e$, $B_x \approx 10.5 \times B_e$

where B is the total buffer space available.

We see that in the memory-rich case we may have up to about 10 times as much buffer space, which may reduce the number of epochs, E, accordingly. In the logic-rich case, we have 6 times as much logic capacity, possibly reducing the number of individual kernels by up to 6 times. In practice the optimal allocation of memory and compute resources will be somewhere in the middle. It should be noted that the limited selection of external memory components will only allow a small number of discrete valid points along the memory-logic tradeoff. This is not the case for the embedded system, where the width and depth of on-chip memory can be varied at a much finer grain. This reduction of the number of reconfigurations, possibly by as much as 10 times, will have a direct effect on the reconfiguration overhead. In the memory-rich case, the total kernel execution time will not change, since larger stream buffers result in fewer but longer kernel runs. In the logic-rich case, however, the additional logic resources will have a direct effect on the total kernel execution time.

It should be stressed again that the above numbers represent a loose upper bound on the extra logic or memory capacity available for the multichip system, under the assumption of large dies. When we work with smaller, low cost dies, the additional cost due to packages and the larger form factor will outweigh the silicon cost savings, limiting the total area budget for the multichip system, some times to less than the area of the single embedded chip. As a more rigorous data point, in Chapter 6 we find a multichip system with about 2.8 times more memory and 1.6 times more logic than the embedded one. This outperforms the embedded chip in some applications.

5.7 Conclusions

By now it is clear that the application of embedded memory technology has widely varying effects on the components of the total application execution time. On one hand, there is a slight reduction on the clock frequency and a likely increase in the number of reconfigurations. On the other, there is a large reduction in the reconfiguration time. Finally, the effect of the technology on the throughput of individual kernels may vary within a large range, from a significant slowdown to a large speedup. It is therefore very difficult to generalize from that and draw general conclusions regarding the value of embedded DRAM technology in our environment. The resulting performance will depend largely on the application domain, the balance of compute and memory resources, as well as the total silicon real estate available. For certain applications, embedded DRAM will not be appropriate, while for others it will enable large performance gains.

For fixed-function devices, such as ASICs, it is easy to tell whether embedded DRAM has a positive or negative effect on performance. With no reconfiguration and the entire algorithm implemented as a single "kernel", one only needs to characterize the access patterns of the given application and compare the effect of the technology on the kernel throughput against the effect on the clock frequency. For programmable devices, like FPGAs, one will still need to narrow down the application domain, profile a number of representative applications and estimate the *expected* performance. Chapter 6 of this thesis presents such a study.

To get more consistent performance from embedded DRAM designs, certain improvements on the basic architecture could be made. First of all, use of on-chip SRAM to some degree would often be beneficial. Small lookup tables, like filter coefficients, can easily be implemented in the fine grain SRAM of LUTs. On the other hand, when read/write access is required, as in the case of the histogram kernel, some kind of user-writable RAM is necessary. The most common approaches taken by FPGA vendors today are embedding of dedicated SRAM banks in the design and LUT bits that can be directly written by the user logic. Both approaches have their advantages and disadvantages. The former approach offers denser memory; on the other hand, in a design with many heterogeneous resources (LUTs, SRAM, DRAM, interconnect) few user designs will actually manage to fully utilize all types of resources, so some underutilization of the hardware will be unavoidable in most cases. Other approaches are also possible for architectures with special features. For example, in Trumpet the long retiming register chains can be used efficiently for FIFO buffers. Unfortunately, this does not allow random accesses, so it is more limited than the above approaches.

Another potential improvement over standard practice is the integration of more SRAM bits into the DRAM design itself. This can be achieved in a number of ways: wider pages, more memory banks, or more row buffers per bank. However, this approach does not compare favorably to the other kinds of on-chip SRAM. The latency is still limited to typical DRAM latencies, usually 2 cycles, mainly because of the required form factor: wide rows that pitch-match the DRAM core, as opposed to more or less square blocks with no limitations on cell size. This latency will have a prominent effect when alternating reads and writes are required, or when data dependencies exist between successive read accesses. Still, more SRAM cache allows a larger number of applications to avoid the penalty of misses to the DRAM core.

Finally, to mitigate the effect of the read latency on alternating read-write streams, delayed write schemes are very useful. Looking back at the FIFO example, even when the array fits in the sense amplifier banks the DRAM read latency limits the throughput to a four-cycle repeat rate. This can be easily brought down to two when a delayed write scheme is applied. This feature has already been applied to many commercial SRAMs, but has been overlooked in the case of DRAMs. The primary application of DRAMs as main memory, hidden behind SRAM caches, may have contributed to that, since the effect of read-write gaps in this case is not so pronounced. In embedded DRAM systems, where the DRAM is accessed directly by the logic, the delayed write scheme may hold the same value that it currently does for SRAMs.

A reasonable set of incremental improvements over the basic architecture discussed in this chapter might consist of: i) implementation of a delayed-write scheme at the DRAM interface and ii) an increased number of DRAM banks. With these techniques, the negative kernel results in Table 5.2 can be largely mitigated: larger data structures will fit in the DRAM sense amplifiers, so the completely random case will be more rare; even with data structures that do not fit in the sense amplifiers, the miss rate will be reduced, and the performance of kernels like the graph manipulation will improve; read-write accesses to small data structures, like the FIFO kernel, which achieves the poorest performance in our set, will approach that of the conventional implementation; finally, read-write graph manipulation will benefit both from a shorter read-write cycle and a reduced miss rate.

Therefore, more consistent performance benefits can be achieved. User-writable LUTs in the configurable logic is also an option if delayed write is not implemented. It would allow data structures like FIFOs to be implemented in the user logic, taking advantage of the high performance of the fine-grain SRAM in the LUTs.

Chapter 6

Resource balance optimization

In this chapter I present a methodology for choosing an optimum design point for a Trumpet-like device, and results derived by applying this methodology. A *design point* is defined as a set of architecture parameters that optimize a certain performance-related cost function over a set of benchmarks. The same methodology, somewhat modified, can be applied to conventional multi-chip architectures, yielding a similar "optimal" conventional design. This approach lends itself to a direct performance comparison between the embedded- and the external-memory architectures.

A number of metrics exist to quantify the value of a given design. The most common are briefly reviewed here:

- **Performance:** This is the most common metric, expressed in a number of ways. Ultimately, the total time to execute a set of applications of interest to the user is what matters. Execution time can be measured as absolute "wall clock" time, or, with a fixed clock cycle time, as a number of cycles.

  In most architectures it is possible to increase the performance of a design by using more silicon area – for example multiple execution pipelines, or larger caches. This is particularly true for "scalable" architectures, such as FPGAs. Usually an instance of an architecture will be designed under a fixed die area budget, determined by cost estimates and pricing goals. Therefore, performance of a scalable design under the constraint of an upper bound on the die area is more realistic.

- **Cost–performance:** This is the most appropriate metric for characterizing the efficiency of alternative functions and has been used extensively both in academic research and in the industry. Typically, cost–performance is expressed as (die area) × (run time). However, the dependence of chip cost to die area is not linear, while other factors, like testing and packaging, contribute to the cost of the finished product. A more complete cost metric is used here, detailed in Appendix B.

• **Cost–performance–squared:** This metric, usually computed as (die area) × (run time)^2, has been used in a number of studies, as a means to overcome a problem of the previous metric: if cost and performance are given equal emphasis, designs that yield very poor performance at a very low cost may result. The cost-performance-squared metric puts an emphasis on the performance, avoiding such "pathological" design points. Because the cost-performance metric has a more natural interpretation and, at least for this study, it did not lead to such pathological points, cost-performance-squared is not used here.

• **Power–related metrics:** For mobile applications, power dissipation may be as important as performance, if not more. Power has been integrated in cost functions in many ways: by itself, often given a fixed performance; as (power) × (run time), which corresponds to a "energy per operation" metric; and even as (power) × (run time) × (area), as a composite figure of merit. Because power dissipation is difficult to quantify, it is not used in this study. With an elaborate energy dissipation model, derivation of appropriate "energy-sensitive" optima might be an interesting followup study.

6.1 Methodology

With the two alternative metrics, performance under a limited area budget and cost-performance, the problem can be stated concisely in two ways:

For performance: *Given a parameterized model of the architecture, a die area constraint and a set of benchmarks, select the device configuration that minimizes the total benchmark run time.*

For cost-performance: *Given a parameterized model of the architecture, a cost model and a set of benchmarks, select the device configuration that minimizes the product of the total benchmark run time and the device cost.*

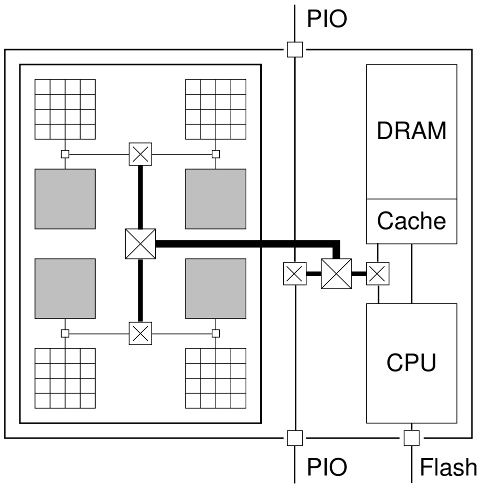

Figure 6.1: Single-chip architecture

Fixed architectural parameters

The overall architecture has been introduced in Figure 5.1, which is repeated in Figure 6.1 for reference. The architecture of the compute page is identical to that of the HSRA, with the exception that somewhat more modest wiring resources are used. Reduced wiring has an effect on the compute page and network area, as well as the latencies through the network. The same pipelined network architecture is used, with registered switch stages and retiming register-rich BLBs, but the smaller physical dimensions of the array, as well as the somewhat more modest cycle time goal, reduce the number of pipeline stages in the path of most signals. Both memory and logic clocks are assumed to be 200 MHz. Memory pages support the DRAM and pseudo-SRAM interfaces, as well as read/write stream accesses.

0.18 μm and 0.25 μm design rules have been assumed for the memory and the logic, respectively. Because of the regular layout of the DRAM core and the much less regular layout of common logic circuits, it is likely that in the near future logic design rules will lag DRAM design rules somewhat in most process technologies.

Variables

An instance of the architecture can be defined with the following 5 parameters that capture the relative balance of compute resources, memory capacity and memory bandwidth:

- P: Number of pages, or, more precisely, pairs of compute and memory pages.
- N_{BLB}: Number of BLBs per page. The BLB is assumed to be the dual 4-LUT logic block used in the Trumpet compute page. The total amount of compute resources is the number of pages times the number of BLBs per page.
- S_{CMB}: Memory capacity per page. The total available memory is the number of pages times the memory page capacity. On-chip memory is used for configuration storage, application data and stream buffers.
- W_{cfg}: Configuration bandwidth. For a fixed clock rate, the bandwidth is just the memory bus width. The entire bus is assumed to be linked to the configuration chain of the compute and memory pages.
- W_{app}: Application bandwidth. This is less than or equal to the configuration bandwidth. This distinction is important, because the cost as well as the benefit of bandwidth for configuration and applications are very different. Providing high bandwidth to the configuration chain is relatively cheap and has a direct impact on the configuration time. On the other hand, providing high bandwidth to the applications is much more expensive, because of the crossbar and retiming used to accommodate the routing requirements of applications and the stall buffers used to handle the variability of the DRAM latency. In addition, the benefit varies, depending on the application characteristics. Therefore, it is assumed that if W_{cfg} data wires come out of the memory, they are compressed by the data crossbar down to a set of W_{app} wires at the CMB interface. W_{app} is allowed to range from 1 byte up to W_{cfg}.

Note the role of parameter P, the number of pages: since the amount of computation is $P \times N_{BLB}$ and the amount of memory is $P \times S_{CMB}$, P may appear to be redundant. However, the aggregate configuration and application bandwidth is also proportional to P. Very high aggregate bandwidth would be more difficult to achieve with fewer larger banks, because the aspect ratio of the memory core would have to be very high. In addition, the independence of ports into each memory page presents many more opportunities for parallelism. We will see later that such parallelism can result in large speedups. Therefore, P captures an important aspect of the design, the *granularity* of mixing logic resources and memory. The benefits that result from fine-grain sharing come at the expense of increased per-page area overhead.

Procedure

To apply the area constraint, a model for the area of each major component has been developed that returns the chip area as a function of the five parameters. Details on the area models for the compute page, memory page, and network are given in Section 6.2.

The cost model, detailed in Appendix B, considers the factors that contribute to the overall system cost: die, testing, package and circuit board costs, and where applicable, the cost of external memories. Although a complete system consists of many other components, this model captures the contribution of the processing logic and associated memory.

To evaluate the total execution time, a performance model is developed for each benchmark individually. In general, applications are assumed to conform to the stream model, with compute kernels time-sharing the compute fabric and intermediate results buffered in stream buffers. It is assumed that when stream buffers fill up, streams can be redirected quickly, so applications can take advantage of all the available memory pages for stream buffering. Switching from one memory page to another is assumed transparent to the application and the time to switch is neglected.

The application execution model has been introduced in Chapter 5. As discussed there, the total execution time is the sum of the configuration time plus the compute time:

$$T_{app} = [\sum_{1}^{N} C_{kernel} + \sum_{1}^{N} C_{cfg}] \times T_{clk}$$

where N is the number of epochs times the number of kernels. The number of epochs depends primarily on the available stream buffer space, since larger stream buffers allow more of the input to be processed in one epoch. The number of kernels within each epoch depends on the diversity of functions in the application and the amount of compute resources. The duration of each reconfiguration depends on the number of BLBs per compute page and the configuration bandwidth, while the duration of each compute kernel depends on the amount of BLBs, the application memory bandwidth and the *memory access patterns* of the executing kernel.

The configuration time is easily computed as the number of epochs times the number of kernels per epoch times the time for one reconfiguration. Estimation of the computation time is more involved and requires use of application-specific models. These compute time models are discussed in Appendix C.

Parameter	Min	Max	Step
P	4	32	(irregular)
N_{BLB}	32	512	$\times$ 2
S_{CMB}	256Kb	16Mb	+ 256Kb
W_{cfg}	1 Byte	16 Bytes	$\times$ 2
W_{app}	1 Byte	W_{cfg}	$\times$ 2

Table 6.1: Parameter range and step

The search for the optimum is a combination of a simple exhaustive search of the entire design space and a second "fine-tuning" step. This procedure will be described in detail in Section 6.4, after a closer look at the search space. The search space is small enough that a full exhaustive search only takes a few seconds on a Sun Ultra-II workstation. The range of the five parameters is given in Table 6.1.

The tree topology of the network and the natural requirement for a rectangular die impose a restriction on the number of pages that can be used. In general, to accommodate page numbers that are not a power of two, a partially depopulated tree is used. Still, not every number of pages is allowable – for example, 10 pages would result in a very large aspect ratio, or a lot of wasted space. On the other hand, a 12-page device could be derived from a 16-page one by depopulating the network, as shown in Figure 6.2. For this reason, a set of valid page numbers is predefined and the search considers only those numbers.

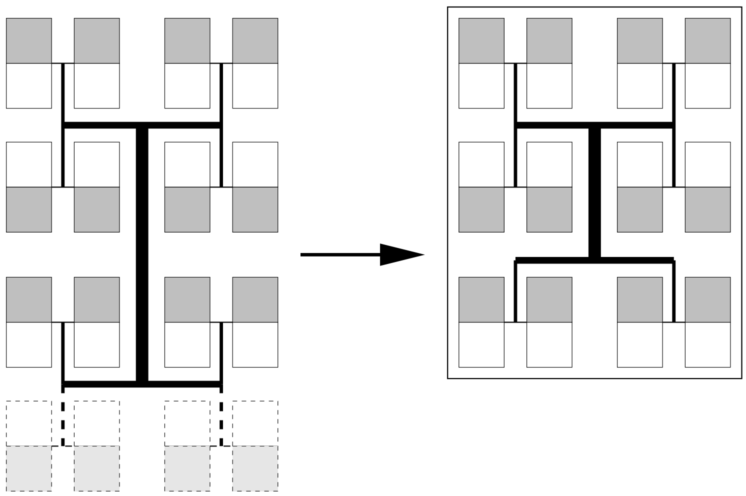

Figure 6.2: Derivation of a 12-page device

The optimization procedure is initially applied to each benchmark individually, resulting in per-application "local" optima, that give the best possible performance for each benchmark. Then, the procedure is applied for all the benchmarks combined. This yields a "global" optimum, that has the best possible average performance over the benchmark set. It also gives a measure of *robustness*: the globally optimum design point will be robust when its performance on any given benchmark is not significantly worse than the performance of the corresponding application-specific optimum on the same benchmark.

Finally, two different area constraints are used: one for a "low cost" design and one for a "high-end" design. The corresponding area bounds have been set (rather arbitrarily) to $64mm^2$ and $256mm^2$ respectively, corresponding to 8 mm square and 16 mm square dies. The search is carried out separately for each design point. In addition, a cost-performance optimum is computed by searching for the point that minimizes the product of chip cost and run time. Three sets of results are given for the low cost, cost-performance and high performance optima.

Perspective

At a high level, this approach is barely new. Benchmarks have always been used to guide designers in the selection of architectural parameters. There are, however, a few points that make this methodology somewhat different from most benchmark-based optimization techniques:

- There is no existing software base. Software for FPGAs is not (yet) portable, so application designs had to be developed from scratch. This limited the number of benchmarks studied in detail to four, whereas more would have been desired.

- Performance scaling is not automatic. Unlike conventional superscalar processors that automatically discover parallelism at the instruction level in an otherwise sequential program, in our environment the parallelism has to be explicit in the implementation[1]. A single "program" will not be appropriate for the entire range of design parameters. Kernels have to be designed in a certain style that allows them to be scaled easily depending on the amount of available resources. Sometimes a single scalable kernel design will not be very efficient across the entire design space. Then, it may be better to switch to an altogether

[1]The SCORE simulator, under development in our research group, automatically exploits task-level parallelism, but currently no data parallelism.

different implementation in some region of the space. Choosing the best possible implementation is not always trivial. Some example of multiple implementations will be described in Appendix C.

A similar methodology has been applied by the RAW group at MIT to size the Raw design[85]. Applications, mostly of numerical nature, are modeled at a high algorithmic level and execution time is computed with equations, rather than simulation. The application model is very simple and appears to underestimate the compute requirements somewhat. In this work applications are modeled in a more detailed manner, at the implementation level. This modeling is still an approximation of the compute resource requirements, but it should be closer to the actual implementation costs.

6.2 Area models

Detailed scalable area models are used for the three main parts of the architecture: the compute page, the memory page and the interconnect network. By interconnect network we refer to the portion of the HSRA network that extends above the level of a compute page and is used to interconnect compute and memory pages. Note however that the definition is somewhat artificial, since this is the the same network that extends into the compute page, down to the leaves. The distinction is made for convenience only.

The Trumpet layout has been used as a guideline for the models that follow. However, experience has shown that the design point chosen for the network, in terms of base channel capacity and growth rate, is too wire-rich, leading to an excessive amount of interconnect and large overall area. For this study, I have adjusted the constants in the models to reflect both a smaller base channel capacity (8 wires instead of 10) and network growth rate (Rent exponent $p = 0.6$ instead of 0.67).

The various constants that appear in the models have been calibrated to the existing layout dimensions, designed with a minimum feature size of 0.25μm for the DRAM and 0.4μm for the logic, and scaled to the $0.18/0.25\ \mu$m design rule used here.

Processor core

Different processor models are used for the low cost and the high performance design points. Both are based on the M32R/D embedded-memory processor. For the low cost point, a single-issue core with a 2KB unified cache and 1 MB internal DRAM main memory is used. The on-chip CPU memory is complemented with a 16-bit interface to an external Flash or PROM memory. For the high performance point the core is assumed to be 2-way superscalar, the cache and main memory capacity are increased by a factor of 4, while the external interface is 32 bits wide. While the amount of CPU memory in both design points may appear small, most of the computation is intended to take place in the array, using the memory pages embedded in the array. So, the demand for CPU memory is reduced accordingly. With 0.18μm/0.25μm design rules for DRAM and logic, the CPU, caches, and CPU memory take about 17mm^2 and 56mm^2 respectively in the low cost and high performance design points.

Compute page

The basic model for the compute page consists of two parts: a model for the "core" of the page (the BLBs and intra-page network) and a per-page overhead (mainly page controller and configuration I/O and clocks).

The core model is superlinear to the number of BLBs, because of the wiring channel growth. The capacity of the wiring channels grows as $W = W_o \times N^p$ as we travel from the leaves (BLBs) to the root, where W_o is the base channel capacity, N is the number of BLBs connected to that channel, and p is the Rent's exponent. Consequently, the compute page area should be expected to grow in a similar way. Indeed, if we assume a similar model for the area, $A = C \times N^k$, and fit C and k to existing layout, k comes out very close to twice the Rent's exponent. For the Trumpet layout at 0.4μm design rule, k=1.30 and C=0.039 mm^2, numbers that correspond to the Rent's exponent and the single BLB area respectively. To reflect the reduced wiring assumed in this study, a value of $2 \times p = 1.20$ is used for k, while C is set to 0.033 mm^2, or 87% of the original value. This value reflects the fact that, while the base channel capacity is reduced to 80% of the original, the area is in part determined by the BLB circuits, that are not affected significantly. The resulting area trend is shown in Figure 6.3.

The per-page overhead is dominated by the configuration I/O and the configuration memory controls. These circuits scale as the lateral dimension of the array, with a fixed depth. Therefore, the model used has the form $C' \times \sqrt{A_{core}}$, where C' is a constant to be determined. Using the core area model from above, the area overhead can be rewritten as

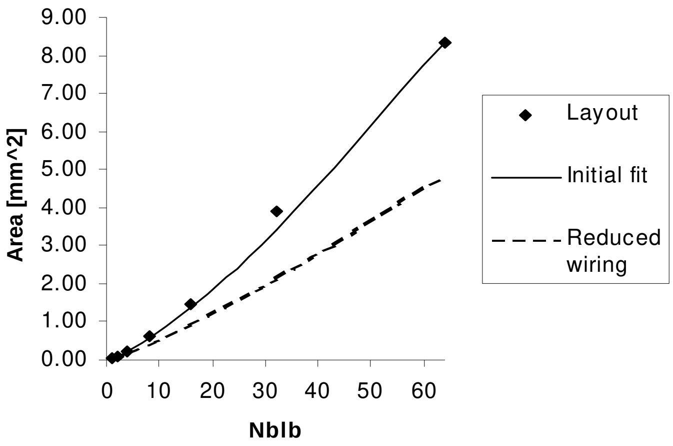

Figure 6.3: Compute page area fit (at 0.4 μm)

$D \times N^{k/2}$, where $D \equiv C' \times \sqrt{C}$. D is derived from the Trumpet layout: D = 0.29 mm.

Memory page

Modeling of the memory page area follows the Trumpet CMB design. Three terms are used: one proportional to the memory capacity, one proportional to the interface bandwidth, and a fixed overhead term.

The memory capacity term models the cell array, X and Y decoder area and the redundancy fuses, that roughly scale as the amount of memory. For the LGS macro with the 0.25μm design rule, the coefficient for the capacity term comes out to $C_c = 1.72mm^2/Mbit$.

The bandwidth term is further decomposed into three parts, corresponding to the configuration bandwidth, the application bandwidth and their product. The configuration bandwidth term models the DRAM macro datapath circuits and part of the CMB datapath, that depend only on the configuration bandwidth. The application bandwidth term models the area of the address crossbar, stall buffers, retiming registers and includes a contribution from the data crossbar. All these terms scale as the application bandwidth. Finally, the product term models the $O(N^2)$ nature of the data crossbar. Remember that this crossbar is not square, but W_{cfg} bytes on the memory side by W_{app} bytes on the user side.

Finally, the constant term, C_o, models primarily the various control circuits, the clock buffer and some remaining unused space. The simplified DRAM macro interface is assumed, without the SDRAM interface controller.

With everything put together, the total CMB area is computed as:

$$A_{CMB} = C_o + C_c \times S_{cfg} + C_{bc} \times W_{cfg} + C_{ba} \times W_{app} + C_{bx} \times W_{cfg} \times W_{app}$$

Although this simple model does not capture every square micron in the layout, it comes within 5% of the area of the Trumpet CMB (without the SDRAM interface controller), or 8.53 mm^2.

Upper-level network

The portion of the network that extends above the level of a single compute page maintains the same growth characteristics as within each compute page. From the point of view of an application netlist, the division of the compute fabric into pages is purely artificial and is done for management purposes only. Therefore, Rent's Rule should apply to the upper levels as much as it applies to the lower ones, yielding channels with $W = W_0 \times N^p$ wires connected to a subarray of N BLBs. In the reduced wiring design, $W_0 = 8$ and $p = 0.6$. Based on this wire population, an iterative procedure is used to calculate the network area at each design point, under the somewhat simplifying assumption that pages are square. When the number of pages is not a power of two, linear interpolation between the closest powers of two is used to approximate the area of the corresponding depopulated tree.

6.3 Applications

Four applications serve as the benchmarks for this study: fingerprint matching, JPEG image compression, wavelet image compression, and speech recognition. For each of these applications, a program module estimates their performance, measured as the number of cycles to completion.

Performance estimates are based on a "paper design" of each kernel, typical data sets and the current set of the device parameters. No simulation is carried out. For the simpler kernels with static control flow, it is usually trivial to count the number of cycles to completion for a given input. For more complex kernels, where the control flow may be data dependent, profiling of typical data sets is used to determine average, "expected" behavior. For example, Huffman coding, which is part of the JPEG and wavelet benchmarks, requires random accesses into the table of Huffman codes. When this table does not fit in the DRAM sense amplifiers, a number of misses will be sustained. Instrumentation of an existing software implementation and profiling on a number of images gives a good estimate of the miss rate. It should be noted that it is not the software implementation that gets characterized, rather the features of typical data sets and algorithms.

For the simpler application kernels, it is easy to develop an implementation that will scale gracefully as the number of available BLBs, memory bandwidth and memory capacity, when applicable, is varied. For more complicated kernels, some times there is no trivial way to scale an implementation to take advantage of varying amounts of BLBs or memory. In that case, it is often necessary to develop multiple implementations, each one valid for a particular range of design parameters. Examples of such cases in the case of the wavelet and speech benchmarks will described in Appendix C. The program that estimates the performance needs to select the appropriate implementation, based on the design parameters. Most of the design effort for this study lies in developing these often multiple implementations of the compute kernels.

Appendix C gives a brief overview of the applications used and highlight some of the more interesting implementation and modeling issues.

6.4 Results and discussion

Each point in the design space, in terms of the five parameters P, N_{BLB}, S_{CMB}, W_{app} and W_{cfg}, yields a certain device area and a certain performance for each of the applications. Figure 6.4(a) is a scatter plot of all valid design points for the fingerprint benchmark with a $256mm^2$ die size limit. Valid design points are defined as the ones that satisfy the area upper bound and provide enough resources for the application to run. It is immediately obvious that there is some structure in the design space. Most points come in clusters in the area-performance space. Closer examination of these clusters shows that they correspond to the parameters that affect application performance most. In most cases these are the total number of BLBs, $P \times N_{BLB}$, and either the memory page interface width, W_{app}, or the total width of all memory pages, $P \times W_{app}$. These parameters directly influence the amount of parallelism achievable by most kernels. Within each cluster, the remaining parameters, S_{CMB} and W_{cfg}, differentiate the various points. The effect of these parameters on performance comes primarily through the reconfiguration overhead. Note the shape of most clusters, consisting of an initial drop and a long tail. This suggests that a certain amount of memory capacity and configuration bandwidth can have a significant

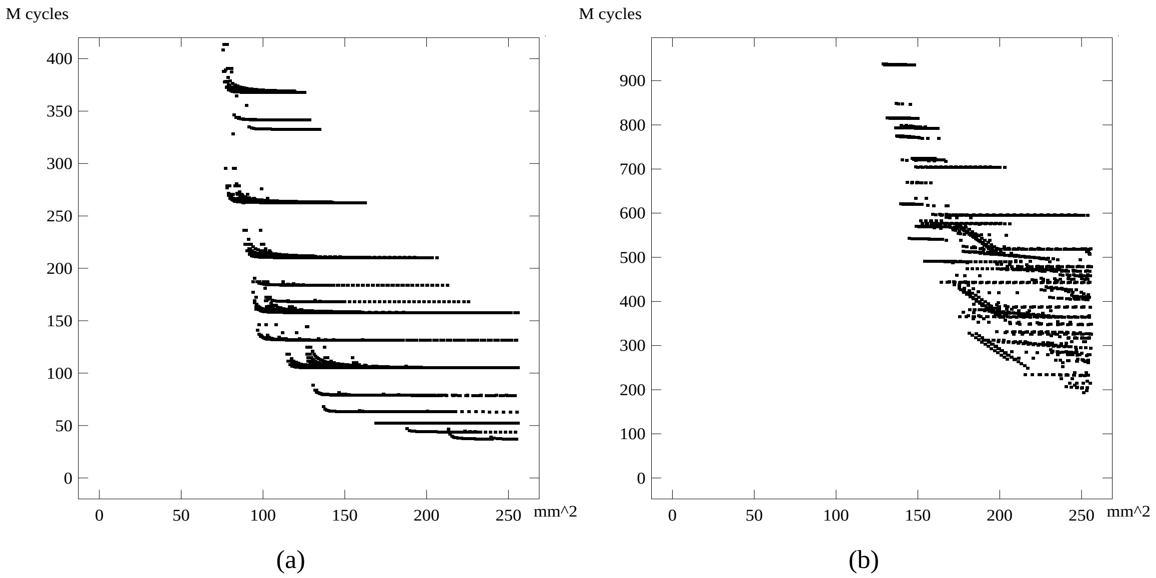

Figure 6.4: Performance vs area scatter plot for (a) fingerprint benchmark and (b) combination of all benchmarks

effect on the total execution time, while above that point the effect is negligible. In general, points that lie on the lower convex hull of the space represent the best design choices along the area-performance spectrum.

Clustering is not always so pronounced. In particular, when all applications are combined to derive the global optimum, the space appears to be populated much more randomly, as illustrated in Figure 6.4(b). This effect suggests that the combination of the four benchmarks is successful, to a certain degree at least, at damping the influence of the particular features of each individual kernel, something that might bias the results of the optimization.

The overall shape of the space highlights a significant limitation of the simple exhaustive search as a means to determine the optimum design parameters. Simply looking for the point that minimizes the run time and overlooking the area (as long as it meets the upper bound constraint) may lead to a choice that is very area-inefficient. For example, in the case of fingerprint matching, the simple search would yield the point at the bottom and far right in Figure 6.4(a). However, at $256mm^2$, this point has almost identical performance as several other points down to about $220mm^2$. The last 30-35 mm^2 allocated have almost no effect on the performance. To overcome this limitation, the search employs a two-step procedure: first, an exhaustive search scans the design space and points with run time close

	Fingerprint	JPEG	Wavelet	Speech	GLOBAL
LOW COST					
P	4	4	6	6	6
N_{BLB}	128	128	64	64	64
S_{CMB} (Mbits)	1.25	1.25	1.25	1.75	1.75
W_{app} (bits)	64	32	64	16	32
W_{cfg} (bits)	64	64	64	16	64
Area (mm^2)	61.6	59.9	62.2	59.4	62.5
Run time (Mcycles)	135.5	116.8	100.0	153.2	662.6
Chip cost ($)	6.50	6.32	6.56	6.27	6.60
$ $\times$ Mcycles	880.8	738.2	656.0	960.6	4373.2
COST-PERFORMANCE					
P	4	4	6	24	8
N_{BLB}	256	128	64	32	128
S_{CMB} (Mbits)	1.25	1.25	2.00	0.50	2.25
W_{app} (bits)	128	32	64	16	64
W_{cfg} (bits)	128	64	64	32	64
Area (mm^2)	102.0	59.9	66.6	126.8	123.1
Run time (Mcycles)	64.6	116.8	85.9	41.3	261.3
Chip cost ($)	11.22	6.32	7.03	14.57	14.04
$ $\times$ Mcycles	725.4	737.7	604.1	601.9	3668.7
HIGH PERFORMANCE					
P	4	24	12	32	20
N_{BLB}	512	64	128	32	64
S_{CMB} (Mbits)	3.50	0.50	2.00	1.50	2.25
W_{app} (bits)	128	64	128	16	64
W_{cfg} (bits)	128	64	128	32	64
Area (mm^2)	222.3	251.1	242.1	237.1	251.2
Run time (Mcycles)	37.7	37.2	32.9	31.1	174.9
Chip cost ($)	33.31	39.62	37.58	36.47	39.65
$ $\times$ Mcycles	1255.8	1473.9	1236.4	1134.2	6934.8

Table 6.2: Optimization results for the embedded DRAM-based device. Results are given for the local optimum for each individual benchmark and for the globally optimum device.

to the running minimum are saved. Then, a *fine tuning* step is applied to find the "best" point within this set. This point is determined as the one that minimizes the chip area while achieving run time "very close" to the minimum. The required proximity to the minimum run time is a parameter to the optimization. It was determined by experimentation that a 2% increase in run time allows all applications studied here to approach the knee of the best performing cluster, minimizing area waste with negligible performance penalty.

Table 6.2 gives the results of the optimization procedure for each application individually, as well as for all the applications combined, for the low cost, the cost-performance and the high performance design point. It is seen that the fine tuning procedure some times leads to designs with total chip area well below the upper bound.

Based on the above results we can make the following observations:

Scaling trends: When scaling from the small to the large design point, two distinct trends can be observed. Fingerprint matching scales best when the page size is increased. JPEG and speech perform better with a large number of small pages. This difference in *granularity* drives the local optima to much different points, with the global optimum being a compromise between the two. These opposing trends can be easily traced back to the characteristics of the individual applications.

Fingerprint matching is mostly computation, without any application memory and only modest stream buffering between pipeline stages that are not coresident on the array. Because of its regular structure, it can make efficient use of all BLBs available. The better area efficiency of larger pages pushes the local optimum to a small number of large pages. Note that in all design points the local optimum for fingerprint has the largest total number of BLBs. Speedup is just about linear with respect to die area, with a slightly superlinear speedup between the low cost and the cost-performance points due to the discrete character of the compute pages.

The rest of the applications contain one or more kernels that require a significant number of individual pages (as well as BLBs) to achieve high parallelism. The entropy coding kernel, two flavors of which appear in the JPEG and Wavelet benchmarks, is characterized by sequential dependencies that make it difficult to exploit parallelism within one block of data. To achieve high throughput, the candidate implementations must encode different, independent blocks concurrently (outer-loop parallelism). The control flow and therefore the memory access patterns are data-dependent and will unavoidably differ from block to block. Therefore, these blocks need to be resident in different memory pages and be processed by independent, identical datapaths. This arrangement makes efficient use of most available BLBs that would remain underutilized otherwise. In their respective optima, JPEG and Wavelet use 4 parallel datapaths each for their entropy coding kernels.

In the same manner, the rather irregular representation of words makes it very difficult and ineffective for speech recognition to process more than one state transition at a time. Outer-loop parallelism can be used again to process multiple words in parallel. Because of the variable length of the word models, this can only be done across independent memory pages and datapaths. In the high performance local optimum for speech 6 parallel datapaths are used, with each datapath using one page for its part of the path memory, three pages for its part of the dictionary, and a page for stream buffer.

Effects of the discrete search space: It is seen that the available area budget is not always utilized in full. For example, the fingerprint local optimum in the high performance point only uses about 222mm^2, or 87% of the 256mm^2 available. This underutilization of available die area is an effect of the discrete nature of the search space. In this case the problem is that the number of BLBs in a compute page has to be a power of 2. With large compute pages it is not possible to increase the number of total BLBs without exceeding the area constraint. Since increased memory capacity or configuration bandwidth have negligible effect at this point, the fine-tuning step of the optimization procedure picks a design that saves 13% area at the expense of less than 2% in performance. These savings in die area correspond to 18% reduction in total estimated chip cost, from \$41.28 to \$33.76.

Another related effect is that some times we see superlinear speedups. For example speech recognition enjoys a 3.7X speedup for under 2.2X increase in die area between the low cost and the cost-performance points. In fact, for most benchmarks the design space is quite discontinuous, as seen in Figure 6.4, with discrete "jumps" in the best achievable performance at certain points along the area axis. In the case of speech, the cost-performance point uses 5 datapaths that share the 768 available BLBs and 24 memory pages. In the small design point, 384 BLBs and 6 memory pages are not sufficient for 2 datapaths, so only one datapath is used, underutilizing both logic and memory resources. (It is not possible to implement 1.5 datapaths!) A certain degree of load imbalance limits the performance of the 5 datapaths to 3.7 times that of the single datapath, an advantage that is still larger than the increase in die area.

Robustness: Table 6.3 shows the performance of the *global* optimum on individual applications, relative to that of each *local* optimum[2]. In addition, an *overall* robustness can be defined as the overall slowdown on the global optimum relative to the sum of the execution times of the individual benchmarks on their respective optima.

[2]The product of chip cost and run time is reported for the cost-performance point.

	Fingerprint	JPEG	Wavelet	Speech	Overall
LOW COST (MCycles)					
Local	135.5	116.8	100.0	153.2	505.5
Global	211.9	141.8	156.8	153.2	662.6
Slowdown	1.56	1.21	1.57	1.00	**1.31**
COST–PERFORMANCE ($\$ \times$ MCycles)					
Local	725.4	737.7	604.1	601.9	2669.1
Global	1110.0	799.7	659.3	1099.7	3668.7
Ratio	1.53	1.08	1.09	1.83	**1.37**
HIGH PERFORMANCE (MCycles)					
Local	37.7	37.2	32.9	31.1	138.9
Global	52.5	47.0	34.0	41.5	174.9
Slowdown	1.39	1.26	1.03	1.33	**1.26**

Table 6.3: Robustness of global optima

We see that the relative performance of the local and global optima ranges from identical to a factor of almost 2. The slowdown from the local to the global optimum is most severe in the case of the speech and fingerprint benchmarks, that consist of a single kernel each. The JPEG and wavelet benchmarks, each consisting of a number of kernels, enjoy a nice "averaging" effect, since some kernels suffer a slowdown while others do not. The same averaging effect applies to the overall robustness of the global optima.

Reconfiguration overhead: Another interesting outcome of this design sizing procedure is that the time spent in reconfigurations is a very small part of the total time. As table 6.4 shows, it is always under 6% of the total execution time in the applications studied here. We can therefore conclude that the on-chip memory organization is quite effective in mitigating the reconfiguration overhead. It does so primarily through the high memory bandwidth, the parallelism possible across pages and the large stream buffers that minimize

	Fingerprint	JPEG	Wavelet	Speech	GLOBAL
Low cost	3.1%	2.4%	5.5%	0.0%	0.7%
Cost-performance	2.5%	2.4%	2.1%	5.9%	1.5%
High performance	2.6%	2.1%	3.6%	1.6%	0.3%

Table 6.4: Reconfiguration overhead

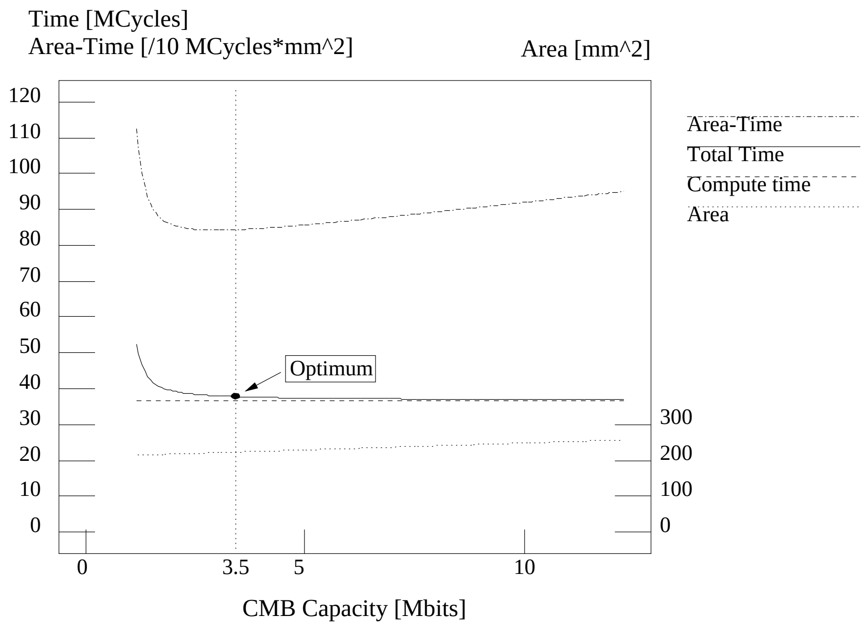

Figure 6.5: Memory page capacity and reconfiguration overhead for the fingerprint benchmark. All parameters except for S_{CMB} are set to the values in the local optimum design.

the reconfiguration frequency. With compute page size ranging from 32 to 512 BLBs and configuration path width ranging from 16 to 128 bits, the mean configuration latency is about **1370 cycles**, or **6.85 μsec**, with a 5 nsec clock cycle. This overhead is amortized over computation epochs with a mean duration of about **53K cycles**, or **265 μsec**, for an average 2.6% reconfiguration overhead across all the local and global optima.

These results on reconfiguration overhead are computed under the assumption that page scheduling occurs immediately after a stream buffer is full (or empty) as opposed to discrete intervals. In addition, any scheduling overhead is neglected. This approximation may be justified by the fact that scheduling is very simple: in principle, it is a static round-robin schedule between the compute kernels of each application. Furthermore, in most cases scheduling decisions could be made by the host during kernel execution in the array. A small overhead for initiating reconfiguration events and restarting the execution will be unavoidable, but it has been neglected. Experimentation shows that the addition of a generous 1000-cycle scheduler overhead to each reconfiguration event changes the results very little.

Generally, CMB capacities of about 1 to 3 Mbits are sufficient for configuration and user data storage, as well as enough stream buffer space to bring reconfiguration overhead

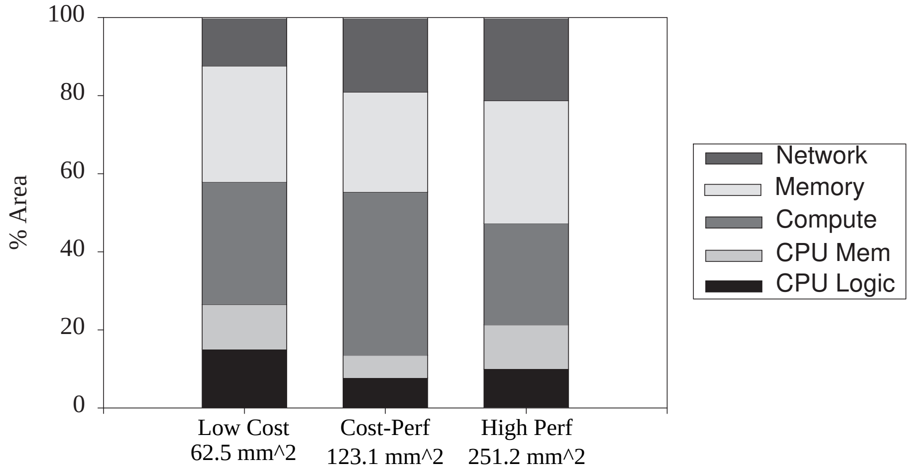

Figure 6.6: Area breakdown of the global optima

down to a small percentage of the total time. Further increase of the memory page capacity yields negligible improvement. This effect is most prominent in the case of the fingerprint matching application: initial optimization for the large design point yields a memory page capacity of 12 Mbits, with resulting chip size just under the limit of 256 mm^2. However, at that point the configuration overhead is virtually nonexistent, which does not justify the implementation of such a large memory page. If we scale back the amount of memory to the final value of 3.5 Mbits, we incur an overhead of only 2%, equal to the fine-tuning amount. At the same time, the area-time efficiency metric drops (improves). Note how the shape of the run time curve in Figure 6.5 is very similar to the shape of the clusters in the performance-area scatter plot. The chosen optimum lies well below the knee of the curve, resulting in a very small reconfiguration time, 2.6% of the total run time in this case. In other applications, the same effect takes place, although it is less prominent.

Area allocation: Figure 6.6 shows the breakdown of the area of the three global optima. Note that with CMB capacity in the neighborhood of 2 Mbits, approximately half of the memory area is actually used for the interface functions.

With these three designs, low cost, cost-performance and high performance, resulting from the optimization procedure, it is interesting to revisit the older multi-chip systems and see how single-chip embedded DRAM implementations compare. The balance metrics for Splash and PAM that were presented in the introduction are listed again in Table 6.5, where the same metrics for the 3 Trumpet-like designs are listed as well. It is immediately

System	Total LUTs	Memory/Compute	Compute/BW	Compute/port
Splash	20,480	205 Bytes/LUT	80 LUTs/bit	640 LUTs
Splash 2	12,800	655 Bytes/LUT	50 LUTs/bit	800 LUTs
Perle 0	3,200	164 Bytes/LUT	50 LUTs/bit	3,200 LUTs
Perle 1	10,240	410 Bytes/LUT	80 LUTs/bit	2,560 LUTs
Trumpet (LC)	768	1,792 Bytes/LUT	4 LUTs/bit	128 LUTs
Trumpet (CP)	2,048	1,152 Bytes/LUT	4 LUTs/bit	256 LUTs
Trumpet (HP)	2,560	2,304 Bytes/LUT	2 LUTs/bit	128 LUTs

Table 6.5: Metrics for existing systems and the designs of this study (Note 1 Xilinx CLB $\equiv$ 1 HSRA BLB $\equiv$ 2 4-LUTs approximately)

obvious that in our case the balance of resources leans much more towards memory, both in terms of capacity and bandwidth. Also, the granularity of the distribution of memory in the system is much finer. These differences are not surprising, given the very different constraints in the design of the two classes of systems. In the case of the multi-chip, SRAM-based systems, memory is less dense, leading to less memory per compute element. At the same time, board-level interfaces limit both the total memory bandwidth and the number of interfaces. In our single-chip systems, the interface constraints are lifted and the use of DRAM technology enables much higher memory capacity.

Given that the designers and users of Splash and PAM have recognized the memory system as often being the performance limitation, we should expect our single-chip devices to be able to exploit parallelism better for the same applications, leading to performance benefits larger than the technology scaling alone would suggest. However, the lower performance of the DRAM may be a limitation. Memory latency is the only factor that does not improve with embedded DRAM technology when compared to internal or external SRAM, and might have an adverse effect on performance. In the following section we revisit conventional technologies – external DRAM and SRAM – and evaluate them as an alternative to the embedded DRAM technology we studied here.

6.5 The conventional alternatives

Figure 5.2, repeated in Figure 6.7 for reference, defines the conventional system architecture, that will be the basis for comparison of the embedded architecture against conventional, external-memory ones.

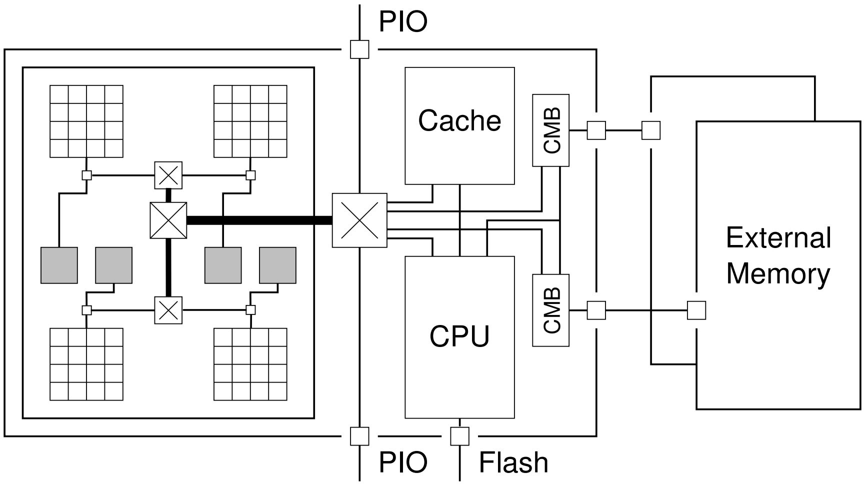

Figure 6.7: Architecture with external memory

The logic chip contains the array, the CPU and its SRAM cache. The external memory can be either SRAM or DRAM, with the SRAM offering higher performance but the DRAM being less costly. It is important to note the existence of two independent memory interfaces that support efficient execution of stream computations. Two external memories are shared between the array and the CPU via the blocks labeled "CMB". These blocks arbitrate between CPU and array accesses, in addition to implementing the other CMB functionality. The available memory capacity has to accommodate all the CPU data, as well as the array configuration bitstreams, data and stream buffers. In the array the CMBs have been replaced by small 4 Kbit SRAM banks, with up to 32-bit wide interfaces. The interface logic is much simpler than the CMB logic, implementing only the retiming and interconnect functions, with almost no control logic. These banks, together with the required interface logic, can be implemented in about 0.35 mm^2 in a 0.25μm logic technology, about an order of magnitude smaller than the 1 to 3 Mbit CMBs that we have in the embedded DRAM devices.

Market constraints will strongly influence the design of the conventional systems. These systems will have to settle for commercially available parts, which may not always fit their needs perfectly. The low cost point will have to use 100 MHz, 16-bit wide parts. These are among the most inexpensive devices in the market, only slightly more costly than slower, narrower options, while faster and wider would be much more desirable. The high performance is based on 32-bit wide, PC-133 SDRAMs. The cost of these two parts is conservatively estimated at about \$3 and \$8 per unit respectively, based on data from large IC trading centers[33, 34]. There is still a certain amount of uncertainty in these prices, so a study of the sensitivity of the results to the cost of DRAM parts is necessary. Experimentation shows that changing the external DRAM cost within a range from 0.5 to 1.66 times the costs assumed here only changes the results by a small amount and certainly does not change the trends observed.

With respect to clock cycle time, the logic device has the performance advantage of a pure logic process. As discussed in Chapter 5, this advantage falls in the range of zero to 15% for current logic and embedded processes. Therefore the internal clock cycle is assumed here to be 220MHz. On the other hand, external memory is significantly slower, running at 100 to 133 MHz. This limitation will have an adverse effect on the memory-limited kernels.

Procedure

We now have a 4-dimensional space to explore. The memory capacity available for the two CMBs is fixed to 3/4 of the external memory capacity. In the low cost design point the CPU memory occupies 1 out of the 4 MBytes of external memory, while in the high performance point 4 out of 16 MBytes are set aside for the CPU. This memory capacity is larger than the array memory capacity in the embedded designs and sufficient for all the needs of the array: it is found that the reconfiguration overhead is always under 3%, which indicates that quite large stream buffers are employed.

The remaining parameters are the number of pages, P_r, the number of BLBs per compute page, N_{BLB}, the external memory interface width, W_{ext}, and the application memory width, W_{app}, which is defined in the same way as for the embedded DRAM device. The available configuration bandwidth will be twice W_{ext}, since both memories can be used concurrently for reconfiguration. W_{ext} ranges from 1 to 2 bytes in the low cost device and from 1 to 4 bytes in the high performance one. W_{app} of course will be bounded by W_{ext}.

The area bound used for the optimization in the previous section is transformed to an equivalent area bound for the logic die based on the constraint that the cost for the embedded device and the multi-chip system (logic chip and external memories) is the same. For the high performance point, the area bound of 256 mm^2 used for the embedded device corresponds to a area bound of 192 to 202 mm^2 for the logic device, depending on the width of the memory interface, which affects the package size. For the low cost point, the conventional system cannot match the embedded memory system with a 64 mm^2 die. To provide sufficient logic resources for each of the four applications, a minimum of 45 to 55 mm^2 are required for the logic die, which is equivalent in terms of system cost to a 95 to 110 mm^2 embedded die. Therefore, for this section the definition of the low cost has been pushed to 110mm^2, to allow the external DRAM implementation to execute all benchmarks. In Table 6.6, the low cost results, the data for embedded DRAM technology are recomputed for the new area bound. In Table 6.7, the high performance results, they are copied directly from the previous section.

Results

The basic result of this section is the performance of the external memory implementations relative to that of the embedded memory device, shown in Tables 6.6 and 6.7. We see that in both design points the globally optimum embedded DRAM device performs better than the corresponding globally optimum conventional device: by a factor of **2.85X** in the low cost design point and a factor of **1.9X** in the high performance point. However, closer examination of the results for the per-benchmark optima shows that only in the low cost point we get consistent speedups, ranging from 2.25 (wavelet) to 5.4 (speech). In the high performance point two out of four benchmarks suffer a slowdown, with speech being the

		Fingerprint	JPEG	Wavelet	Speech	GLOBAL
Ext.DRAM	P	6	6	16	4	6
	N_{BLB}	128	128	32	64	128
	W_{app}	16	16	16	16	16
	W_{ext}	16	16	16	16	16
Ext.DRAM	MCycles	253.8	169.1	311.4	301.9	1067.3
	Time (msec)	1153.6	768.6	1415.5	1372.3	4851.4
eDRAM	P	4	8	6	20	12
	N_{BLB}	256	128	128	32	64
	S_{CMB}	1.25	1.00	2.50	0.75	1.25
	W_{app}	128	32	128	16	32
	W_{cfg}	128	64	128	16	32
eDRAM	MCycles	64.6	68.2	55.1	50.8	340.6
	Time (msec)	323.0	341.0	275.5	254.0	1703.0
SPEEDUP		3.57	2.25	5.14	5.40	**2.85**

Table 6.6: Results for the external and embedded DRAM designs at the low cost design point. The last column corresponds to the globally optimum organization for each technology.

		Fingerprint	JPEG	Wavelet	Speech	GLOBAL
Ext.DRAM	P	6	16	32	6	32
	N_{BLB}	256	128	64	64	64
	W_{app}	8	16	32	16	32
	W_{ext}	8	16	32	16	32
Ext.DRAM	MCycles	56.3	28.7	18.1	262.9	365.9
	Time (msec)	255.9	130.5	82.3	1195.0	1663.2
eDRAM	P	4	24	12	32	20
	N_{BLB}	512	64	128	32	64
	S_{CMB}	3.50	0.50	2.00	1.50	2.25
	W_{app}	128	64	128	16	64
	W_{cfg}	128	64	128	32	64
eDRAM	MCycles	37.7	37.2	32.9	31.1	174.9
	Time (msec)	188.5	186.0	164.5	155.5	874.5
SPEEDUP		1.36	0.70	0.50	7.68	**1.90**

Table 6.7: Results for the external and embedded DRAM designs at the high performance design point. The last column corresponds to the globally optimum organization for each technology.

main contributor to the overall speedup. Therefore, simply looking at the average speedup is somewhat misleading. What is more interesting is the very wide range of per-benchmark speedups, from 0.5 (wavelet) to 7.7 (speech).

Figure 6.8 shows the minimum, maximum, geometric mean and median speedups achieved by the embedded DRAM device as a function of die or equivalent area[3]. The median speedup filters out "outlier" points, in our case the speech benchmark. We see that as we go towards larger devices, the gap between the best and worst performing benchmarks widens. This trend reflects the vastly different memory usage by the various applications. We can distinguish three regions along the area axis. For very low cost (below about \$12.50 in our case, corresponding to the 110 mm^2 embedded chip), the contribution to the system cost of external parts, larger package and increased board area, is so large that the area left for the logic device is not sufficient to fit all the benchmarks. For moderately low cost (between \$12.50 and \$17.00, or equivalent 110 to 145 mm^2 embedded chip), the cost of external components is still a significant fraction, limiting the amount of logic resources and leading to poor performance relative to the single-chip system. As we go to larger

[3]Note the mean is not exactly the same as the speedup of the global optimum because its is based on the execution time of each application local optimum.

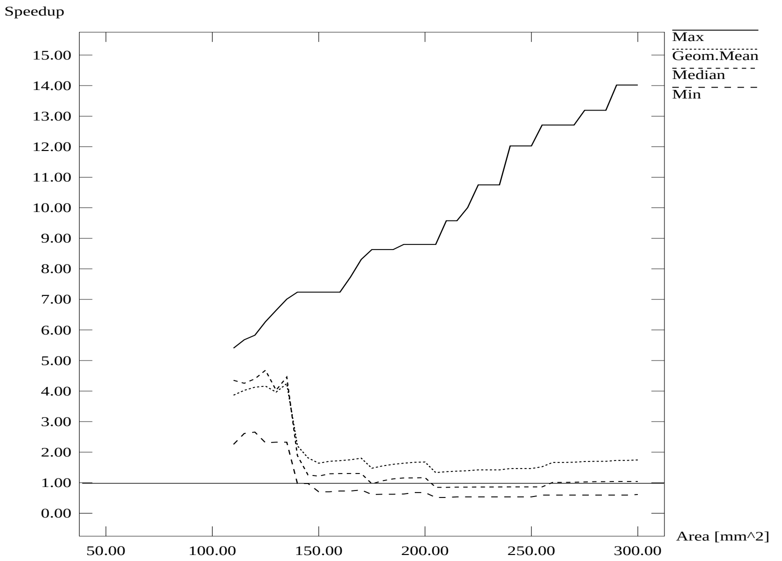

Figure 6.8: Speedups for the embedded DRAM device as a function of die area. Note that the "small" design point is assumed throughout the range, with the small CPU, limited I/O bandwidth and slower, narrower external memory.

budgets, this becomes less and less important. On the contrary, the improved yield of separate memory and logic dies becomes a major factor that favors the multi-chip system. After a certain point, the dependence of total cost to die area is strong enough that a modest area difference between the logic and embedded dies can compensate for the added cost of external parts. With all the memory moved off the logic chip, there is now space for more logic resources than there was in the embedded chip. When there are enough logic resources on the logic chip to implement one or more benchmarks in a fully spatial manner, the performance of those benchmarks becomes independent of the memory. The increased logic capacity allows them to run more efficiently on the logic-only device. On the other hand, for those benchmarks that still need large data structures, as in the case of speech recognition, the external memory bandwidth limit will remain. In such cases, even though we can afford a large amount of logic resources, they will remain underutilized. Notice how the high performance optimum for speech recognition with external DRAM consists of only six small compute pages. The corresponding logic die area is only 45 mm^2, compared to 160 to 210 mm^2 for the other benchmarks. More logic resources would not help performance, because at this point the external interfaces are saturated with accesses to the dictionary and path memory data structures.

Speech recognition represents by far the best case for the embedded device in the high performance point. Limited by the absence of multiple memory ports, the conventional system can achieve no parallelism. In addition, the slow external DRAM limits the performance of the single datapath. In contrast, the embedded device uses six parallel datapaths and a higher bandwidth memory to achieve an overall 7.7 times speedup.

On the negative side, the performance of the wavelet benchmark on the embedded device is limited by the inefficiency of the large CMBs when it comes to handling small data structures. The conventional device can fit a fully spatial datapath. *Line buffers* are used between the horizontal and vertical filter stages for the "corner turn", the alternation between row major and column major accesses, as required by the algorithm. The size of these buffers ranges from 2 to 8 Kbits and they can all be implemented in 12 internal SRAM banks. If the embedded device followed the same implementation, it would be dedicating most of its memory resources to these small buffers, wasting almost the entire capacity of 12 CMBs, that would otherwise be used as stream buffers. This waste would cause the reconfiguration overhead to increase dramatically, impacting performance severely. Instead, each filter stage is implemented as an independent kernel, reading and writing on-chip stream buffers. Wide, data parallel implementations are used in most kernels, except for the first filter stage that is limited by I/O bandwidth. The time for this first stage is the single largest contributor to the overall execution time, utilizing only 17% of the available BLBs and 9% of the internal memory bandwidth!

It is interesting to note that in the low cost point, the situation with wavelet is reversed, since the conventional design cannot fit the fully spatial datapath. There, it is limited by the slow and narrow external memory and the limited area budget for the logic chip.

External SRAM

While SRAM is also an option for the external memory, it suffers a serious handicap from a cost-performance perspective. In addition, the limited density of commodity SRAM chips may require use of more than two memory chips for the external memory, so that enough capacity is available for use by the CPU and the array. This cost disadvantage is especially prominent in the low cost design point. With a minimum of 55 mm^2 needed for the logic die and two 8-Mbit, 16-bit wide SRAMs, the lowest cost for a logic device and its memories will be around \$46.00, which is equivalent to a 288 mm^2 embedded DRAM chip, larger than what we have used for the high performance point!

Only when cost is not an important consideration does SRAM become a viable alternative. Systems based on external SRAM will perform better than DRAM-based systems, because i) commodity SRAMs support higher clock frequencies, frequently up to 200 MHz, and therefore higher bandwidth than commodity DRAMs and ii) there is no overhead caused by row misses as in DRAM. However, a similar trend is observed as for the external DRAM systems: for applications like speech recognition that depend on large data structures, SRAM-based systems still perform worse than the embedded DRAM device, because of the memory bandwidth limitation and the lack of parallelism. Applications like fingerprint, JPEG and wavelet, that do not make heavy use of the memory, perform much better than on the embedded chip – but such applications, ironically, do not justify the high cost of large external SRAMs.

Discussion

The results presented here suggest that the single-chip design based on embedded DRAM technology works best in the low end of the cost-performance spectrum. In low cost designs the cost of the external parts plays a major role in the sizing of the system. At the same time, with the small dies the yield is less sensitive to die area, so the cost advantage of splitting memory and logic into separate chips is reduced. Finally, the limited amount of logic resources often requires that the application logic is split into multiple kernels, with buffering of intermediate data between them. This makes the execution time much more dependent on the performance of the memory system.

On the other hand, it is striking that in the larger, higher performance devices there is such a wide range of speedups. These vastly different trends, as highlighted by the speech recognition and wavelet benchmarks, reflect the different requirements of applications. As discussed earlier, the three primary uses of memory by the array are for configuration storage, stream buffers and application data. As we move to larger (and costlier) high performance devices, there comes a point that stream buffers and frequent reconfiguration are not needed: the logic device provides enough resources for a fully spatial implementation of the datapath. Where this point lies on the area axis depends on the complexity and computational demands of the application. When the application uses large data structures, external memory still limits the achievable speedup. With smaller data structures, like small lookup tables, line buffers, and so on, the internal SRAM banks of the conventional device work much better than the large CMBs.

It should be noted that as fabrication technology advances and the capacity of chips increases, the dichotomy between low end and high end devices is expected to persist. The low end of the cost-performance curve will still be defined based on cost, so that low-end compute devices will be cheap compared to external components, such as memories. Such devices are expected to maintain high yields, close to 100%. Finally, as hardware gets more powerful, in both ends of the cost-performance curve, applications also get more demanding – both in the amount of data they handle and in the number of different functions. In other words, all the factors that contribute to the good relative performance of embedded devices in the low end are expected to be true in the foreseeable future. Referring back to figure 6.8 (page 114), the basic shape of the speedup curves is expected to be the same.

The inappropriateness of the CMBs for small data structures stems from the fact that a single type of resource cannot satisfy very diverse requirements easily. With one type of memory, sized in the Mbit range, we can easily and efficiently support relatively coarse grain data storage needs, like stream buffers and configuration storage. But when we have to allocate CMBs to hold Kbit sized data structures, we end up wasting most of those CMBs' capacity. At the same time, the capacity available for stream buffering is reduced, increasing the configuration overhead considerably.

In the applications studied in this section, when there was such a need for fine-grain user data the optimization procedure led to a very fine-grain array design as well: a large number of relatively small compute and memory pages. This reduced the area wasted in "user data" CMBs and at the same time provided a relatively large storage in the remaining CMBs for stream buffers. Unfortunately, this fine-grain array design is considerably less area-efficient than coarser-grain designs, resulting in smaller logic and memory capacities.

Interestingly, in a general sense this is the same effect that motivates the use of embedded DRAM in the first place: that relatively small distributed memories have a higher performance potential than a single monolithic memory system. We see the same effect when comparing the DRAM memory pages against the fine-grain SRAMs, only at a different scale: that performance benefits may still arise in some gases from smaller, finer grain memories.

Another disadvantage of the CMBs is of course the poor performance of the DRAM: Mbit-sized DRAM blocks cannot match the single-cycle latency of the Kbit-sized

SRAM banks present in most modern FPGAs. This relatively long latency hurts performance considerably when alternating reads and writes are required, as in a FIFO. At the same time, the variable latency of the DRAM core complicates matters and degrades performance further.

The application performance experiments highlight two complementary ways to exploit parallelism: data parallelism and pipelining, or task-level parallelism. The latter is seen to place higher demands on the flexibility of the memory system: data often need to be reordered between kernels, something that requires small data structures such as line buffers, accessed with non-sequential patterns. At the same time, when multiple kernels are executing at the same time, multiple unrelated table lookups may need to take place in parallel, mainly for tasks like coefficient lookup. Such requirements are difficult to satisfy with the DRAM CMBs. Therefore, the embedded DRAM devices rely more on data parallelism than task-level parallelism. If finer grain memory was available, like the internal SRAMs of the conventional devices, it would be easier to exploit both kinds of parallelism as needed. Data parallelism alone is often sufficient to fully utilize the compute resources, but this is not always the case. The structure of the algorithm or limited I/O bandwidth will some times limit the amount of data parallelism available and the amount of logic resources that can be utilized productively. Both these effects are observed in the case of the Wavelet benchmark, which has the lowest performance on the embedded architecture.

Finer grain memory would also improve energy efficiency: small table lookups are inherently more energy efficient on small memories. At the same time, use of more pipelining, made easier with fine-grain memories, relies less on stream buffering and therefore spends less energy moving intermediate data between functional units and memory.

Since small SRAM banks are found to be about an order of magnitude smaller in area than our Mbit-sized CMBs, it may not be unreasonable to combine these two types of memory. In a more advanced design, the basic unit for scaling might be a combination of a compute page, a bulk memory page (a DRAM CMB) and a fine-grain memory page (a small SRAM). This approach of course adds another dimension in the design space, since now we have two different kinds of memory to size. But since such SRAMs are quite small, even if they are not utilized efficiently, the overhead they incur on the design will be rather small.

Chapter 7

Energy efficiency

Energy efficiency is receiving increasing attention in digital system design. Battery powered portable devices, like digital cameras, smart mobile phones, personal digital assistants (PDAs) etc, are quickly becoming ubiquitous. Maximizing the battery life of such devices is a primary consideration for the designers, some times even more so than pure performance.

Systems based on embedded DRAM technology have a significant advantage in energy efficiency over conventional systems, where computation and bulk memory are located in separate chips. The large density of DRAM allows more memory accesses to stay on-chip, so the energy required to drive off chip buses is often avoided. In addition, small internal memory banks are usually more energy efficient than large memories commonly available off-the-shelf. Finally, DRAMs are generally more energy efficient than SRAMs of the same technology generation (and much smaller capacity).

In this chapter I evaluate the energy efficiency benefits of embedded DRAM, specifically for reconfigurable architectures. Basic metrics related to power dissipation and energy efficiency are derived for the Trumpet CMB, generalized for embedded DRAM and compared to a conventional design with external memory, either SRAM or DRAM. To put the results in context, typical power dissipation for active compute pages is estimated and an application case study is examined. Finally, the idle power dissipation is compared to that of conventional systems, something very important for battery-operated devices.

7.1 Sources of energy dissipation

The basic operations of a DRAM are row activation, column read and column write. Different components dominate the energy dissipation in each of these steps.

Energy for row activation is largely determined by the amount of charge drawn from the supply to drive the bit lines to the rails. With the common half-V$_{dd}$ precharge scheme, a charge equal to $n \times C_{bl} \times V_{dd}/2$ is required to activate n bit line pairs, where the capacitive load of each bit line is C_{bl}, typically in the 200-250 fF range. Usually a significant amount of energy is also required to drive the various control transistors in each memory column: precharge transistors, bit line switches, sense amplifier enable gates, and others, depending on the details of the design. For our memory macro, this is comparable to the bit line energy. Driving the word line takes some additional energy, proportional to the number of bits activated, but this is usually much smaller. The above components of energy dissipation scale linearly with the number of bit lines activated. Finally, energy is also required for the row decoders and miscellaneous bank control. This component will often be fixed for one memory bank. However, in large memories one `ACT` command will often activate multiple arrays, each with its own decoder and control, so in some cases this component will scale with the memory capacity.

On a column read, energy is dissipated primarily in the column decoders, Y-select lines, I/O lines and read amplifiers. Additional energy is dissipated in the datapath that follows, which depends on the size of the memory: for small devices this will only be a register stage, while larger ones will usually have a second level of I/O lines and amplifiers in addition. Low swing schemes are typically used for the I/O lines, for reasons of both performance and energy dissipation.

Writes usually dissipate more energy than reads. The same parts are active, with two major differences: the I/O lines are driven full swing with strong write amplifiers, to overcome the sense amplifiers that hold the old data; also, for those bits where the new data differs from the old, the bit lines need to be flipped to the new value. When writes are masked, so that only a subset of the full word is written, corresponding energy savings arise, because part of the bit lines and datapath remain quiet.

Energy dissipation during column reads and writes scales with the word width and the total memory capacity. The dependence on word width is obvious, corresponding to a capacitive load switched per bit: a pair of I/O lines, a read amplifier activated, etc.

The dependence on the total capacitance is somewhat indirect, although significant: larger devices have longer I/O lines, resulting in a larger capacitive load per bit.

Additional overhead is present because of the memory controller, power supply generators (charge pumps and DC-DC downconverters) and for standalone devices possibly a PLL or DLL. The supply overhead generally depends on the total chip size, while the controller overhead will be fixed and may dominate on small devices. Finally, the need to refresh the contents of the memory cell add a small amount of "background" energy dissipation, even when the memory is otherwise idle.

In SRAMs the fundamental energy dissipation mechanisms are somewhat different. In common SRAM designs active memory cells will draw a static current from the bit line loads for as long as they are active. This current will cause a small voltage drop on the bit line, to be sensed by a sense amplifier along the periphery. Since the read is not destructive, no rewrite is necessary, so the bit lines are only driven full swing during writes. With SRAMs, there is no equivalent to the DRAM row activation, so each read and write operation reaches the memory array and causes some energy to be dissipated there, as well as in the periphery. Each memory access will spend energy in the X and Y decoders, a single word line, the bit line loads, the sense amplifiers and the I/O path. This typically consists of some long interconnect lines, another set of sense amplifiers and a register bank. In addition, some energy overhead for the interface controller is required. Finally, a small amount of leakage through the memory cells adds some "background" dissipation, similar to the refresh energy dissipation in the case of DRAM.

Table 7.1 shows the energy required for DRAM row activation and read and write of 16-bit and 128-bit words, for various memory organizations. Data for the internal DRAM was based on `powermill` simulations of the Trumpet memory macro. All internal voltage levels used in the macro are driven from external power supplies. However, for the purpose of this study I have assumed that they are all derived from an external 3.3V supply, with a 90% typical efficiency of the charge pumps and DC-DC converters. Note that all reads occur at 128-bit I/O width. When narrower accesses are desired, Trumpet's CMB logic is responsible for delivering the appropriate bits to the application. Data for external DRAM and SRAM were computed from operating currents and cycle times quoted in datasheets. 16-bit wide 16Mbit DRAMs were used for this study[1], as the smallest ones widely available. Wider I/O

[1]NEC μPD4516161AG5-A80-9NF and Samsung KM416S1120DT-G/F8

would be desirable, but such parts come only in x4, x8 and x16 organizations. Note that current data for 64 Mbit DRAMs are quite similar. This similarity can be attributed, in part at least, to the finer process technology and improved low power techniques in the later devices, trends that result in a slow scaling of power by memory generation[63]. SRAMs, on the other hand, are available at 2Mbit capacity, allowing for a more direct comparison with the on-chip DRAM macro. 16-bit wide parts were used[2] and their characteristics were averaged for this study.

	Int.DRAM	Ext.DRAM	Ext.SRAM
Activate row	6.34	23.90	—
Read 16b	(3.26)	3.30	8.36
Write 16b	1.63	3.30	8.36
Read 128b	3.26	26.40	66.88
Write 128b	2.21	26.40	66.88

Table 7.1: Energy for fundamental operations, in nJoules, for different memory organizations. Note that in the internal DRAM case a 16-bit read is essentially a 128-bit read with all but 16 bits discarded.

From Table 7.1 we can see that the internal DRAM macro is significantly more energy efficient than external DRAMs, while SRAMs are even more inefficient. Note that in the Trumpet macro a read is more expensive than a write, which is rather unusual. It appears that the peripheral data path uses more energy in the read direction, than in the write.

Row activation takes much less energy on the DRAM macro than on external DRAM, because a smaller number of bits is activated with every `ACT` command (1K as opposed to 4K). Large commodity DRAMs usually provide about as many bits in the row address as in the column address. In this way the number of address pins is minimized, but a relatively large number of bits is activated by a row access (a large *page*). For example, the 16 Mbit DRAMs used for the above table have 11 row address and 9 column address bits, in addition to 1 bank address bit. This organization results in 12 address pins total, with a 4K bit page. On-chip macros will usually be smaller and more flexible with the allocation of address bits, often resulting in a smaller page. Small memory pages will be advantageous when access locality is poor. In such cases, activating a large number of bits is just overhead, since they are not likely to be needed. On the other hand, when there is

[2] Micron MT58L128L18P-7.5 and Samsung KM718V789-7

high locality, as in sequential accesses, longer pages fare better, since each command carries a certain energy overhead for control and decode. For sequential accesses this overhead is paid less frequently with a wider page.

Column accesses are also seen to be more efficient in the on-chip DRAM. This is primarily a matter of capacity: the DRAM macro is much smaller than the commodity DRAM. Short interconnect and a single level of read amplifiers are used, so the corresponding energy is smaller. Wide reads and writes can also have a significant effect on the energy efficiency. Reading 128 bits from the external DRAM will require reading 8 16-bit words, with corresponding overhead in the decoder and controller. Of course, for the wide I/O to be advantageous, sufficient locality must exist, so all the bits accessed are actually used. Interestingly, locality within the I/O word makes the internal macro with the wide I/O more efficient, while locality across a page has the opposite effect, closing the energy gap between embedded and external memory.

For accesses to external memory, energy is dissipated in the I/O pads and the external bus, in addition to that spent in the memory core. The same system with two external memories is used here as in the previous chapters. Short buses can be assumed, so the energy spent in switching individual lines is minimized. However, input and output pads still contribute a large amount to the energy dissipation. Energy dissipation on the external bus for basic operations is shown in Table 7.2. For the data in this table, short buses with 4pF capacitance per line and pads with typical capacitance in the 3 to 5pF range have been used. Comparing this data to that of Table 7.1 shows that for small external memory systems with short buses, the bus and pads consume about 10% to 15% of the total memory system energy.

	DRAM	SRAM
Activate row	0.46	—
Read 8x16b	4.92	5.76
Write 8x16b	5.05	6.04
Read 4 Kb page	143.30	184.32
Write 4 Kb page	143.44	193.28

Table 7.2: Energy, in nJoules, dissipated on the external bus for basic operations

In addition to the above, in the page-based architecture that we have been discussing a considerable amount of logic is needed to implement the memory page functionality. The energy dissipation of this block may be significant. For the Trumpet CMB, the energy dissipation of the CMB logic has been estimated with `powermill` simulations. For architectures based on external DRAM or SRAM, it is possible to extrapolate from the Trumpet data for an approximate estimate of the CMB logic power dissipation. The main difference between the internal and external memory implementations is that external memory interfaces are much narrower; 16 bits are assumed here. Therefore the CMB logic for the external memory case is much smaller than the one in Trumpet, dissipating less energy. In addition, when external SRAM is used the controller is much simpler and the stall buffers can be eliminated. Table 7.3 shows the energy dissipated in the logic portion of the existing Trumpet CMB and hypothetical designs based on external DRAM and SRAM for data transfers of different widths.

	Int.DRAM	Ext.DRAM	Ext.SRAM
8-bit R/W	1.56	0.89	0.66
64-bit R/W	2.20	—	—

Table 7.3: Energy, in nJoules, dissipated in the logic portion of the CMB

We observe that because of the significantly larger dimensions, the CMB logic for the wide embedded memory dissipates much more energy than its external memory counterparts. We also see that wider accesses are more energy efficient (about 5.5 times less energy per byte) because the overhead for control, clocking and other factors in every clock cycle is amortized over a larger number of bytes. Finally, we observe that in the case of embedded DRAM the energy overhead in the CMB logic is comparable with the energy dissipated in the memory macro.

7.2 Energy efficiency metrics

The energy efficiency of a memory system can be concisely captured in GBytes/sec/Watt, or GBytes/Joule. Note that in this text a GByte is defined as 1024^3 bytes, *not* as 10^9 bytes. Energy efficiency will normally depend on the memory usage pattern. For example, DRAMs achieve near peak GBytes/Joule in sequential accesses, while random accesses involve a larger number of page activations, increasing the amount of energy required to access a certain amount of data. In addition, read and write streams may have different energy dissipation components. Therefore, to properly characterize a memory system in terms of energy efficiency, the peak as well as effective GBytes/Joule for typical access patterns must be examined.

In our computation model, memory is used for stream buffers, configuration storage and application RAM. The energy efficiency of alternative memory organizations for these three uses is examined in the following sections.

7.2.1 Stream buffers

In the case of stream buffers, the access pattern to the memory consists of long sequential reads or writes. Because of the need to minimize reconfiguration frequency, appropriate implementations of the stream buffer would be large internal DRAM banks, or external memory. In the latter case, both SRAM and DRAM might be acceptable.

Since for a stream it is known that accesses are sequential, it is easy to concatenate successive data elements to minimize the actual accesses to memory. In the case of wide internal DRAM, this scheme allows accesses at the full interface width, which is more energy efficient. In the case of narrow external memory, it is possible to use read/write bursts. Bursts reduce the energy dissipated in the external bus somewhat, because address and control lines only need to switch once per burst. A burst length of 128 bits was chosen for this study, since it matches the width of the Trumpet DRAM macro. For our external memory organizations, this length corresponds to an 8-word burst in the case of DRAMs, or two 4-word bursts for the SRAMs. Note that this mode has not been implemented on Trumpet, but is quite similar to the configuration load and dump operations.

Using the data from Tables 7.1, 7.2 and 7.3, we can easily compute the GBytes/Joule for read and write streams, assuming internal DRAM, external DRAM, or external SRAM. The results of this computation are listed in Table 7.4.

	Int.DRAM	Ext.DRAM	Ext.SRAM
8-bit Read stream	0.51	0.32	0.18
8-bit Write stream	0.53	0.32	0.18
64-bit Read stream	1.76	—	—
64-bit Write stream	2.01	—	—

Table 7.4: Energy efficiency, in GBytes/Joule, for internal and external stream buffer implementations

We observe that internal DRAM is considerably more energy efficient than external memory organizations. This advantage is more prominent when transfers happen at a 64-bit word width, when the CMB logic is more efficient. This difference is attributed primarily to the wide interface and the small dimensions of the internal DRAM macro, as well as the lack of an external bus. Because of the sequential nature of the access stream, the smaller page size of the internal DRAM does not help – instead, it causes some energy to be wasted, since the controller overhead for `ACT` commands will be paid four times as often as for the external DRAM.

7.2.2 Configuration

Reconfiguration of a compute page amounts to a wide sequential read from a contiguous area of memory, of length equal to the configuration bitstream size. Therefore, it is identical to the stream buffer case. For the HSRA as implemented in Trumpet, the configuration bitstream is 68.25 Kbits long. Configuration from the internal memory occurs at the full DRAM width of 128 bits (64 bits at the CMB interface), while configuration from off-chip is limited by the 32-bit width of the combined external interfaces.

	Int.DRAM	Ext.DRAM	Ext.SRAM
Energy (μJ)	4.62	25.43	45.20
Time (μs)	8.80	69.80	69.80
Power (mW)	525	365	650

Table 7.5: Memory system power and energy for compute page reconfiguration

Applying the data from Table 7.4 with an 8 ns clock cycle yields the results listed in Table 7.5. It is evident that reconfiguration from internal DRAM is not only faster, but several times more energy efficient, too. On the other hand, since internal reconfiguration is much faster, the peak power is comparable to the external memory case.

7.2.3 Accesses from application logic

Unlike the previous cases, where the access pattern was known, the access patterns from the used logic are unknown and often very irregular. In general, accesses to small tables that can fit in the page buffers are the best case for DRAM, either internal or external, since no row activations are necessary. More banks, wider pages and multiple row buffers per

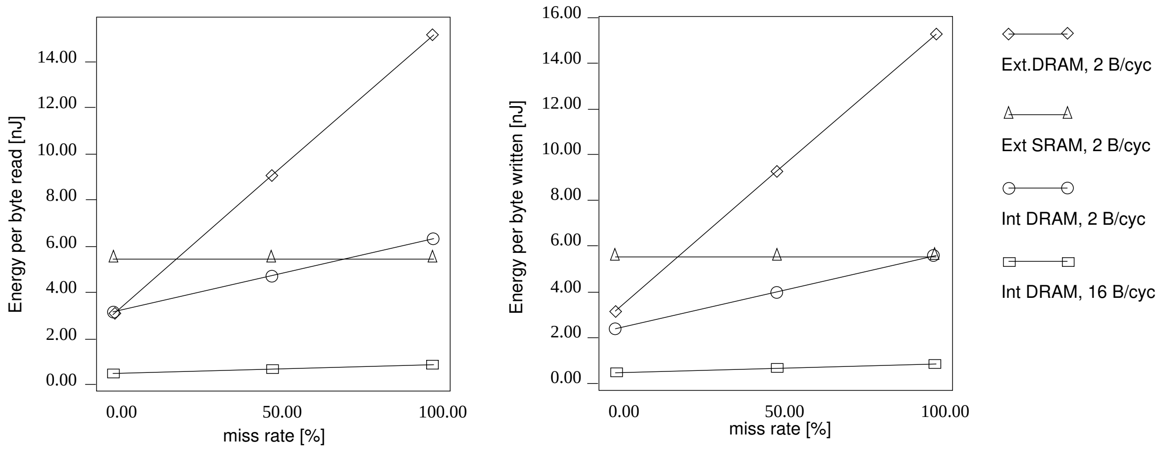

Figure 7.1: Energy per byte as a function of the page miss rate, for various memory organizations, for reads (left) and writes (right)

bank (essentially more SRAM) all help make this case more common. Sequential accesses are near optimal too, since few page activations are needed. In this case too, wide pages contribute to improved energy efficiency, since there are fewer page activations for a given number of bits and the control and decoder overhead is minimized. The worst case for DRAM is random narrow accesses, where a large number of bits is activated and a wide word is accessed with every transaction, while only a few bits are actually needed. With most SRAMs, the random access pattern is not a problem, since all accesses reach the memory core anyway[3].

Technologies competitive to embedded DRAM in this case are both external memory (either DRAM or SRAM), most commonly used for large arrays, as well as small on-chip SRAM banks, as in most current FPGAs, typically used for small lookup tables, FIFOs and other small structures.

Figure 7.1 shows the average energy dissipated per byte accessed for reads and writes, as a function of the page miss rate, for a number of memory organizations, with capacity in the Mbit range. The internal DRAM plots correspond to simulation data for the Trumpet CMB, while the external SRAM and DRAM data correspond to the x16 organizations discussed in the previous sections and includes the external bus overhead. The overhead for the CMB logic is also included in all three cases. Although some variations across existing devices is inevitable, these numbers have beed derived from typical device

[3]Notable exception is the StrongARM cache, where a row buffer is used to avoid accesses to the core when possible and thus reduce energy dissipation.

data and clearly demonstrate the related trends. For the external SRAM, the miss rate is by definition zero, yielding a straight line. With a fixed 128-bit wide interface, the internal memory macro is less efficient for narrow accesses, because of the simple policy used in Trumpet: a full read access is performed, the requested bits are returned to the application and the remaining bits are discarded. For writes, this problem is mitigated somewhat by the use of the byte-level DQ mask bits; however, a significant periphery overhead remains. Note that the external commodity DRAM with the wider pages will usually sustain a smaller miss rate than the internal DRAM with the smaller pages. For example, a 16-bit sequential access will have a miss rate of 0.4% on the external memory (4K bit page) and 1.6% in the internal (1K bit page). A 128-bit stream into the internal memory will suffer from a miss rate of 12.5%. However, as can be seen in the graphs, wide accesses to the internal memory are always the most energy efficient, since the energy of each read operation is amortized over a larger number of bytes.

For smaller data structures, the situation is quite different. Using a Mbit sized memory block to hold a Kbit sized table is clearly wasteful not only in area, but in performance and energy efficiency as well. The unavoidable long interconnect and possibly heavy multiplexing of large memories makes accesses require more time as well as energy. For such cases small internal SRAM banks are ideal.

For example, the Xilinx *Virtex* FPGAs, fabricated in a 0.22μm process, include a number of 4 Kbit dual-ported SRAM banks, with configurable width. In the x16 configuration, each such bank requires only 0.23 nJ per byte, running on a 2.5V supply, as reported by the Xilinx *Virtex Power Estimator* tool[59]. For comparison, the 64-bit wide Trumpet CMB requires 3.19 nJ per byte for a 16-bit read. On the other hand, for wide accesses, the wide interface of the memory macro pays off by amortizing the control and decode overhead of a single access over a larger number of bytes, resulting in an average cost of 0.48 nJ/byte, whereas a small SRAM bank would require multiple accesses, not being able to benefit from the larger number of bits.

7.3 Computation energy

As we saw in the previous sections, on-chip DRAM can reduce the memory system energy dissipation considerably. Of course, the memory system is only part of the problem. If computation energy is significantly larger than memory system energy, the latter will not matter much. Related studies have shown that for typical RISC microprocessors with on-chip caches, the caches consume from 35% to 45% of the total power[42, 108, 109]. Note however that in all cases the processor appears to be running out of the caches and no main memory power is considered. In a recent publication[38], we examined the energy dissipation in a complete memory system, assuming a simple RISC processor executing common applications. With a low-power processor core, like the StrongARM SA110[108], and a conventional memory system, we found that the memory system energy ranges approximately from 40% to 80% of the total, depending on the system configuration and application executed.

Reconfigurable architectures have significant potential energy advantages over conventional architectures, due largely to datapath width specialization, data value specialization, direct data flow that preserves data value correlations and lack of instruction fetch and decode overhead, among other factors. J.Rabaey reports that reconfigurable implementations are between 7 and 350 times more energy efficient than an ARM 6 core on certain DSP functions[99]. Such numbers suggest that memory system energy in reconfigurable systems might be an even larger fraction of the total, than in microprocessor based systems.

For the HSRA, a number of data points have been derived using various methods. Energy dissipation in the interconnect network was computed with `jpsim`, an internal tool used in the BRASS project. `jpsim` performs a cycle-accurate simulation of a given design, as coded in java for the `BOOM` system[23]. The design must have been mapped to the HSRA and placed and routed beforehand. `jpsim` uses placement and routing information, as well as extracted capacitive loads for all levels of the interconnect, contained in a technology file. During simulation, it tracks toggle counts for every node and from that it computes the total energy dissipated. Energy dissipated in the BLBs is calculated with `Hspice` simulations of a single BLB. The average LUTs per BLB (1 to 2) and activity factor derived from the `jpsim` simulations is applied to the Hspice results. Finally, the clock power is easy to compute from the total clock load estimated from the Trumpet layout.

Results from a small number of designs simulated at 125 MHz are summarized in Table 7.6. It is notable that, unlike most common FPGA architectures, the energy dissipation in the network is far from being dominant. This result can be attributed in part to the reduced voltage swing in the interconnect. In addition, certain aspects of the BLB design contribute to somewhat increased energy dissipation there. The clock power is also a considerable part, due primarily to the register-rich architecture of the HSRA.

This contributes a fixed amount per subarray, independent of the number of BLBs actually used. This is something that could be improved upon in later versions of the design, with conditional clocking a possible technique. For comparison, a CLB slice in the Xilinx *Virtex* architecture with minimum wiring and a 100% activity factor, running at 250 MHz, consumes from 5.0 to 11.6mW, depending on the configuration[59]. Note that a CLB slice is comparable to the Trumpet BLB in complexity, process technology and supply voltage.

Test case	BLBs/LUTs	Activity	Net	BLBs	Clock	Power	Energy
8-bit adder	11/11	0.284	1.85	12.10	35.0	48.95	391.6
Invert	37/38	0.092	2.15	28.60	35.0	65.75	526.0
Downsample (dp)	32/40	0.452	15.80	47.55	35.0	98.35	786.8
Downsample (agen)	24/41	0.083	1.85	19.30	35.0	56.15	449.2
Downsample (all)	56/81	0.282	17.35	67.40	35.0	119.75	958.0

Table 7.6: Power dissipation of some test circuits on the HSRA at 125 MHz. Power is expressed in mWatts and nJoules of energy dissipated in a clock cycle.

Overall, the logic power dissipation for small designs on Trumpet falls in the 50mW to 120mW range. If we compare that to the power required by the memory system for typical operations, for example a stream read or write, we see that memory system energy dissipation is significant and can dominate over the compute energy in the external memory configurations. Consider for example a simple 8-bit read stream, that reads a byte every 8 ns cycle. From the data in Table 7.4 it follows that the power dissipated in the stream buffer would be 228mW in the internal DRAM case and 364mW in the external DRAM case. Clearly, the internal DRAM can make a significant difference in the overall power dissipation.

7.4 Case study: downsampling

In this section we take a look at the execution of downsampling, a typical application kernel encountered most commonly in JPEG compression, as well as other image processing applications. Downsampling is a fairly simple kernel, reading and writing 8-bit input streams. For an N-by-N input image, the output is an (N/2) by (N/2) image, where each pixel is the average of 4 input pixels from a 2-by-2 neighborhood. In color images, depending on the application, downsampling can be performed to all or a subset of the color planes. For example, in JPEG compression the input image is represented in the Y-Cb-Cr

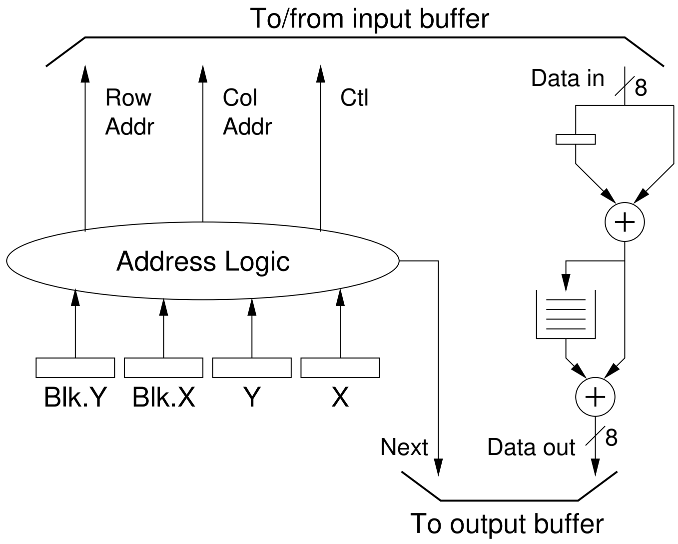

Figure 7.2: Downsampling kernel implementation

color space and downsampling is performed in the Cb and Cr spaces only. Processing of a single color plane is examined in the following.

In a typical implementation of JPEG, color space conversion of the input RGB image will be followed by downsampling of the Cb and Cr planes, and then the 2-dimensional DCT transform will be applied to 8 by 8 blocks of the resulting image. The input to the downsampling kernel is assumed to be in the natural 2-dimensional layout. Because of the irregular access pattern required by the DCT step, it is desirable that 8 by 8 blocks are "gathered" into consecutive memory addresses by the downsampling step. This gather operation is performed in the following way: 16-by-16 blocks of the inputs are processed in sequence. Within each block, each scan line is read in order and neighboring pixels are averaged. During even scan lines, the result is buffered for 16 cycles in a FIFO. During the odd scan lines, new pairs of pixels are averaged and the result is combined with the value coming out of the FIFO. This value corresponds to the 2 pixels in the same X coordinates along the previous scan line. The result is output in sequential order into an 8-bit wide stream buffer. A block diagram of the address generator and datapath are shown in Figure 7.2.

Accesses into the input buffer occur at an 8-bit width, one per cycle. With successive image rows mapped to alternating memory banks, 16 accesses into one bank provide enough time to precharge and activate the next. For every 16 pixels read we have 16 8-bit accesses and one row activation command. This pattern repeats every 16 cycles, or 64ns. Using the data in Tables 7.1 and 7.2, we can easily compute the power dissipation into the input buffer for the different memory organizations, summarized in Table 7.7. The output is an 8-bit write stream, with writes occurring on 1 out of 4 cycles. Finally, as shown in Table 7.6, the downsampling logic consumes about 120mW when running at 125 MHz.

	Int.DRAM	Ext.DRAM	Ext.SRAM
Input	896.6	1,031.6	1,361.3
Output	109.2	181.4	326.1
Compute	239.5	239.5	239.5
TOTAL	1,218.3	1,452.2	1,926.9
Normalized	1.00	1.19	1.58

Table 7.7: Power dissipation, in mWatts, for the downsampling kernel at 125 MHz

The difference in power dissipation between the random-access input buffer and the output stream buffer is quite striking. Largely this is due to the 4 times higher bandwidth at the input. In addition, the output stream buffer can take advantage of the sequential nature of the access stream and coalesce successive bytes; this results in fewer wider accesses in the internal memory case and the use of bursts in the external case, with considerable energy savings in both cases. Also note the rather poor performance of the internal memory in the case of reads, which is a consequence of the simple narrow-read policy used in Trumpet: read 128 bits, return the requested amount and discard the rest. Overall, it is seen that with the particular devices in question, use of the internal DRAM macro reduces the total power dissipation by 15% to 35%.

7.5 Idle power dissipation

Finally, an important consideration is the power dissipation during the idle, or “data retention” mode. This is very important for portable battery-operated devices. Such devices will often lack any non-volatile storage and rely on a battery backed-up memory to retain the data. If one considers that a mobile phone, personal organizer or other such device often spends 99% or more of the time in this mode, then it is obvious that the battery life depends strongly on the data retention power dissipation, possibly more than it depends on the active energy efficiency.

DRAM and SRAM have different power dissipation mechanisms during data retention. In both cases, cell leakage is the major contribution, but the way the lost charge is replenished is different. The cross-coupled SRAM cell has a built-in regenerative mechanism, so the charge lost due to leakage is replenished directly from the power supply. The effect is a small continuous leakage current. On the other hand, DRAM cells have no direct connection to the power supply, so there is a need for the periodic refresh. In this case we have periods of no current draw from the supply, alternating with periods of significant current draw, about as high as in the active mode. Minimizing the leakage of the SRAM cell reduces the level of the "background" current. Minimizing the leakage of the DRAM cell makes the periodic activation less frequent. However, on a large time scale, the periodic refresh of a DRAM manifests itself as a small current draw, allowing for a direct comparison between SRAM and DRAM chips. Review of current datasheets shows that high-end DRAMs tend to have lower idle power dissipation than corresponding high-end SRAMs, by a factor of about 10 to 1: typical supply current for DRAMs in the self refresh mode is about 1mA, while the standby current for SRAMs is typically in the 10mA range.

It is common practice for DRAMs to be "binned" into standard and "low power" parts. Low power parts meet higher retention time, resulting in a smaller self refresh current, typically 0.25 to 0.5 mA. Unfortunately, internal DRAM has the disadvantage that the system designer cannot choose individual parts for their self-refresh current. It is not possible to group parts into standard and low power bins, unless all the logic fabricated on the same chip follows.

On the other hand, use of external parts often requires the designer to use more memory than actually needed, because of the large capacity of commodity chips. In that case, the self-refresh current is determined by the leakage in the entire device, whether used or not. For such cases, a "bounded" self refresh mode, where the internal DRAM controller only refreshes a certain region of the memory, might be beneficial. But no such feature is currently available. Such an innovation would have to make it through the standardization process, that maintains the commodity nature of "off-the-shelf" parts. With internal memory, where the designer can use the exact amount of memory needed, this kind of problem does not exist and self-refresh current is reduced. Whether the small capacity or the lack of a "low power" bin dominates depends on several factors, like the process technology, memory size and application.

7.6 Summary

We have seen that internal memory macros can be much more energy efficient than external commodity memories, primarily because of the smaller interconnect overhead and the lack of an external bus. The wide interface can lead to further energy savings when the accesses from the application are also wide. The data in Tables 7.1 and 7.2 support these findings. There is, however, a significant overhead to support a wide range of interface widths and a flexible interconnect topology, as seen in Table 7.3. Such flexibility cannot be achieved without energy expenditure and partly offsets the energy savings from the internal memory. In the Trumpet CMB, for 8-bit wide accesses, 60% to 65% of the total CMB energy is spent in the interface logic. This overhead is estimated to be only 45% and 20% in the cases of external DRAM and SRAM respectively, because of the larger energy consumption in these memories and the more limited interface logic. Wide accesses, that match the memory macro width, are the most energy efficient, 6 to 11 times better than external memory.

The end result is seen to be a significant reduction of energy dissipation in the memory system. Most importantly, reconfiguration consumes 5 to 10 times less energy, because of the wide accesses. But even the worst case of narrow accesses from the application logic, like the ones in the downsampling kernel, achieve modest energy savings. It is important to note that the memory system energy dissipation is a large fraction of the total, so reducing the energy dissipation there can have a large impact on the overall energy efficiency.

It should be pointed out that the comparison in this chapter is based on mature commercial products on one hand and a first implementation of a research prototype on the other. Little effort has been made to optimize either the CMB logic or the memory macro for power dissipation. In a mature, production level implementation of the CMB design, power dissipation could likely be reduced considerably. In that case, even larger power savings could be achieved than the results presented here.

Chapter 8

Conclusions

In this thesis we have seen that embedded DRAM technology opens up significant opportunities for innovation. With respect to reconfigurable computing architectures in particular, we have seen that the technology can have a strong effect on application performance and energy efficiency. One possible implementation of integrated memory-rich reconfigurable architectures is the *Trumpet* design. Trumpet can utilize embedded DRAM for configuration storage, stream buffering and application memory, with large benefits in reconfiguration latency, reconfiguration frequency and compute kernel performance.

However, embedded DRAM is not without its limitations and it is not entirely appropriate for all architectures and applications. In the following we review the domain where embedded DRAM performs best, identify the limitations of current DRAM designs and discuss the possibilities for future improvements on the basic DRAM design. Such improvements promise to make the technology applicable to a larger class of architectures and applications.

8.1 Application domain

The analysis presented in Chapter 6 shows that single-chip devices based on embedded DRAM technology perform significantly better than their conventional multichip counterparts primarily in the low cost range of the architecture space. Evaluation of four benchmarks with very different characteristics shows consistent speedups over conventional designs, averaging 2.85. As explained in Chapter 6, there are three main effects that make embedded DRAM more cost-effective in this region than in the high end: first, the cost of external parts in the conventional low cost systems is large relative to the cost of the device itself. Elimination of such parts allows a significant area budget for the single-chip device. Second, larger devices suffer from poor manufacturing yield, which limits the amount of memory and logic that can be integrated in a single chip in a cost-effective manner. With relatively small devices, that achieve die yields near 100%, this is not an important problem. Finally, small reconfigurable devices are often unable to implement applications in a fully spatial datapath, but need to break up the computation in separate kernels. This breakdown stresses the memory system more, making flexible, high bandwidth memory more important in such devices than in larger ones.

In larger devices we see two distinct trends in application performance: Applications that do not depend heavily on user data, such as fingerprint matching and JPEG and wavelet compression benchmarks, perform better on conventional reconfigurable systems. In these systems, the separation of memory and logic into different chips has a large effect on die yield and therefore silicon cost, allowing for significantly larger total area compared to the single-chip case. Then, more logic resources are available to the application, enabling fully-spatial implementations of the relevant datapaths in some cases. Such implementations make reconfiguration and stream buffers unnecessary. With user memory demands low as well, the limitations of the external memory interfaces do not affect performance much. The larger amount of logic resources allows increased performance relative to the single-chip devices.

On the other hand, when large data structures are central in the algorithm, such as the dictionary and path memories in speech recognition, the external memory interfaces can be a severe performance limiter for the multichip systems. For application performance to scale, both logic capacity and memory bandwidth must scale accordingly. Once the external memory bandwidth limit is reached, additional logic resources remain underutilized. Such is the case for the implementation of the speech recognition benchmark, where the conventional high-performance optimum consists of a mere 384 BLBs, compared to 1.5K to 2K BLBs for the other applications.

For a general purpose system, it appears that embedded DRAM technology may be the better choice, since such a system will have to satisfy the requirements of a wide variety of applications. While a conventional multi-chip implementation might lead to modest speedups in non-memory intensive applications, the potential speedups offered by the embedded memory device in the memory intensive applications can be very large, as seen in the case of speech recognition that achieves 7.7 times speedup.

Given the strong influence of die size to cost at the high end and the importance of distributed memory bandwidth for some applications, another possible line of research might be that of partitioning a large system similar to our embedded architecture into multiple chips. It is possible to build systems with multiple Trumpet-like devices in the fashion of the older multi-FPGA systems, like Splash and PAM. That would combine the cost-effectiveness of relatively small silicon dies with the performance benefits of distributed, high bandwidth memory. This approach would introduce the need to partition a logic design across multiple chips, but this is not a new problem; in fact, a large body of research exists on the subject of partitioning (see for example [6, 46]). This approach is a possible area for future research.

8.2 Limitations of current DRAM technology

We saw that with its high density, up to 20 times higher than SRAM, embedded DRAM allows much more data to be stored on chip, enabling high bandwidth access from the logic. In Chapters 5 to 7 we examined the effect of this increased on-chip storage to performance and energy efficiency. It is important, however, to realize the shortcomings of current DRAM technology compared to the more common SRAM-based designs.

First of all, the latency of DRAM is generally behind that of SRAM. Even the shorter column latency does not always compare favorably to SRAM read latencies. Typical column latencies of current embedded DRAMs fall in the range of 6 to 10 ns. This range may be comparable to the latency of commodity high-capacity synchronous SRAMs, but it is much longer than the latency of embedded SRAM banks, sized in the Kbit range, like the ones found in most modern FPGAs. This long latency is a result of the constrained layout of the DRAM sense amplifiers, that must be pitch-matched to the DRAM core, resulting in a small number of very wide rows. In contrast to that, an independent SRAM bank will typically be designed with an approximately square aspect ratio, which yields much better performance.

Furthermore, the unavoidable accesses to the DRAM core introduce a strong dependence of performance on the memory access patterns. In our environment, this is not a problem with reconfiguration and stream buffering, where accesses are sequential; but accesses from the application logic may experience a very large variation in performance, as seen in Table 5.2. It is therefore very important for the application designer to be aware of such patterns and use appropriate data layout for the major data structures. As a case in point, the *multiresolution* representation of an image during wavelet compression, illustrated in Figure C.4, creates an intricate recursive 2-dimensional layout of pixel data. To handle this pattern without significant loss of performance, a rather involved mapping of image rows to memory banks is required. This kind of data layout and corresponding ordering of the memory accesses often represent a significant increase in the design effort on DRAM-based devices compared to SRAM-based systems, where no such problems exist.

Interface options for embedded DRAMs also introduce certain limitations. The simple interface of common SRAMs is easy to use; however implementation of such an interface on a DRAM makes it impossible for the user logic to schedule precharge and activate commands at appropriate times, so that a continuous data flow can be maintained. The DRAM interface makes such scheduling possible, allowing for the best possible performance, but is more difficult to use, in particular when accesses are very irregular or random.

Finally, the need to refresh DRAM and its increased susceptibility to soft errors introduce another level of complexity that will have to be addressed by the hardware designer, system software developer, or application programmer. While ways to handle these issues exist, as discussed in Chapter 2, a small overhead in area and performance is unavoidable.

Another limitation has to do with the granularity of memory pages. It should be noted that most existing architectures include some resources for very fine-grain storage and retiming tasks. Such are the register files in conventional microprocessors, or the small embedded SRAMs in common FPGAs. Pages like the ones in Trumpet, sized in the Mbit range, are more appropriate for relatively coarse-grain data storage and retiming needs. Independent of the base technology, memories in this range of capacity are too "heavyweight" for fine-grain tasks like line buffers, small tables, or FIFOs. Allocating a Mbit-sized memory for a data structure of a few dozen words that is accessed continuously renders almost the entire capacity of that memory unusable. Kbit-sized pages could serve the same retiming purposes without the area waste.

The good performance of the JPEG and wavelet benchmarks on the conventional high end designs is largely due to the small internal SRAM banks that are used for such fine-grain tasks as coefficient lookup and line buffers. Use of internal SRAM for such tasks greatly relieves the pressure on the external memory bandwidth. Such SRAM banks could also be used efficiently in conjunction with the large embedded DRAM banks. Fine-grain tasks could be allocated to the small SRAMs, potentially leading to improved performance in two ways. First, more of the larger DRAM-based pages would be available for use as stream buffers, leading to fewer reconfigurations and reduced configuration overhead. More importantly, small SRAMs can be used efficiently to reorder the data access sequence between kernels when necessary, which might allow more spatial implementations of the application logic. Such implementations will often be able to utilize the existing logic and memory resources more fully. For example, we saw that the wavelet implementation on the high performance embedded DRAM device leaves 80% to 90% of the available resources unused during the first horizontal filter pass. If a more efficient implementation of line buffers was available to handle the "corner turn" problem, it would be possible to combine that with subsequent filters and utilize these resources, for a large reduction of run time.

Using DRAM pages to *augment* embedded SRAMs may be a better approach than *replacing* them. This approach however introduces another level of complexity, both in the hardware design and the application development. The hardware designer would have to balance more parameters in the system – in effect, an additional dimension would be introduced in the search space of the resource balance optimization process. In addition, programming tools would have to get more complicated. It would be more difficult for such tools to map high level designs down to such a heterogeneous architecture efficiently. However, because Kbit-sized SRAMs are about an order of magnitude smaller than Mbit-sized DRAMs, little is lost in terms of area efficiency when they are not utilized effectively. It is therefore plausible that an architecture consisting of three different types of resources, a compute page, a coarse-grain, DRAM-based memory page and a fine-grain SRAM bank, may be an effective combination of the best features of embedded DRAM and conventional SRAM-based designs.

8.3 Possible improvements to the basic DRAM design

Aside from combining coarse-grain DRAM storage with more fine-grain retiming resources, it is also possible to improve common DRAM in a number of ways. In most cases, such improvements would lead to somewhat reduced density of the memory. However, reduced density is less of a problem for embedded DRAM macros that it is for commodity DRAM designs. Whereas the competition in the commodity DRAM market is based primarily on cost per bit, a function of the memory density, the improved performance of embedded designs can have a much larger added value than density alone.

The amount of SRAM storage built in to the DRAM block in the form of sense amplifiers can be increased in a number of ways: with a larger number of banks, wider memory pages, or implementation of multiple row buffers per memory bank. Such approaches will allow larger user data structures to be contained entirely in the faster storage and limit the dependence of application performance on the speed of the DRAM core. This can be an important factor in environments where DRAM is the most fine-grain storage resource. On the other hand, when additional resources are available for the fine-grain tasks, the impact of this approach is greatly reduced.

The performance of the DRAM core itself can be greatly improved, as demonstrated by existing prototypes[40, 92, 124]. This may be the decisive factor that will largely determine how suitable the technology is for a general purpose computing environment. In such an environment, there will always be an important class of applications with access patterns that expose the DRAM core performance. As seen in Chapter 5, there is a very large variation in the performance of compute kernels on embedded DRAM architectures. In the worst case, large slowdowns can be sustained compared to conventional designs. A fast DRAM core will not impact the best case results significantly, since these are the cases that are largely independent of the core performance. It will however greatly improve the worst-case results, leading to a smaller variation of application run time and more consistently good performance.

With respect to interfaces, there is significant room for innovation. Embedded memory designs are not bound by the need to be interchangeable. The common SDRAM interface that is standardized across virtually all DRAM vendors need not be the only option available to the embedded DRAM designer and user. As discussed in Chapter 2, separation of the row and column command interfaces can lead to a protocol that is simpler to implement and more flexible at the same time.

Finally, delayed write schemes appear to be promising for embedded DRAM architectures. In common personal computer and workstation designs, the DRAM is "hidden" behind SRAM caches and most transfers occur in cache line-sized bursts. In that environment, the long turnaround latency of the DRAM data bus, a function of the column latency, is not critical. In systems where the memory is directly used by the computation core, such as network switches and routers, memory bus turnaround is quite critical. Even though such systems are generally based on SRAM, they face the same bus turnaround problems.

This situation has led to the development and deployment of delayed-write schemes in high capacity SRAMs (commonly known as `ZBT` or `NoBL`), widely used in telecommunication applications. With embedded DRAM replacing such high capacity SRAMs in system-on-a-chip designs, it is expected that DRAM bus turnaround will be of equal importance. We have already seen this effect in the implementation of the speech recognition benchmark, both in the embedded and the conventional architecture: the multicycle column latency of the embedded DRAM or external SRAM slows down the read/write accesses to the path memory. Because of such considerations, application of existing delayed write schemes to DRAMs will be a natural evolutionary change.

8.4 Other effects

Performance benefits aside, application of embedded DRAM technology to reconfigurable computing has a very significant effect on the energy efficiency and the form factor of the resulting systems. This is of course true not only for reconfigurable architectures, but other architectures as well. In the embedded and portable application domain in particular, these factors alone will often be the primary reasons to employ this technology. As portable consumer electronic devices get smaller and are more often required to run on small batteries, the use of embedded DRAM will be more beneficial.

8.5 Summary of contributions

Trumpet is the first chip to integrate a reconfigurable computing array with a configurable memory block, based on embedded DRAM technology. Design and implementation of this chip, as well as development of applications, exposed and helped us understand design issues related to the architecture of such devices. In particular, among the many implementation details, definition of an appropriate logical interface to the application and ways to handle the variable memory latency of DRAM stand out.

Aside for the implementation of Trumpet, this thesis presents a methodology for sizing such designs, in terms of the appropriate balance of memory capacity, memory bandwidth and logic resources. Application of this methodology to the Trumpet architecture identifies fairly fine-grain designs as cost-performance optimal: such designs consist of compute pages with 64 to 128 logic blocks and memory pages with approximately 2 Mbits capacity and 32 to 64-bit wide interface.

Another contribution of this work is a detailed comparison of embedded DRAM and conventional architectures based on external memory, in terms of performance and energy efficiency. It is found that for low cost designs, integration of large memory capacity on a reconfigurable logic device offers performance consistently superior to that of conventional designs. On the other hand, for high performance designs, this advantage is dependent on the application's use of large data structures. Significant energy savings are also demonstrated, more importantly for wide accesses. Reconfiguration, as a wide sequential access stream, receives the largest benefit, taking up to ten times less energy than it takes on external memory devices.

Finally, it was made evident by the application performance study that Mbit-sized DRAM pages, although very appropriate for stream buffering, configuration storage and many application data needs, are rather inefficient for the more fine-grain retiming tasks. This experience suggests that a combination of small distributed SRAMs, similar to those of current FPGAs, with coarser-grain DRAM pages, sized in the range of a few Mbits, may be better suited to the needs of most applications.

The current proliferation of embedded DRAM macros offered by a number of DRAM vendors is a promising trend. It appears that there is already significant interest in the industry for this technological direction. As "systems-on-a-chip" become more commonplace, absorbing analog, digital logic and memory functions, it is likely that embedded DRAM technology will become mainstream in the near future.

FPGA technology has made significant progress since its initial conception fifteen years ago. From a simple means to collect random "glue logic" into a single chip, with circuit board area savings being the primary goal, it has reached the capacity and performance that allows it to be employed on high throughput computational tasks. Such tasks require a flexible, high performance memory system to support the logic resources. Embedded DRAM can provide the capacity and high bandwidth required for such tasks. One such initial design was the DRAM-FPGA designed by NEC[86], while our prototype, *Trumpet*, is the first actual chip based on this technology. It is my expectation that the synergy of FPGA and embedded DRAM technologies is recognized by other researchers and that more mature designs of this kind will be developed and deployed in the future.

Appendix A

Implementation of the Trumpet CMB

In this appendix, the implementation of the Trumpet Configurable Memory Block, or CMB, is described in detail.

The high level structure of the CMB, shown in figure 3.4, is repeated in figure A.1 for reference. In summary, the functions provided by the CMB are:

- Implementation of an SRAM-like interface to the user logic.
- Masking of the variable DRAM latency from the user logic.
- Matching the clock rate of the DRAM and the logic, which differ by a factor of 2.
- Some hardware support for DRAM refresh.

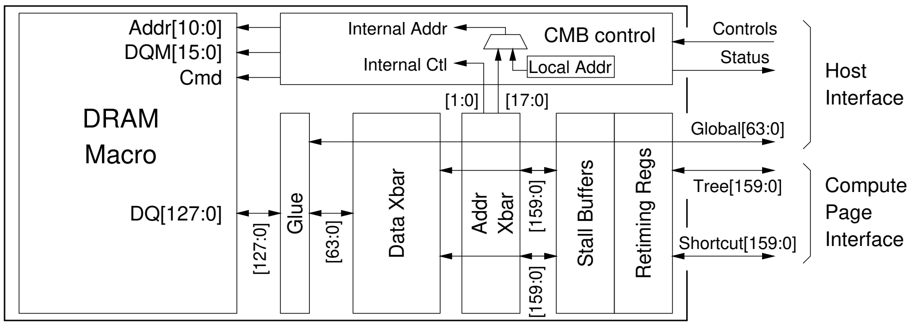

Figure A.1: CMB block diagram (not to scale)

- Control for loading and unloading configuration and state bitstreams.
- Block I/O.
- Flexible mapping of the logical interface signals (address, data, and control) onto the physical network wires.
- Input retiming, as in the compute page.

Physical interfaces

The CMB provides two separate interfaces, one to the user logic executing on the compute page and one to the external controller, or host. Whereas the host interface is hardwired, all the signals of the compute page interface can be mapped onto any of the network wires. This mapping is determined by configuration information loaded into the CMB.

The host can transmit one of 15 commands to the CMB controller over the 4-bit `Op` bus, bundled with a strobe signal (`NewOp`). In response to this signal, the CMB controller enters a certain state. The host can monitor the CMB state which is transmitted to the external interface over a 4-bit `State` bus. For transactions that require a data transfer, a 64-bit "global" data bus, `DGlobal` is used. A strobe signal from the host, `nGlobal`, and an acknowledgement from the CMB, `Ack`, implement a simple handshake. The protocol is such that no explicit address is required, as explained later in this appendix.

Host Interface		
`Op[3:0]`	in	Command from host
`NewOp`	in	Command strobe
`State[3:0]`	out	CMB controller state
`DGlobal[63:0]`	I/O	Data transfer bus
`nGlobal`	in	Data strobe
`Ack`	out	Acknowledge data transfer
Compute Page Interface		
`Tree[159:0]`	I/O	Tree network wires
`Short[159:0]`	I/O	Shortcut wires
`Stall`	out	CMB busy
`ArrayRun`	in	Compute page running

Table A.1: Memory page interfaces

The interface to the compute page consists of 160 tree wires and 160 shortcut wires. These numbers correspond to the bandwidth of the fat pyramid network at the level where compute and memory pages are connected. In Trumpet, where only one page of each kind is present, this is the highest tree level. These are clearly too many wires even for the widest transfers, so as soon as they enter the CMB, they are multiplexed by a factor of 4 down to a set of 80 internal wires. This multiplexing is also controlled by configuration bits. Each wire is bidirectional, but during the lifetime of a configuration it is configured to one direction only. The same direction constraints apply to the compute page, so separate read and write data buses are provided to the application logic.

In addition, a `Stall` signal, generated by the CMB controller, is monitored by the compute page controller. This provides the handshake required for masking the variable latency of the DRAM, as explained in Chapter 3.

Operating modes

Most of the commands from the host make the CMB controller enter a certain operating mode, or *state*. Table A.2 lists all the host commands and corresponding states, where applicable. Note that the first 10 states have direct counterparts in the compute page controller.

Most of the "states" listed here further decompose to sequences of states that generate the detailed timing required by the SDRAM interface of the memory macro. A total of 9 bits of state are used. Four of these bits are exposed to the external interface, to convey the current operating mode to the host, for diagnostic and handshake purposes. The remaining 5 bits allow each operating mode to implement a state machine of up to 32 states. Care has been taken to exploit similarities between operating modes, to simplify the resulting state logic as much as possible.

Addressing

All of the states listed in Table A.2, with the exception of the `RUN` state, involve sequential data transfers. Since the external interface of the chip would likely be the bandwidth bottleneck, it was decided that we can save some bandwidth by not transmitting an explicit address from the external host to the CMB controller. Instead an internal address

Command	State	
`LCD`	`LCD`	Load configuration from DRAM
`DCD`	`DCD`	Dump configuration to DRAM
`LSD`	`LSD`	Load state from DRAM
`DSD`	`DSD`	Dump state to DRAM
`LCX`	`LCX`	Load configuration from ext.I/O
`DCX`	`DCX`	Dump configuration to ext.I/O
`LSX`	`LSX`	Load state from ext.I/O
`DSX`	`DSX`	Dump state to ext.I/O
`RUN`	`RUN`	Application execution
`STOP`	`IDLE`	Idle
`LLA`		Load local address
`DLA`		Dump local address
`READ`	`GREAD`	Input from ext.I/O
`WRITE`	`GWRITE`	Output to ext.I/O
(`LCD/LSD` $\rightarrow$ C.P.)	`LREAD`	Transmit bitstream to compute page
(`DCD/DSD` $\rightarrow$ C.P.)	`LWRITE`	Receive bitstream from compute page
`REF`		Refresh one DRAM row
(chip reset)	`RESET`	DRAM initialization sequence

Table A.2: Host commands and corresponding operating states of the CMB controller

counter, the `Local Address` register is used, in conjunction with a single-bit strobe from the host. Any data transfer between the host and the CMB is preceded by a `LLA` command, that sets the local address to the desired starting address. The appropriate value is transmitted over the `DGlobal` bus. Then, for every word transferred, the host just places the data on `DGlobal` (on writes) and pulses the `nGlobal` strobe. The local address is autoincremented with every word. The local address can also be output to the external interface in response to the `DLA` command, mainly for diagnostic purposes.

The `Local Address` register doubles as a form of base pointer during the application execution state. Remember that a single CMB can hold configuration and state bitstreams, stream data, as well as data for several application kernels at the same time. Some form of relocation and protection is deemed necessary. This is accomplished with the `Local Address` register and an additional register, the `Address Mask`. When an application does not use the entire 2 Mbits, it only transmits as many address bits as necessary. The internal address within the CMB is always 18 bits wide, enough to address the entire memory space at a byte granularity. When fewer address bits are provided by the applica-

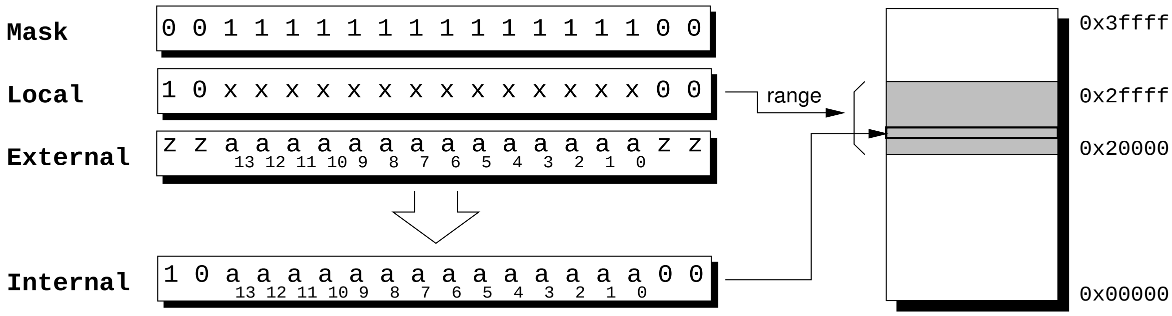

Figure A.2: Base+offset addressing mode. `x` = don't care, `z` = floating input. In this example the third quarter of the address space is used, a condition enforced by the upper 2 bits of the address mask and local address. The application addresses 32-bit words, so the lowest 2 bits of the internal address are cleared.

tion, the remaining bits are filled in from the local address. The address mask functions as a bit field that determines which internal address bits come from the application and which from the local address. This scheme is not as flexible as a complete base + offset scheme, since it requires address space boundaries to be powers of 2. It is however simpler, since the internal address is formed by concatenation instead of addition, so the amount of logic in the memory access critical path is reduced.

The `Address Mask` register is implemented as part of the CMB configuration. In this way, the proper sequence to start (or resume) the execution of an application consists of three steps: load `Local Address` in the memory page with the appropriate base address; load the cycle counter in the compute page with the number of cycles to run; finally, issue the `RUN` command to the compute and memory page controllers at the same time.

In retrospect, it seems that a better approach would have been to make the address mask writable by the host, instead of linking it into the CMB configuration. It is the responsibility of the host to enforce protection between processes, and to do that effectively it has to have control of both the base and the bounds for each address space. On the other hand, the CMB configuration is automatically generated starting with the application source code. Relying on that for part of the protection makes it possible for errors in the initial design to corrupt unrelated memory space.

DRAM refresh

Two ways of refreshing the DRAM are provided on Trumpet. There is actually little hardware support provided in the CMB for refresh. Most of the related functionality was left for the external controller (the host), with the reasoning that in an environment with multiple pages the common functionality should not be replicated in every memory page. Unfortunately, because the DRAM in Trumpet is not functional, the refresh functionality in the host was not implemented; instead, a number of variants have been examined.

The standard *Autorefresh* command of SDRAMs can be used to refresh one memory row. This operation is triggered by the host with the `REF` command, when the CMB is in the `IDLE` state. The host is programmed with the refresh interval and the number of rows to refresh each time, and performs a countdown. When the refresh timer expires, the host must examine the state of the memory page. If it is in the `RUN` state, the host will interrupt execution with a `HALT` command and then issue one or more `REF` commands. Upon receipt of each `REF` command, the CMB controller will perform an autorefresh operation, refreshing one row of memory, as in conventional SDRAM. Finally, if an application has been running, it will be restarted with a `RUN` command. But since the refresh operation pollutes the row buffers, they have to be restored before the application can be restarted[1].

With the above protocol, it takes 32 internal logic cycles to refresh one row, plus 15 to 20 cycles to halt and restart the application, when necessary. (Note that commands from the host are issued at the external clock rate, with a period of 8 internal clock cycles.) Therefore, if there are no real time constraints for the application, the refresh overhead is minimized when a large number of rows are refreshed each time, so the stop and restart overhead is amortized over a larger number of DRAM rows. Note however that with this mode there is no way to limit the amount of memory refreshed. In other words, relying on the DRAM macro's internal refresh address counter will result in refreshing the entire memory, even if only a small part of it is actually in use.

This problem can be handled with the second type of refresh. After halting the current application, the host can issue a `READ` or `WRITE` command, to drive the CMB into one of the I/O states. The start address must be loaded into the `Local Address` register beforehand. During the transition into the `GREAD` or `GWRITE` state, the CMB controller will

[1]This is only necessary when the raw DRAM interface is used, since in that case the application logic expects to know what memory page is active at any time.

activate the DRAM row that contains the starting address. For refresh no actual transfer needs to take place. The host can then take the CMB back to the `IDLE` state, load another address into `Local Address`, corresponding to the next memory row, and repeat the dummy I/O sequence. This process is slower for each row, but allows the host to limit the address space that will be refreshed to the region actually in use. It takes 64 cycles to load the local address and move into and out of the I/O state. The same row buffer considerations apply here as to the previous method.

The problem with both the above approaches is that the application has to be interrupted for each refresh cycle. This is unavoidable, since applications work directly out of the DRAM, without a cache to isolate them from the main memory. In most conventional architectures, refresh can be performed in the background while the application is running out of the instruction and data caches. Then, the DRAM refresh appears only as an occasional increase in the cache miss latency. With Trumpet on the other hand, the entire time spent refreshing the memory directly adds to the total execution "wall clock" time.

As discussed in Section 2.4, in some situations it may be possible for refresh to be omitted, or left up to the application logic. In our environment, where each memory page may contain configuration, state and stream data, as well as application data for multiple kernels, this is not possible. The application has no knowledge of what else is stored in any memory page, other than its own data. Therefore, it will be up to the host to keep track of the memory range in use at any one time and adjust the refresh range and period accordingly.

Unfortunately, since no functional DRAM was available to us, it was not possible to make any data retention time measurements. Based from earlier characterization of this process performed by LGS, a retention time of 8 to 16 ms is expected at 120°C. When the entire memory is in use, 2K refresh cycles are needed. In that case, use of the `REF` command is most efficient. If all refresh cycles are performed at once, an overall overhead of 0.26ms is required. On the other hand, if refresh is performed one row at a time, the overhead amounts to about 0.39ms, because of halting and restarting the executing application multiple times. In either case, the refresh overhead represents from 3.25% to 4.9%, which is comparable with commodity products. When no more than half of the total memory needs to be refreshed, refresh via the global I/O states becomes more efficient.

The time the CMB needs to spend in each of the other operating modes listed in Table A.2, like configuration loading or unloading, is short enough that they do not need

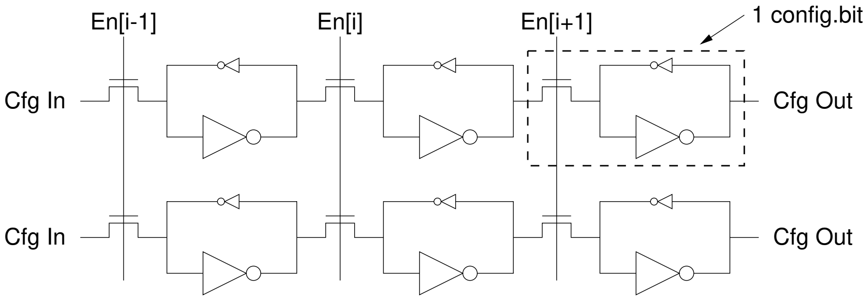

Figure A.3: Configuration memory implementation

to be interrupted for refresh. The longest possible such transaction would be transfer of the contents of the entire CMB to or from the host. The rate of this transfer is limited by the throughput of the host's PCI bus, which has been measured to be about 50 MB/sec for our test setup. This requires about 5ms to read or write the entire memory space, an end-to-end latency less than the smallest expected data retention time.

Configuration and state

The process of loading and unloading configuration and state bitstreams is very similar for the compute and memory page. A set of eight commands and corresponding operating modes allow loading or unloading of configuration or state bitstreams to and from the DRAM or the external interface, as summarized in Table A.2. Loading and unloading of state bitstreams is necessary in order to maintain user state when more than one kernels, or *operators* are time-multiplexed on a page. Unloading of a configuration bitstream may appear somewhat redundant, but is useful for testing the configuration memory.

Configuration bits are implemented as a parallel shift register, as shown in Figure A.3. There is a large number of configuration bits in both the compute and memory page, so it was important to minimize the cell area. Strong drive was not necessary, since configuration bits are static during normal operation. So, the 5-transistor design with small devices was chosen. In this cell, the feedback inverter is very weak, since its only purpose is to maintain the static nature of the cell, and is easily overcome by the forward inverter in the previous cell during the load/unload states.

Area was not as important a consideration for the state bits, since there is a lot fewer of them. On the other hand, drive strength was an issue, since during application execution they will be switching on a cycle-by-cycle basis. Full-custom master-slave static flip-flops were used in the CMB datapath and the standard library flip-flops and latches were used in the controller, which was synthesized from Verilog. Note that we still chose to use flip-flops without a clear or reset signal to save some area and complexity. After power up, an initial "idle" state has to be shifted into both the compute and memory pages, before execution of an application can start.

To match the bit-slice structure of the layout, both the configuration and state chains are 162 bits wide. However, in some columns there are fewer than 162 actual bits, depending on the layout details. The missing bits appear as "holes" in the configuration and state maps, in other words don't care bits that do not map to actual cells. At the page interface (memory as well as compute) the 162-bit wide configuration and state chains are multiplexed down to the 64 bits of the `DGlobal` bus. A 3-phase sequence performs the appropriate folding and unfolding as the bitstream enters or leaves the datapath. Since 162 is not a multiple of 64, this multiplexing leaves some more "holes" in the configuration bitstream, in particular $3 \times 64 - 162 = 30$ dummy rows of configuration bits. This results in a $30/162 = 18.5\%$ overhead in the bitstream size and load/unload bandwidth. The overall *density* of the configuration and state maps, the ratio of actual bits to the total length of the bitstream including all holes is 71.56% and 60.0% respectively.

When a configuration or state bitstream transfer takes place between the CMB (or the compute page) and the host, the 64-bit `DGlobal` bus is further multiplexed down to the 32-bit external interface of Trumpet. The transfer rate is limited by the bandwidth of the external interface, 32 bits at 31.25 MHz, or 125 MBytes/sec peak. In practice, we have found that protocol overhead for crossing the host's PCI bus limits the effective transfer rate to about 60 MBytes/sec.

The CMB can also load its configuration or state from its own memory. This is an entirely local operation – the host triggers the operation with the `LCD` or `LSD` command and then it is free to do something else. The CMB controller will enter the appropriate operating mode and start generating both the requests to the DRAM macro and the timing signals for the configuration or state shift chain. Once the transfer is finished, the CMB will return to the `IDLE` state. By looking at the CMB state, the host can determine that the transfer is completed and issue the next command. During such transfers, the 128-bit DQ bus of the DRAM (running at half the logic rate) is multiplexed down to 64 bits, which are then expanded to the internal shift chain width of 162. Since memory accesses are sequential, in this mode the CMB controller will use prefetching and "ping-pong" the two memory banks, to achieve the peak data transfer rate of 125 MHz $\times$ 128 bits, or 2 GBytes/sec.

Finally, the host can initiate a bitstream transfer between the compute page and the memory page. This is done with one of the first 4 commands of Table A.2, issued to the compute page. The memory page, which should be idle at that time, snoops on the compute page command input and jumps to the `LREAD` or `LWRITE` state. In this case the CMB makes no distinction between a compute page configuration or state bitstream. In these modes, the CMB controller will generate the appropriate timing for the DRAM macro and transfer the data to/from the `DGlobal` bus, which is also accessible from the compute page. A handshake signal from the compute page will instruct the CMB controller to terminate the transfer. In this case, as before, the peak transfer rate of 2 GBytes/sec can be sustained for the duration of a transfer.

Based on the above, Table 3.2 summarizes the latencies for the various bitstream transfers. Note that the length of the bitstreams includes all the holes. The number of actual configuration or state bits can be computed as the bitstream length times the density. Context switch consists of state dump, configuration load and state load, for both the compute and memory pages (it is assumed that the application in question uses the CMB). We observe a factor of 16 difference between the external and memory transfers, which corresponds to the 4X difference in clock rate and 4X difference in width, between the internal and external interfaces. This is actually an optimistic estimate for the external interface, since it is assumed that the host I/O bus can keep up with the peak bandwidth of Trumpet's external interface. In practice, buses like PCI often saturate at about half their peak bandwidth.

	Length [Kb]	Density [%]	Host xfer	DRAM xfer
Memory state	3.000	60.00	3.07	0.19
Memory config	3.750	71.56	3.84	0.24
Compute state	17.125	(?)	17.54	1.10
Compute config	68.250	(?)	69.89	4.37
Context switch	112.250	—	114.95	7.19

Table A.3: Configuration and state bitstream transfer latencies. Times in μsec.

I/O

The primary means of I/O on the Trumpet are a set of high-speed I/O pads and block transfers directly into and out of the memory[2]. The high-speed I/O's are directly connected to the HSRA network and are not examined here. The I/O through the memory is the subject of this section.

It was decided that memory I/O will occur as block transfers: the host downloads some data into the memory, certain processing takes place and finally results are uploaded back to the host. The `GREAD` and `GWRITE` operating modes of the CMB support these block transfers. If this "batch" type of processing is not appropriate for the application, the high-speed I/O's can be used for a more direct on-line method of I/O. Random I/O is not directly supported, but can be implemented if needed as a sequence of block transfers, one word long each. Obviously this will not be very efficient and is not intended as the primary mode of I/O.

Before entering one of the I/O modes, the starting address of the block transfer must be set via the `LLA` host command. Upon receipt of the `READ` or `WRITE` command, the CMB controller will transition to the appropriate operating mode. A single pulse on the `nGlobal` strobe will trigger transfer of each word. The 64-bit `DGlobal` bus is used for the transfer. Because the external interface is only 32 bits wide, only every other pulse on `nGlobal` will actually trigger a DRAM transfer. For global read, one `nGlobal` pulse will read out 64 bits and return 32 bits to the host, while the remaining 32 bits are saved in a register at the chip I/O interface. The next `nGlobal` pulse will just retrieve those saved 32 bits. For writes, the first pulse will write 32 bits into the interface register; the second will provide 32 more bits and trigger writing of the entire 64-bit word into memory. Therefore writes have to consist of an even number of words. If an odd number of words is available, they should be padded with one dummy word at the end.

With 32 bits running at 31.25 MHz, Trumpet can support a peak I/O bandwidth of 125 MBytes/sec. It is likely that in most systems the external bus will limit that to something less. For example, our test system is based on the PCI bus and the `PLX 9050` interface chip from PLX Technology, a setup that has been clocked in the lab to peak at about 50-60 MBytes/sec.

[2]I/O through the root of the tree network was not implemented in this test chip.

Interconnect

In terms of area, a significant part of the CMB interface is taken up by the interconnect structure. This consists of a pair of circuits, the *address and data crossbars*, designed to allow a flexible mapping of the logical CMB I/O's (address, data and control bits) to the internal wires and eventually to the DRAM macro terminals. Despite the name, these circuits are less than complete crossbars.

The need for some kind of interconnect network arises from the properties of the tree network in the compute page. Ideally, we want to be able to connect any BLB input or output to any DRAM macro terminal (data, address, or control). A simple static connection of the DRAM macro terminals to a fixed subset of the tree wires at the compute-memory interface is not sufficient. The fat tree topology can satisfy the any-to-any connectivity only when both sides of the tree are traversed: moving up the network from one BLB towards the root, then moving down to the destination BLB. In this sense, the tree is similar to other multistage networks, like the Beneš network, that rely on an intermediate permutation of the routes in the first half of the network and final routing in the second[135]. More specifically, the existence of 2:1 switchboxes, illustrated in Figure A.4, creates the possibility of conflicts in the network. Furthermore, the "diagonals-only" switchbox design, both 1:1 and 2:1, creates a set of "routing domains". A signal that enters the network on one domain can never jump to another.

We can easily see the flexibility required of the DRAM to BLB interconnect, by inspecting some typical scenaria: First, assume the common case of an 8-bit data stream that has to be connected to the input of an adder. To take advantage of the carry chains in the compute page, the natural layout for an adder is linear, parallel to the CMB interface.

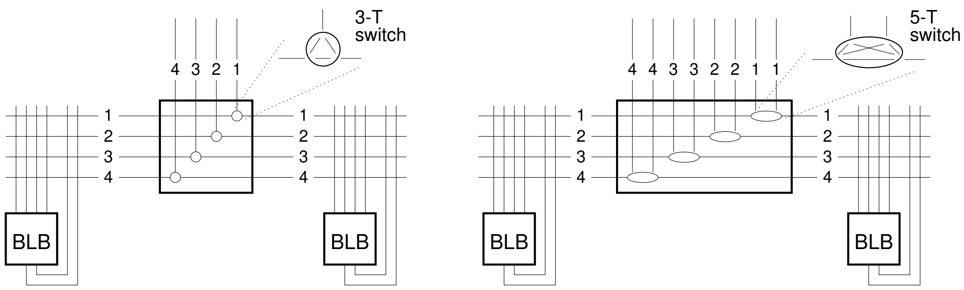

Figure A.4: 1:1 and 2:1 switchbox designs with the routing domains annotated (note there are 10 domains in Trumpet).

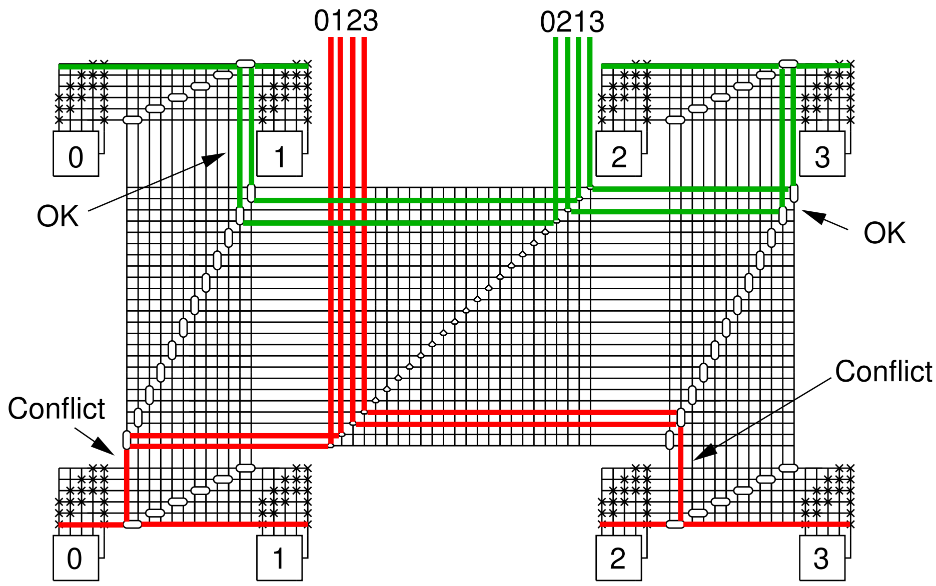

Figure A.5: Conflicts caused by the 2:1 switches in the tree network. On the right, the conflict is avoided by reordering the bits as they enter the network at the root.

In fact, with regular datapath-like structures, large portions of the logic will often be laid out this way. By tracing the lines in Figure A.5 (where only 4 bits are shown for clarity), it is easy to see that a simple linear arrangement of the data bits at the top level of the tree will lead to conflicts at the 2:1 switches further down. Some way to rearrange the bits at that stage is necessary, as demonstrated by the second group of signals on the right side.

As a second example, consider a kernel that searches for a predefined value in an 8-bit data stream. When the compute page configuration can be specialized for this value, the 8-bit comparator can be implemented in only 2 BLB's, linked with the cascade connection. To route 4 data bits to each BLB, we need to route them over at least 4 different domains, because each BLB can only be connected to one wire in each domain. Therefore, some flexibility in choosing the routing domain is required.

Topology

The obvious way to satisfy the above requirements would be to replicate the tree within the CMB. However, the structure must be adapted somewhat. First of all, the 320 wires at the interface, 160 tree wires and 160 shortcuts are clearly a lot more than necessary for the memory interface. To limit the amount of internal wiring, the 320 wires are multiplexed down to two sets of 80 wires each. Of those, one set will be used to carry data and one will be used for address and control bits. Each set is connected to one of the two "crossbar" circuits, data and address. The multiplexer is organized as 80 bit slices, with each slice consisting of two 4:1 single-bit multiplexers, one for address and one for data. Each bit slice has its own configuration bits that control the multiplexers. While the internal wiring is reduced significantly after this multiplexer, some connectivity is compromised: among the four wires of the interface that are grouped together in a bit slice, at most two can be used at the same time, one for data and one for address or control.

In a similar way, on the DRAM macro side the 128 bits of the DQ bus are reduced to 64. The multiplexer control in this case is a bit of the memory address provided by the application, as opposed to the static settings at the compute page side. Between the two multiplexers, the 320:80 at the interface and the 128:64 at the memory side, the data crossbar must interface the 64 data bits to the 80 internal data wires. Since 80 is not a power of 2, a pure tree will not "fit" both sides. Instead, a set of eight 8:10 crossbars expands the 64 data bits to 80, as shown in Figure A.7. Then, three stages of 2:1 switchboxes, similar to those in the compute page, complete the fat tree. It is possible to flatten this design somewhat to improve the latency. In the final design, shown in Figure A.7, the two lowest levels of the tree have been replaced by one level of 4:1 switches, arranged in a diagonal, in the same way as the 1:1 and 2:1 switchboxes in Figure A.4. The resulting network consists of three switch stages instead of four, and the worst-case effective RC, assuming unit switch sizing and ignoring wiring parasitics, is 27% better than that of the non-flattened case. Note that this network does not use any 1:1 switchboxes. In addition, unlike the tree network in the compute page, no shortcuts are used and the 2:1 switch design is missing the "cross" route, since all routes in this case traverse the entire tree, between the leaves (DRAM macro terminals) and the root (CMB I/O's) – so each 2:1 switch consists of 4 transistors and 4 configuration bits instead of 5.

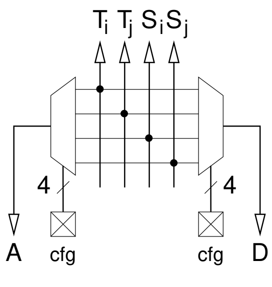

Figure A.6: Bit slice of the 320:80 multiplexer

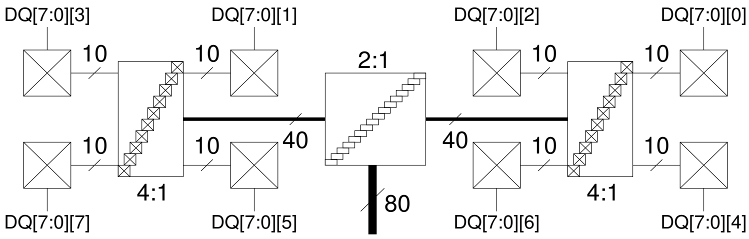

Figure A.7: Data crossbar design. The eight "crossed" boxes represent fully populated 8-to-10 crossbars. The central box is a 2:1 switchbox, similar to the ones in the compute page network, but with 40 inputs and no cross routes in each switch. The two boxes in the middle stage represent a diagonal arrangement of 4:1 crosspoints, similar in topology to the 2:1 switchbox but without the cross route.

The resulting design was still deemed too large. A final way to size it down was to share some of the configuration bits that control the switches at one or more stages. The network, as shown in Figure A.7, requires a total of 1120 pass transistors and corresponding configuration bits. In the final implementation, the left and right half (8:10 and 4:1 stages only) share the same set of configuration bits. In this way the number of configuration bits is reduced to almost half, while the same number of pass transistors is maintained. This of course compromises the connectivity somewhat. Essentially, instead of eight independent 8:10 crossbars we have four 2-bit wide crossbars. In the same way, instead of two 4:1 switchboxes, we only have one, with 2-bit wide paths. This results in data bits traveling in pairs for the first two stages of the network. Bits of a pair can be separated at the final stage, through the 2:1 switchboxes. In this way, every two bits cannot be routed entirely independently, but a number of cases that were routed by hand indicate that this limitation does not create any problems. The final design requires 640 configuration bits.

Handling narrow data

At the logical level, there are separate read and write data buses, the width of which can be independently configured from 0 to 8 bytes. Since the internal network consists of 80 data wires, the combined width of the read and write data buses should not exceed 8 bytes. This is not expected to be a problem for Trumpet, since the amount of logic resources available in the compute page (64 BLB's) are rather limited and it would be difficult to do any useful computation with a 64-bit input and a 64-bit output. For larger devices, where one memory page could interface to a number of compute pages, somewhat richer internal interconnect might be considered. The read and write paths are merged at the 8:10 crossbar stage, the one closest to the DRAM. The word width control also takes place there. Finally, the same crossbar circuits are responsible for part of the addressing: when the read data width is less than 64 bits, the appropriate data bits from the memory side of the crossbar must be addressed and forwarded to the CMB interface. For writes, narrow logical buses are effected by the CMB controller, that drives the macro's DQM signals appropriately based on the least significant bits of the address.

The data bits from the memory side, 8 bytes total, are connected to the data crossbar in the following way: out of each byte, bit i will be connected to the i'th 8:10 crossbar. In this way, each of the eight 8:10 crossbars routes the same-order bits of all the bytes coming out of the memory interface (note the DQ subscripts in Figure A.7). As illustrated in Figure A.8, each terminal on the 10-wide side of the 8:10 crossbar has a 1-hot, 8-bit wide configuration field associated with it, that is used to select the appropriate terminal on the 8-wide side. This field is termed the *Byte Offset*. In addition, a *Read* bit and a *Write* bit are associated with each of the 10 terminals. Wires that are part of the read data bus route have the Read bit set. During a read, the three least significant bits of the address are added to each Byte Offset field. In the i'th 8:10 crossbar, this selects the i'th bit of the memory byte whose address is equal to (Byte Offset) + (Addr) modulo 8. Then, a signal from the controller, gated with the Read bit, enables the appropriate

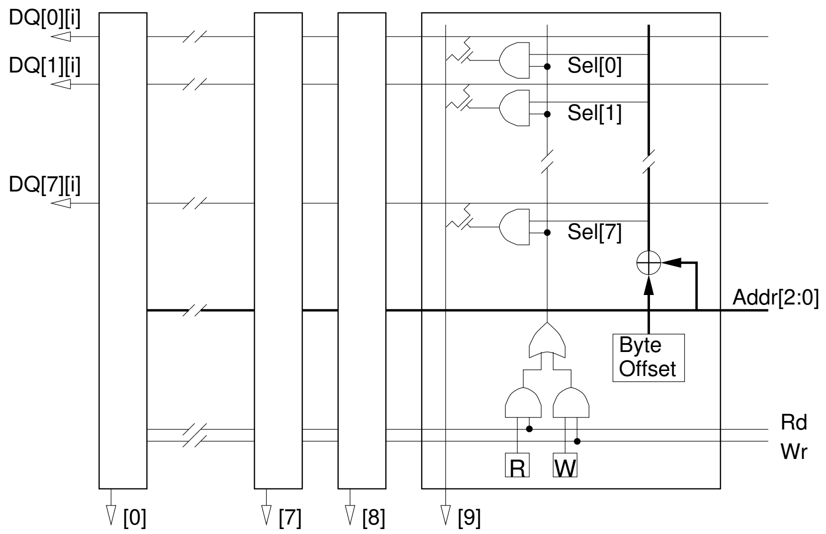

Figure A.8: Detail of the 8:10 stage in the i'th data crossbar connection. The same happens during writes for the wires that are part of the write bus. Wires that are not part of either bus have neither the Read nor the Write bit set and will be tied to ground. By setting the appropriate number of Read and Write bits in the CMB configuration, the required bus widths are implemented.

The address crossbar is similar to the data crossbar, but smaller and somewhat simpler. It is used to route the address and control wires (`Req` and `RWB`), a total of 20 wires maximum, to the CMB controller. The 20 wires on the controller side are rounded up to 24 with the addition of 4 no connects and the resulting 24:80 network is implemented with a topology identical to the data crossbar. However, 3:10 instead of 8:10 crossbars are used for the first stage. Note that since the address/control path is unidirectional, the 3:10 crossbars are simpler than the 8:10 stage in the data crossbar: instead of the Read/Write bits, a single Enable bit enables the appropriate number of wires. Furthermore, there is no need to combine the routing configuration with any address bits, as is the case in the data crossbar. This reduces the 3:10 stages to simple crossbar circuits. The address crossbar consists of 720 pass gates and 400 configuration bits.

The data and address crossbars were laid out in an area of 1.75 by 0.29 mm^2 (data) and 1.75 by 0.17 mm^2 (address), where 1.75 mm is the width of the datapath. Together they take up 33% of the datapath area. Based on `Hspice` simulations, the address crossbar can be traversed in 1ns in the worst case. This is a critical number because the address crossbar is in series with the tag comparator and much of the controller logic, which is likely the critical path of the design. The critical path in the data crossbar consists of the controller driving the address and read/write controls, followed by the data bits traversing the crossbar from the memory to the interface. The controls are heavily loaded by 80 bit slices of datapath and require large buffers on the controller side. This is is also a candidate for the critical path of the entire design, as discussed in Section 3.6.

Retiming

As in the compute page, all CMB inputs include a configurable retiming bank. Each bit can be skewed independently for a number of cycles, in order to compensate for delays sustained through the pipelined interconnect network of the compute page. At the CMB interface all inputs and outputs are registered once. In addition, inputs can be retimed for up to 4 cycles in the retiming bank.

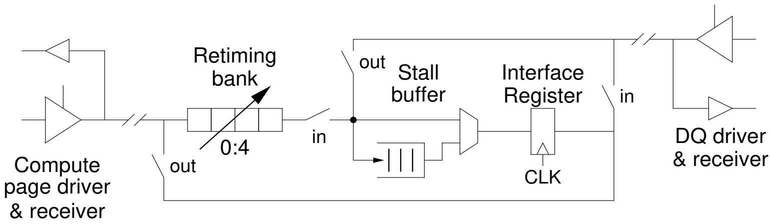

Figure A.9: Bit slice of the retiming bank and stall buffer combination

To reduce the size of the resulting datapath, the retiming bank follows the 320:80 interface multiplexer, so, like for most of the datapath, there are 80 bit slices, with one address and one data bit in each slice. This also turned out to be a convenient place to connect the stall buffers, which follow the same bit-sliced layout. The resulting structure is shown in Figure A.9, where the entire path is one bit wide.

Appendix B

A simple cost model

Die area is commonly used as the sole cost metric in most VLSI-centered cost-performance studies. Unfortunately, this approximation falls short for the purposes of this work. While silicon or die cost, the cost to fabricate the integrated circuit, is still a large contributing factor, other factors, such as packaging and circuit board fabrication and assembly, also contribute to the system cost. In Section 6.5, we compare our single-chip system against conventional implementations with external memory. The multi-chip nature of the conventional implementation incurs additional costs to the system, by requiring more packages and consuming more circuit board real estate. On the other hand, it has the advantage that each of the dies used for logic or memory will likely be smaller than the embedded-memory die, leading to better yields. The tradeoff of die yield against system-level costs is not entirely straightforward. To compare the performance of the two alternatives, single-chip versus multi-chip, on a comparable cost basis, a detailed accounting of all relevant costs is required.

The production costs of a system (or, more accurately, a *sub*-system), from the silicon to the circuit board level, can be broadly categorized into silicon, testing, package and board costs. The subject of this study is assumed to be a product manufactured in high volume. In this way, non-recurring engineering costs can be factored out and only incremental per-unit costs are considered.

Silicon cost

Silicon cost, or the pure manufacturing cost per die, depends strongly on the die area. The average cost per functional die fabricated is the cost per wafer, distributed over all good dies. In other words, the good dies have to shoulder the expense for fabricating the bad ones. Simply expressed,

$$C_{die} = \frac{C_{wafer}}{N_{die} \times Yield}$$

One of the most widely used yield models is the *negative binomial*:

$$Y = (1 + \frac{\lambda A}{\alpha})^{-\alpha}$$

where λ is the defect density and α is the clustering parameter. Typical values of these parameters for current processes are given in SEMATECH's International Technology Roadmap for Semiconductors[112], where a *mature* fabrication process is assumed[1].

With this, the cost equation becomes:

$$C_{die} = \frac{C_{wafer}}{(\frac{A_{wafer}}{A_{die}} - \frac{\pi D_{wafer}}{\sqrt{2A_{die}}}) \times (1+\frac{\lambda A}{\alpha})^{-\alpha}}$$

where the term containing the square root of the die area compensates for the fact that wafers are round but dies are square.

The per-wafer fabrication cost, C_{wafer}, and the yield parameters, λ and α, depend on the fabrication process. Leachman[76] reports an average wafer cost between \$1520 and \$1560 for typical 0.25μm logic processes. Hein[49] estimates a 20-40% wafer cost premium for embedded DRAM processes. Because of the increased number of processing steps, embedded DRAM wafers likely yield less than their logic counterparts, even though part of the above cost differential probably reflects higher effort to limit defect density, so that reasonable yields can be attained. Wafer diameter is currently 200mm. When the transition to 300mm wafers takes place, die costs will be reduced, making the relative contribution of packaging and board costs larger. Unless significant reduction in these cost parameters is achieved, this will make multi chip architectures even less cost-effective.

The silicon cost parameters used are listed in Table B.1.

Testing cost

The cost for testing a silicon die depends primarily on the test time and the capital cost for the purchase of the test equipment. The overall test cost is generally difficult to quantify, because no simple closed form models exist for the length of a set of test vectors.

[1] λ=0.00174 defects/mm^2 and α=5.0, resulting in a yield of 75% for a 170mm^2 chip

	wafer cost	λ	α	yield (170mm^2)	cost (170mm^2)
Logic	\$ 1540	0.175 defects/cm^2	5.0	75%	\$ 13.64
eDRAM	\$ 1925	0.217 defects/cm^2	5.5	70%	\$ 18.27

Table B.1: Silicon cost parameters

With techniques such as scan path and built-in self-test (BIST)[2], test time can be made approximately proportional to the chip complexity, or number of gates. In general, test costs are significantly lower than the fabrication cost, with up to 20-25% of the fabrication cost for testing being typical for most logic chips.

For the purposes of this study, I model the test cost as proportional to the die area, which approximates the number of gates. The constant is chosen to yield costs in the above mentioned range. Simply,

$$C_{test} = \$\ 0.0125\ /\mathrm{mm}^2 \times A_{die}$$

Package cost

Packaging cost depends on the package type and the number of pins. Average package costs per pin can be found in the SEMATECH technology roadmap[112], for various performance and price ranges, corresponding to different package types. For our low-cost and high-performance design points, typical costs of 0.65 and 1.4 cents/pin are used respectively. These correspond to the "low cost" and "cost-performance" categories in the roadmap. While not explicitly specified in the roadmap, these costs probably correspond to plastic and BGA packages.

Table B.2 lists the package costs for the embedded and external memory architectures defined in Section 5.1.

	Embedded	Ext.DRAM	Ext.SRAM
Pins (Small)	136	208 - 232	252 - 306
Cost (Small)	\$ 0.88	\$ 1.35 - 1.51	\$ 1.64 - 1.99
Pins (Large)	220	292 - 364	340 - 394
Cost (Large)	\$ 3.08	\$ 4.09 - 5.10	\$ 4.76 - 5.52

Table B.2: Package costs for the embedded and external-memory devices

Cost of external memory

With the commodity nature of memory chips, their price is strongly influenced by market fluctuations, driven by the laws of supply and demand. In the last few years, weak demand and oversupply have driven DRAM prices down. Data from large IC trading centers[33, 34] show a wide pricing range depending on the interface width and operating frequency. For the devices of interest, 100 MHz 16 Mbit and 133 MHz 64 Mbit SDRAMs, an average spot market price of $3 and $8 is estimated.

The same sources show costs for SRAM about 15 times higher per bit than for DRAM, on average about $2.5 per Mbit, corresponding more or less to the difference in density.

Circuit board cost

Finally, we have to account for the contribution of the chip and possible external memories to the cost of the circuit board. A simple model charges board area proportional to the combined area of all chip packages. This is of course only an approximation. Often there will be a fixed form factor requirement, in which case more chips may force a denser board with more layers but the same area. This leads to a more stepwise dependence of board cost to part count. But since we are interested in general trends, as opposed to a specific design, the linear model is a reasonable approximation.

A number of data points from the industry suggest board fabrication costs upwards of $0.10 per square inch for 2-layer boards, in large volume. In addition, assembly often costs more than fabrication. Therefore, for the low cost design point I assume $\$0.20/\text{in}^2$ for a 2-layer board fabrication and assembly. At the high performance point the figure is $\$0.30/\text{in}^2$, corresponding to a 4-layer board. These numbers represent the lowest data points available, which maintains the conservative character of the results, when the embedded DRAM design is compared to external memory implementations.

Appendix C

Applications

The following sections give a brief overview of the applications used for the optimization study and highlight some of the more interesting implementation and modeling issues.

C.1 Fingerprint matching

This application is a simple instance of the more general pattern matching problem. It is a good starting point because it is simple enough to demonstrate the concepts of the application performance modeling.

The basic algorithm is taken from [100]. A fingerprint record consists of a set of features or *minutiae*, extracted from a binary fingerprint image – typically between 50 and 100 features per record. Each feature is characterized by x and y coordinates and an angle, θ. Given are a large database of fingerprint records and a "query" record, the task is to obtain a measure of distance between the query and each record in the database. This distance is simply defined as the number of features matched between the query and each database record, divided by the number of features in the query and the number of features in the database record. This computation requires an all-to-all comparison between the features of the query and those of each database record. A pair of features is defined to match when all three parameters of one fall within a tolerance cube around the parameters of the other:

$$(x_q - \Delta x \leq x_r \leq x_q + \Delta x) \bigwedge (y_q - \Delta y \leq y_r \leq y_q + \Delta y) \bigwedge (\theta_q - \Delta\theta \leq \theta_r \leq \theta_q + \Delta\theta)$$

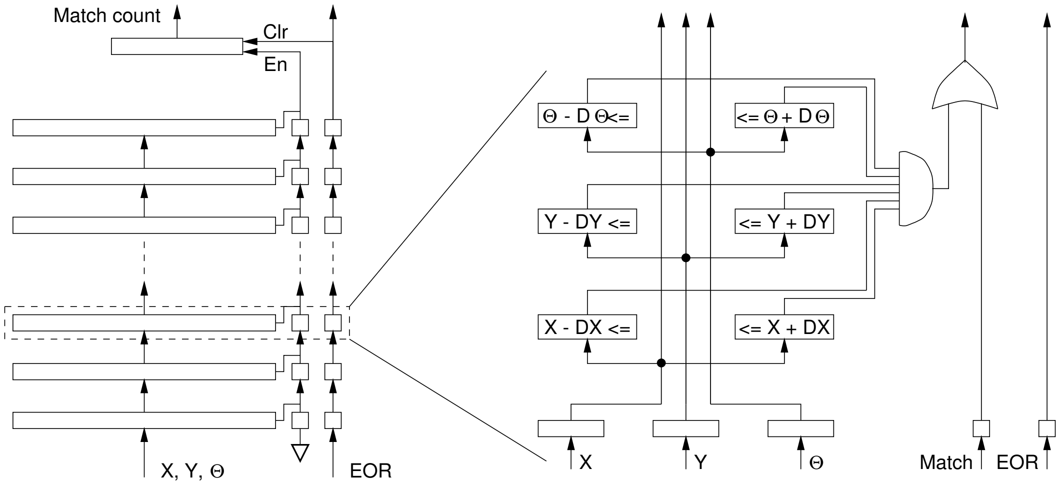

Figure C.1: Fingerprint matching pipeline and feature comparator design

Note that the quantities $x_q \pm \Delta x$, $y_q \pm \Delta y$ and $\theta_q \pm \Delta\theta$ depend only on the query. It is easy then to construct a specialized *feature comparator* for each feature in the query fingerprint. Then, a pipeline of such feature comparators is assembled, with the x, y and θ parameters from the database entering one end and the match counts collected at the other. As shown in Figure C.1, a *match* bit follows the flow and enables a counter at the end. In addition, an *End-Of-Record* (EOR) bit delineates the different database records, by clearing the match counter. In a stream-based environment, the same bit can serve as a data presence bit for the output stream.

With specialization, one feature comparator can be implemented in 16 Trumpet-like BLBs. When all feature comparators for one query, N_q, fit on the device, we achieve a throughput of N_q feature comparisons per cycle and the configuration is static. When we cannot fit all the feature comparators on the array at once, we can break down the pipeline into a number of "kernels", each kernel consisting of a number of feature comparators. All kernels are made of an identical template, with different constants filled in. The outputs of each kernel are buffered into a stream buffer, until that buffer is full. Then, the next kernel takes over. 24 bits per input datum (database feature) need to be buffered between kernels.

With R_{db} records in the database, N_{avg} features per database record on average, and N_q features in the query, the total number of operations is $R_{db} \times N_{avg} \times N_q$ feature comparisons. The division of the match count by the number of features in the two fingerprint records, $(N_q \times N_{db})$, is carried out at the host. With $P \times N_{BLB}$ total BLBs in the device, we can execute $P \times N_{BLB}/16$ feature comparisons per cycle within each kernel execution. The number of kernels alternating within each computation epoch will be equal to the total number of feature comparators, N_q, divided by the number of comparators that can fit in the array at any one time, $P \times N_{BLB}/16$. Because of the stream buffering requirement, the amount of data (database features) processed within each epoch will be the total stream buffer space divided by 24 bits. Then, we have $R_{db} \times N_{avg} \times N_q \times 24/S_{buf}$ epochs altogether, where S_{buf} is the total stream buffer size in bits. This is equal to the total memory capacity, $P \times S_{CMB}$, less the storage required for the different kernel configurations (which in turn depends on the number of BLBs). Finally, the total reconfiguration time is

(Number of epochs) $\times$ (Number of kernels) $\times$ (Single configuration time)

or

$$\frac{R_{db} \times N_{avg} \times N_q \times 24}{S_{buf}} \times \frac{N_q \times 16}{P \times N_{BLB}} \times T_{cfg}$$

and the total computation time is

(Number of operations) / (Operations per cycle)

or

$$(R_{db} \times N_{avg} \times N_q) \times \frac{16}{P \times N_{BLB}}$$

The total run time is the sum of the above.

C.2 JPEG compression

JPEG[134] is a very common still image compression standard, used for image storage and transmission in diverse environments, such as desktop computers and digital cameras. It consists of five basic kernels, that follow each other in succession:

- Color space conversion: the input image, usually represented in the RGB color space, is transformed to the YCbCr space with a 3-by-3 linear transformation. This separates the brightness information (Y component) from the color information (Cb and Cr components).

- Downsampling: the color planes, being less critical to the visually accurate representation of the image, are downsampled in both dimensions by a 2:1 rate, reducing the amount of information in each color plane by a factor of 4. The Y plane is kept in the original resolution.

- DCT transform: 8-by-8 blocks of each color space are subject to the 2-dimensional DCT transform. This extracts the spatial frequencies from the original pixel data.

- Quantization: The spatial frequency data (DCT coefficients) are quantized down to a smaller number of bits. Data that correspond to higher frequencies are deemed less critical to the accuracy of the resulting image and are quantized to fewer bits than low frequency data. A *quantization table* provides the 64 coefficients that correspond to the 64 spatial frequencies.

- Entropy coding: At this point, there are many zeroes among the quantized DCT coefficients. Each 8-by-8 DCT block is scanned in a zig-zag order (accessing the low frequency components before the high frequency ones) and zero-length coded. The resulting data are mapped to a variable-length Huffman code, based on the frequencies of individual data words. The sequence of Huffman codes plus certain format bits form the final output of the algorithm.

The multiple kernels involved in this application pose varying requirements on the memory architecture of the device. Trying to optimize for the entire application has a nice "averaging" effect on the parameters chosen. Some of the more interesting implementation issues are highlighted in the following.

Color space conversion: This kernel consists of three dot products per pixel, all fixed point, three element long, with constant coefficients. Among several possible implementations, the distributed arithmetic[136] was chosen because of its good efficiency and graceful scaling when variable amounts of resources are available. The distributed arithmetic algorithm is based on precomputed sums of the fixed coefficients that are stored in a ROM. The variable inputs (in our case, the pixel data) are accessed serially, LSB first. The 3-bit quantities formed by the same bit position, or bit plane, of each input value are used to index this ROM. Because such ROMs are small (8 words by 56 bits for 16-bit coefficients), it would be a waste of resources to store them in CMBs, so they are implemented with BLBs. 27 BLBs are needed for one such table.

A basic implementation, as well as one with twice the throughput, are illustrated in Figure C.2. 100 BLBs and one byte of memory I/O are required for the baseline implementation, which processes one bit plane at a time and takes 8 cycles to process one pixel. With increasing amounts of available BLBs and a wider memory interface, more parallel

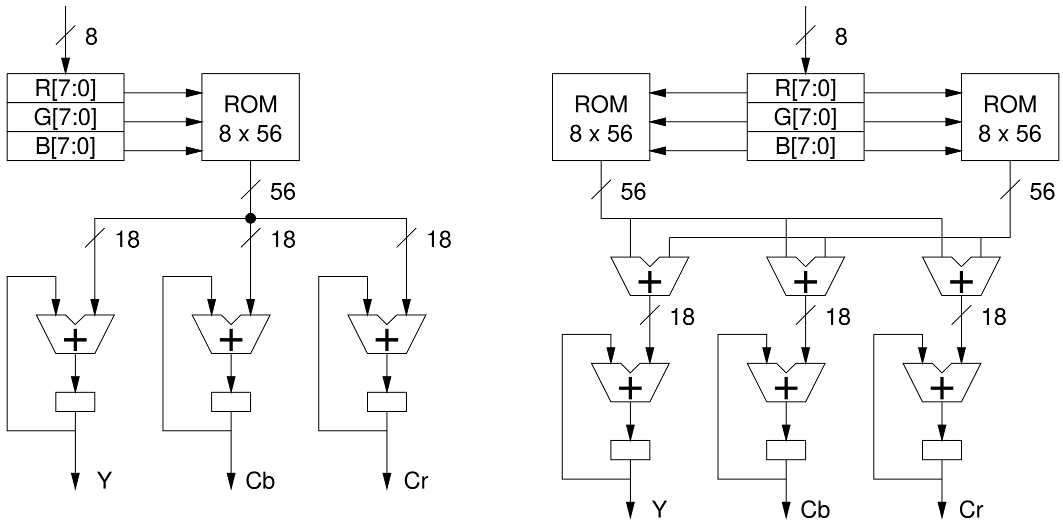

Figure C.2: Color space conversion datapaths

implementations are possible, generated by replicating the ROM table, constructing a summation tree and processing multiple bit planes at a time. An 8X parallel implementation requires 601 BLBs and 24-bit memory I/O and fully processes one pixel per cycle.

Downsampling and data layout: The compute kernels that follow downsampling access the data in 8-by-8 blocks. The obvious row-major data layout will cause a large number of misses in the DRAM for most of these kernels. For others, enough time is available during the processing of one row to precharge and activate the next (bank ping-pong), but this has the disadvantage of very frequent bank activations, that wastes energy. To avoid both misses and energy waste, care must be taken at the downsampling output to reorder data appropriately. This means that the downsampling output is not a stream. Instead, random access in needed at the output buffer.

First, different color spaces are separated, resulting in three different pixel arrays per image. Second, 8-by-8 blocks in each color space are "collected" into contiguous memory areas. With 16 bits per pixel, an 8-by-8 block is 1024 bits long, small enough to fit in a memory page in most DRAM organizations. This reordering penalizes the downsampling kernel with the energy required to cycle DRAM banks every 8 input elements, but this penalty is paid only once, while larger savings arise in subsequent kernels. In addition, when a large number of BLBs is available, downsampling may proceed at a high enough throughput that does not leave enough time to ping pong a pair of DRAM banks. In that case, some misses are sustained, diminishing the benefit of large devices. Still, this reordering leads to much larger performance benefits in the following kernels.

Entropy coding: This is a rather irregular control-dominated kernel. The input is scanned in a zig-zag order. Again, this does not strictly fit in the stream formalism and requires a small address generator of moderate complexity. Misses in the input buffer are avoided, since each 8-by-8 block occupies no more than a memory page. Part of the process is a lookup into a table that contains the Huffman codes. This table can be stored in 3 Kbits. With a 1Kbit memory page and 2 banks, it is not possible to fit the entire table in sense amplifier SRAM at the same time. Profiling of a software implementation showed that the upper third of the table is accessed very rarely (codes are stored in order of frequency), so misses are rare (under 1% miss rate). Still, there is a certain implementation complexity required to handle the misses, no matter how rare. A few options are:

- More sense amplifiers per memory page. Either wider pages, or more banks.
- More than one memory pages allocated for the table.
- SRAM-like memory access mode (as in Trumpet) that essentially performs the row/column operations in custom hardware.
- Maintaining a cache tag and scheduling row and column accesses in the user logic.

In this study, both the SRAM and DRAM access modes are assumed available, so the SRAM mode can be used to handle the irregular access pattern.

C.3 Wavelet compression

This application is a more recent image compression algorithm, with the fundamental difference from JPEG being that the localized DCT transform step is replaced by an image-wide wavelet transform. This computation requires global accesses, so it is a little more complicated than JPEG. Following the transform are quantization and entropy coding steps similar to those of JPEG. No standard exists yet, so a number of variations exist for each step of the algorithm. Wavelet compression is used as the "versatility stressmark"[55], part of the configurable computing benchmark suite developed by Honeywell. As specified in the stressmark, the algorithm is applied to 512×512 grey scale images.

The most interesting part of the algorithm is the wavelet transform. The wavelet datapath is quite simple, consisting of a pair of simple digital filters. The full transformation

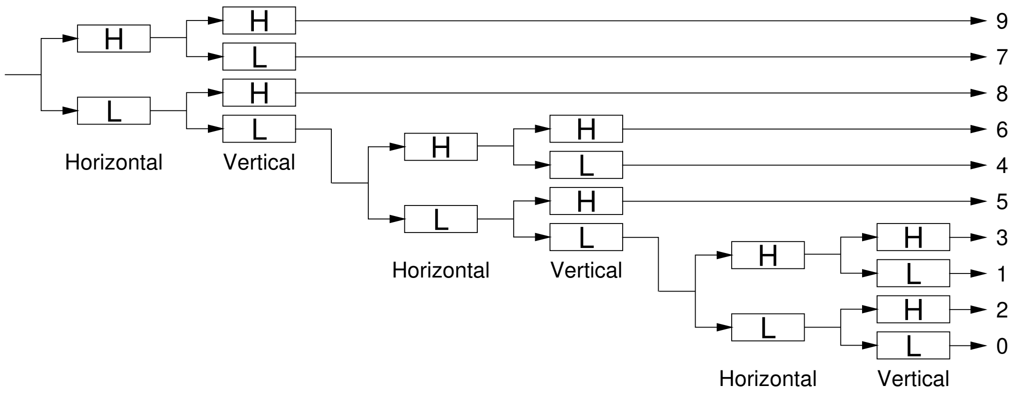

Figure C.3: 3-pass recursive wavelet transform

consists of three steps, with each step having a horizontal and a vertical pass. Each step decomposes its input into four quadrants, corresponding to high and low horizontal and vertical spatial frequencies. Each step is applied to the "Low X – Low Y" output of the previous step, decomposing the image to a "multiresolution" representation. Ten different blocks result, of varying size and resolution. Different quantization and entropy coding parameters will be applied to each block, based on its contribution to the perceptual quality of the resulting image.

The ten output streams shown in Figure C.3 create an interesting data layout issue. To share hardware resources and to avoid multiple passes over each data stream, each pair of horizontal and vertical filters is typically implemented as one unit with two outputs. High and low frequency outputs come out of each filter bank interleaved. With reconfiguration between every step and every pass (horizontal/vertical), and with appropriate mapping of image rows to memory banks, it is possible to write out these outputs in the "natural" blocked data layout shown in Figure C.4, with two buffers used in a ping-pong fashion. Mapping of image rows to memory banks to avoid misses is somewhat involved, but possible if at least two memory pages are used per buffer, with at least two banks per page. In limited

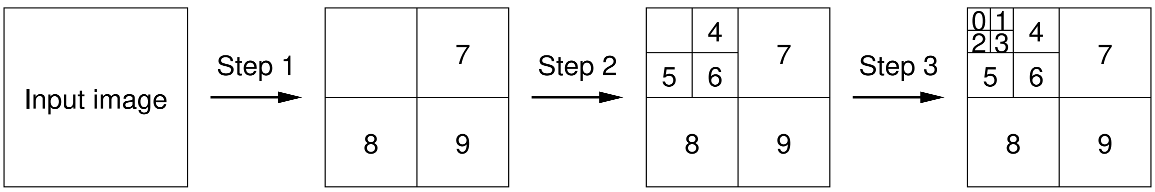

Figure C.4: Multiresolution decomposition of an input image memory devices, a better option is to do the computation in place, reducing the memory requirements in half and so making reconfigurations less frequent. In that case, the datapath must follow a read-ahead, write-behind design, with the high and low frequency outputs stored interleaved, in the order they come out of the filter banks. In two dimensions, this creates a checkerboard pattern at the output instead of the natural block layout. Again, to avoid frequent misses in the DRAM, proper mapping of image rows to memory banks must be done, which can be done when at least two memory pages are used. The checkerboard pattern necessitates strided accesses for the next step, which may limit the parallelism that can be achieved in the second and third steps. Depending on the amount of memory available and the cost of each reconfiguration, one or the other approach, buffer ping-pong or in-place computation, may be better. The optimization program considers both options and picks the one with the least total cost.

A basic horizontal and vertical filter datapath can be implemented in about 85 BLBs, requiring 16-bit memory I/O. To scale up from there, one can either implement wider datapaths that process more than one pixels at a time, or cascade multiple filter stages. Because of the "corner turn" problem, the switching between row-major and column-major accesses in successive passes, cascading two filter passes requires two full line buffers between stages and is quite expensive. For example, the line buffer between the first horizontal and vertical pass would take 640 BLBs in the register-rich HSRA. Using memory would also require two memory pages, because of the concurrent reads and writes required. For this reason, only the wide datapath option is considered for scaling performance. As we see in Section 6.5, things are considerably different when it comes to the external memory implementation.

C.4 Speech recognition

Speech recognition is becoming more popular as a user interface paradigm and will likely become a standard part of handheld devices soon. A large body of research exists on different algorithms for speech recognition. While continuous speaker-dependent recognition in a controlled environment can be performed on a desktop fairly well today, the more general problem of speaker-independent recognition in a noisy environment is still an area of active research, in both fundamental algorithms and implementations.

In this work, we study a simple instance of one of the more established architectures, the hybrid connectionist/HMM approach[84]. In brief, the audio input is divided in 10ms frames and a signal processing front-end is used to extract a number of spectral *features* from each frame. These features are then processed with a 3-layer neural network that has been trained to recognize the various *phones*, or sounds, of a language. The output of the neural network at each time frame can be interpreted as a vector of probabilities: each element of this vector is the probability that the input in the current time frame represents a particular phone. These phone probabilities are further processed with a dynamic programming algorithm, that finds the most likely sequence of phones, based on the sequence of probability vectors and constraints from a set of acoustic word models (the vocabulary) and possibly a language model, or grammar. The Viterbi algorithm[35] is one of the most common implementations of this final step.

The computation in the front end and the neural network stage are dominated by multiplications. Because of the large number of coefficients in the neural network (several hundred thousand in practical systems such as the one described in [84]), specialization is not practical and general multipliers must be used. This type of computation does not map very well to reconfigurable fabrics and could be executed much more efficiently on the host processor. Therefore I make the assumption that only the dynamic programming step is performed in the reconfigurable array. Processor and array are running concurrently, with a new set of phone probability vectors provided to the array every 10ms.

The sequence decoding is based on the *Hidden Markov Model* (HMM) acoustic word models. An HMM represents a word as a sequence of states. Transitions between states happen with certain probabilities, while each state is associated with a certain phone. The set of states, associated phones and transition probabilities for each word make up the HMM for that word. The set of HMMs for all words constitutes the *dictionary*, or *vocabulary*. The task of the dynamic programming is to find the most likely sequence of states within each utterance, given the sequence of input probability vectors and the relevant transition probabilities. This computation is equivalent to finding the shortest path through the lattice shown in Figure C.5, where the horizontal axis is time. The "distance" of each link corresponds to the likelihood of the corresponding transition. In addition, a "local cost" associated with each node at each time step represents the likelihood that the input at that time step matches the corresponding phone. At each time step, the output of the neural network provides the vector of local costs. Every time step, a new *score*, or likelihood, is

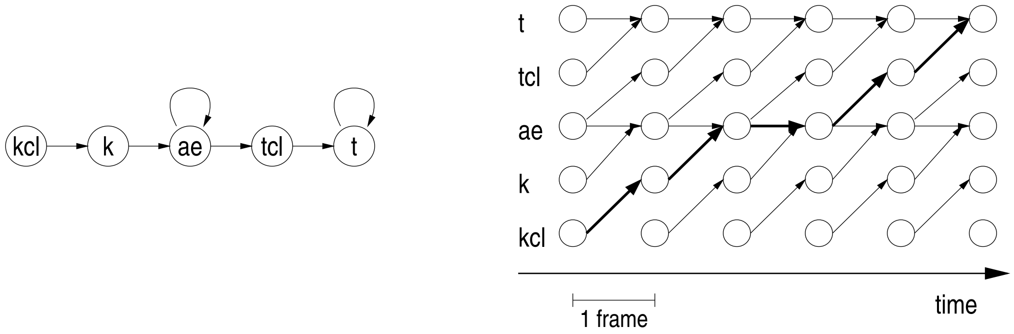

Figure C.5: Hidden Markov Model and state decoding

computed for each state in each word[1], based on the current scores of its successors and the appropriate transition probabilities. When the score of the final state of any word exceeds a certain threshold, the word is said to have been recognized. A large data structure, the *path memory*, is used to store the scores of all states in the vocabulary.

The Viterbi implementation has certain interesting features: first, unlike the other applications, it requires large data structures, the dictionary and path memory. For a 25,000 word vocabulary, approximately 24 Mbits and 8 Mbits are required for the dictionary and the path memory respectively. This requirement will skew the required balance of logic and memory resources to more memory and less logic. Second, while the accesses to the dictionary are read-only, the ones to the path memory are read-write. This exposes the memory latency every time we switch between reads and writes. To update the score of each state, the scores of its predecessors must be read from the path memory. With an average state in-degree of two, this amounts to approximately two reads for every write. There are a number of ways to mitigate this effect: if enough memory pages are available, it is possible to double-buffer the path memory, so reads and writes are separated. Alternatively, the computation may be broken down in two kernels, with only reads in the first, mostly writes on the second and a stream buffer between the two kernels. In spite of the 2-pass character of this approach, it is surprisingly competitive, because of the segregation of the reads and writes. Whether this implementation is a win over the baseline, depends on the amount of available memory and the resulting reconfiguration frequency. This number of options highlights the need for multiple implementations and on-line selection during the optimization process.

[1] actually, a subset of "active" words, based on a *pruning* criterion

Finally, because of the irregular structure and variable length of each word model, it is difficult to achieve parallelism within each word. It is much more effective to use *outer-loop parallelism*, that is process different words in parallel. For this to be possible, both the dictionary and path memories must be distributed over as many memory pages as possible. Then, independent datapaths can perform the computation on each fraction of the vocabulary. This approach performs best when many independent memory pages are available.

Because of the prominence of the dictionary and path memory, different vocabulary sizes are used for the low cost and high performance design points: 5,000 and 25,000 words respectively. To keep the total workload similar, the length of the input is varied from 1200 frames in the low cost model down to 240 frames in the high performance model. This corresponds to 12 and 2.4 seconds of speech respectively.

Bibliography

[1] A.L. Abbott, P.M. Athanas, L. Chen, and R.L. Elliott. "Finding Lines and Building Pyramids with Splash 2". In *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, USA, 10-13 April 1994. pp. 155–163.

[2] M. Abramovici, M.A. Breuer, and A.D.Friedman. *Digital Systems Testing and Testable Design*. IEEE Press, 1990.

[3] Silicon Access. `http://www.siliconaccess.com/`.

[4] A. Ahi, A. Bomdica, G. Shippen, et al. "R10000 Superscalar Microprocessor". In *Hot Chips VII*, Stanford, CA, USA, August 13-15 1995.

[5] J.H. Ahn, T.H. Kim, S.M. Park, et al. "Bidirectional Matched Global Bit Line Scheme for High Density DRAMs". In *IEEE Symposium on VLSI Circuits*, Tokyo, Japan, 19-21 May 1993. pp. 91–92.

[6] C.J. Alpert and A.B. Kahng. "Recent Directions in Netlist Partitioning: a Survey". *Integration, the VLSI Journal*, 19(1-2):1–81, August 1995.

[7] Altera. "FLEX 10K Embedded Programmable Logic Family Databook". `http://www.altera.com/documents/ds/dsf10k.pdf`.

[8] R. Amerson, R.J. Carter, W.B. Culbertson, P. Kuekes, and G. Snider. "Teramac – Configurable Custom Computing". In *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, USA, 19-21 April 1995. pp. 32–38.

[9] J.M. Arnold, D.A. Buell, D.T. Hoang, et al. "The Splash 2 Processor and Applications". In *IEEE International Conference on Computer Design*, Cambridge, MA, USA, 3-6 October 1993. pp. 482–485.

[10] M. Asakura, T. Ohishi, M. Tsukude, et al. "A 34ns 256Mb DRAM with Boosted Sense-Ground Scheme". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 16-18 February 1994. pp. 140–141.

[11] P. Barnes. "A 500 MHz 64b RISC CPU with 1.5MB On-chip Cache". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 15-17 February 1999. pp. 86–87.

[12] P. Bertin, D. Roncin, and J. Vuillemin. "Introduction to Programmable Active Memories". In *International Conference on Systolic Arrays*, Killarney, Ireland, 31 May – 2 June 1989. pp. 301–309.

[13] T.N. Blalock and R.C. Jaeger. "An Experimental 2T Cell RAM with 7ns Access Time at Low Temperature". In *IEEE Symposium on VLSI Circuits*, Honolulu, HI, USA, 7-9 June 1990. pp. 13–14.

[14] N. Bowman, N. Cardwell, C.E. Kozyrakis, C. Romer, and H. Wang. "Evaluation of Existing Architectures on IRAM systems". In *Workshop on mixing logic and DRAM. International Symposium on Computer Architecture*, Denver, CO, USA, 1 June 1997.

[15] G. Braceras, D. Evans, J. Sousa, and J. Conner. "A 350MHz 3.3V 4Mb SRAM Fabricated in a 0.3μm Process". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 6-8 February 1997. p. 404.

[16] M. Budiu and S.C. Goldstein. "Fast Compilation for Pipelined Reconfigurable Fabrics". In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, USA, 21-23 February 1999. pp. 195–205.

[17] D.A. Buell, J.M. Arnold, and W.J. Kleinfelder, editors. *Splash 2. FPGAs in a Custom Computing Machine.* IEEE Computer Society Press, 1996.

[18] D. Burger, J.R. Goodman, and A. Kagi. "Memory Bandwidth Limitations of Future Microprocessors". In *Proceedings, International Symposium on Computer Architecture*, Philadelphia, PA, USA, 22-24 May 1996. pp. 78–89.

[19] T.J. Callahan, J.R. Hauser, and J. Wawrzynek. "The Garp Architecture and C Compiler". *IEEE Computer*, 33(4):62–69, April 2000.

[20] W.S. Carter, K. Duong, R.H. Freeman, et al. "A User Programmable Reconfigurable Logic Array". In *IEEE Custom Integrated Circuits Conference*, Rochester, NY, USA, 12-15 May 1986. pp. 233–235.

[21] E. Caspi, M. Chu, R. Huang, et al. "Stream Computations Organized for Reconfigurable Execution (SCORE): Introduction and Tutorial". In *Proceedings, Field-Programmable Logic and Applications*, Villach, Austria, 28-30 August 2000.

[22] A. Cataldo and R. Wilson. "NEC Turns Virtual Channel on Memory Latencies". *EE Times*, November 17 1997.

[23] M. Chu, N. Weaver, K. Sulimma, A. DeHon, and J. Wawrzynek. "Object Oriented Circuit Generators in Java". In *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, USA, 15-17 April 1998. pp. 158–166.

[24] Digital Equipment Corp. "PCI Pamette V1". `http://www.research.digital.com/SRC/pamette/`.

[25] A.B. Cosoraba. "Double Data Rate Synchronous DRAMs in High Performance Applications". In *WESCON*, 1997, pp. 387–391.

[26] S. Crowder, R. Hannon, H. Ho, et al. "Integration of Trench DRAM into a High-Performance 0.18μm Logic Technology with Copper BEOL". In *International Electron Devices Meeting*, San Francisco, CA, USA, 6-9 December 1998. pp. 1017–1020.

[27] M.F. Deering, S.A. Schlapp, and M.G. Lavelle. "FBRAM: A new Form of Memory Optimized for 3D Graphics". In *Proceedings, International SIGGRAPH Conference*, Orlando, FL, USA, 24-29 July 1994. pp. 167–174.

[28] A. DeHon. *Reconfigurable Architectures for General-Purpose Computing*. PhD thesis, MIT, September 1996.

[29] A. DeHon. "The Density Advantage of Configurable Computing". *IEEE Computer*, 33(4):41–49, April 2000.

[30] K. Diefendorff and P.K. Dubey. “How Multimedia Workloads Will Change Processor Design”. *IEEE Computer*, 30(9):43–45, September 1997.

[31] M. Disman. “Stalking the Chameleon Computer”. *OEM Magazine*, pp. 67–73, December/January 1996.

[32] K. Dosaka, Y. Konishi, K. Hayano, et al. “A 100-MHz 4-Mb Cache DRAM with Fast Copy-Back Scheme”. *IEEE Journal of Solid State Circuits*, 27(11):1534–1539, November 1992.

[33] American IC Exchange. `http://www.aice.com`.

[34] NECX Global Electronics Exchange. `http://webexchange.necx.com`.

[35] G.D. Forney, Jr. “The Viterbi Algorithm”. *Proceedings of the IEEE*, 61(3):268–278, March 1973.

[36] R.C. Foss. “Implementing application specific memory”. In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 8-10 February 1996. pp. 260–261.

[37] R.C. Foss, J. Wu, J. Benzreba, et al. “Re-inventing the DRAM for Embedded Use: A Compiled Wide-Databus DRAM Macrocell with High Bandwidth and Low Power”. In *IEEE Custom Integrated Circuits Conference*, Santa Clara, CA, USA, 11-14 May 1998. pp. 283–286.

[38] R. Fromm, S. Perissakis, N. Cardwell, et al. “The Energy Efficiency of IRAM Architectures”. In *Proceedings International Symposium on Computer Architecture*, Denver, CO, USA, 2-4 June 1997. pp. 327–337.

[39] P. Gillingham, B. Hold, I. Mes, et al. “A 768k embedded DRAM for 1.244Gb/s ATM switch in a 0.8μm Logic Process”. In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 8-10 February 1996. pp. 262–263.

[40] P.N. Glaskowsky. “MoSys Explains 1T-SRAM Technology”. *Microprocessor Report*, 13(12), September 13 1999.

[41] M. Gokhale, W. Holmes, A. Kopser, et al. “Building and Using a Highly Parallel Programmable Logic Array”. *IEEE Computer*, 24(1):81–89, January 1991.

[42] R. Gonzalez and M. Horowitz. "Energy Dissipation in General Purpose Microprocessors". *IEEE Journal of Solid State Circuits*, 31(9):1277–1284, September 1996.

[43] J. Greason, D. Buehler, J. Kolousek, et al. "A 4.5 Megabit, 560MHz, 4.5GByte/s High Bandwidth SRAM". In *IEEE Symposium on VLSI Circuits*, Tokyo, Japan, 12-14 June 1997. pp. 15–16.

[44] R.I. Greenberg. "The Fat-Pyramid: A Robust Network for Parallel Computation". In *Advanced Research in VLSI*, 1990, pp. 195–213.

[45] M. Hashimoto, K. Abe, and A. Seshadri. "An Embedded DRAM Module using a Dual Sense Amplifier Architecture in a Logic Process". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 6-8 February 1997. pp. 64–65.

[46] S. Hauck and G. Borriello. "An Evaluation of Bipartitioning Techniques". *IEEE Transactions on Computer Aided Design*, 16(9):849–866, August 1997.

[47] S. Hauck, T.W. Fry, M.M. Hosler, and J.P. Kao. "The Chimaera Reconfigurable Functional Unit". In *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, USA, 16-18 April 1997. pp. 87–96.

[48] J.R. Hauser and J. Wawrzynek. "Garp: A MIPS Processor with a Reconfigurable Coprocessor". In *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, USA, 16-18 April 1997. pp. 12–21.

[49] S. Hein and N. Wehn. "Embedded DRAM Applications and Challenges". In *Tutorial: Embedded Memories in System Design. Design Automation Conference*, 1999.

[50] J. Hennessy, N. Jouppi, F. Baskett, et al. "MIPS: A VLSI Processor Architecture". In *CMU Conference on VLSI Systems and Computations*, Rockville, MD, USA, 19-21 October 1981. pp. 337–346.

[51] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach, 2nd edition.* Morgan Kaufmann, 1996.

[52] J. Hensley, M. Oskin, D. Keen, L-V. Lita, and F.T. Chong. "Active Page Architectures for Media Processing". In *First Workshop on Media Processors and DSPs, 32nd Annual Symposium on Microarchitecture*, November 1999.

[53] B. Von Herzen. “Signal Processing at 250 MHz using High-Performance FPGA’s”. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, USA, February 1997.

[54] M. Heshami and B.A. Wooley. “A 250-MHz Skewed-Clock Pipelined Data Buffer”. *IEEE Journal of Solid State Circuits*, 31(3):376–383, March 1996.

[55] “Benchmark Specification Document: Versatility Stressmark”. Technical Report CDRL A001, Honeywell, November 4 1997.

[56] M.A. Horowitz. “Merged DRAM/Logic”. In *Short course: DRAMs in the 21st Century. International Electron Devices Meeting*, San Francisco, CA, USA, 6-8 December 1996.

[57] IBM. *ASIC SA-27E Databook, Rev.2.* 1999.

[58] Rambus Inc. “Rambus Technology Overview”. `http://www.rambus.com/docs/techover.pdf`.

[59] Xilinx Inc. “Virtex Power Estimator”. `http://www.xilinx.com/support/techsup/powerest`, 1999.

[60] Infineon. “Embedded DRAM Solutions”. `http://www.infineon.com/edram/`.

[61] K. Inoue, H. Nakamura, H. Kawai, et al. “A 10Mb 3D Frame Buffer Memory with Z-Compare and A-Blend Units”. In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 15-17 February 1995. pp. 302–303.

[62] K. Ishibashi, K. Komiyaji, H. Toyoshima, et al. “A 300MHz 4-Mb Wave-Pipelined CMOS SRAM Using a Multi-Phase PLL”. In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 15-17 February 1995. pp. 308–309.

[63] K. Itoh, K. Sasaki, and Y. Nakagome. “Trends in Low-Power RAM Circuit Technologies”. *Proceedings of the IEEE*, 83(4):524–543, April 1995.

[64] S. Iwata, T. Shimizu, J. Korematu, et al. “Performance evaluation of a microprocessor with on-chip DRAM and high bandwidth internal bus”. In *IEEE Custom Integrated Circuits Conference*, San Diego, CA, USA, 5-8 May 1996. pp. 269–272.

[65] S.S. Iyer and H.L. Kalter. "Embedded DRAM Technology: Opportunities and Challenges". *IEEE Spectrum*, 36(4):56–64, April 1999.

[66] H.L. Kalter, C.H. Stapper, J.E. Barth, et al. "A 50-ns 16-Mb DRAM with a 10-ns Data Rate and On-Chip ECC". *IEEE Journal of Solid State Circuits*, 25(5):1118–1128, October 1990.

[67] R.E. Kessler. "The Alpha 21264 Microprocessor". *IEEE Micro*, 19(2):24–36, March-April 1999.

[68] T. Kirihata, S.H. Dhong, K. Kitamura, et al. "A 14-ns 4-Mb CMOS DRAM with 300-mW Active Power". *IEEE Journal of Solid State Circuits*, 27(9):1222–1227, September 1992.

[69] G. Kitsukawa, K. Yanagisawa, Y. Kobayashi, et al. "A 23-ns 1-Mb BiCMOS DRAM". *IEEE Journal of Solid State Circuits*, 25(5):1102–1109, October 1990.

[70] H. Koga, T. Matsuki, N. Kasai, et al. "Two-Dimensional Borderless Contact Pad Technology for a 0.135 μm^2 4-Gigabit DRAM Cell". In *International Electron Devices Meeting*, Washington, DC, USA, 7-10 December 1997. pp. 25–28.

[71] H. Kotani, H. Akamatsu, Y. Naito, et al. "A 256Mb DRAM with 100MHz Serial I/O Ports for Storage of Moving Pictures". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 16-18 February 1994. pp. 142–143.

[72] C.E. Kozyrakis and D.A. Patterson. "A New Direction for Computer Architecture Research". *IEEE Computer*, 31(11):24–32, November 1998.

[73] C.E. Kozyrakis, S. Perissakis, D.A. Patterson, et al. "Scalable Processors in the Billion-Transistor Era: IRAM". *IEEE Computer*, 30(9):75–78, September 1997.

[74] S. Kuhara, H. Toyoshima, K. Takeda, et al. "A 6ns 1.5V 4Mb BiCMOS SRAM". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 8-10 February 1996. pp. 144–145.

[75] B.S. Landman and R.L. Russo. "On a Pin Versus Block Relationships For Partitions of Logic Graphs". *IEEE Transactions on Computers*, C20(12):1469–1479, December 1971.

[76] R.C. Leachman, J. Plummer, and N. Sato-Misawa. "Understanding Fab Economics". Technical Report ESRC 99-05, Engineering Systems Research Center, UC Berkeley, June 1999.

[77] D.D. Lee and R.H. Katz. "Non-refreshing Dynamic RAM for On-chip Cache Memories". In *IEEE Symposium on VLSI Circuits*, Honolulu, HI, USA, 7-9 June 1990. p. 111.

[78] E. Lemoine and D. Merceron. "Run Time Reconfiguration of FPGA for Scanning Genomic Databases". In *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, USA, 19-21 April 1995. pp. 90–98.

[79] N. Margulis. "The Intel 80860". *Byte Magazine*, pp. 333–340, December 1989.

[80] Y. Matsui, K. Sakakibara, A. Yamamoto, et al. "64Mbit Virtual Channel Synchronous DRAM". *NEC Research and Development*, 40(3):282–286, July 1999.

[81] S. Eto M. Matsumiya, M. Takita, et al. "A 1Gb SDRAM with Ground Level Precharged Bitline and Non-Boosted 2.1V Word Line". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 5-7 February 1998. pp. 82–83.

[82] S. McGeady. "A Programmer's View of the 80960 Architecture". In *COMPCON*, San Francisco, CA, USA, 27 February – 3 March 1989. pp. 4–9.

[83] G.E. Moore. "Cramming More Components onto Integrated Circuits". *Electronics*, pp. 114–117, April 19 1965.

[84] N. Morgan and H. Bourlard. "Continuous Speech Recognition". *IEEE Signal Processing Magazine*, 12(3):24–42, May 1995.

[85] C.A. Moritz, D. Yeung, and A. Agarwal. "Exploring Optimal Cost-Performance Designs for Raw Microprocessors". In *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, USA, 15-17 April 1998. pp. 12–27.

[86] M. Motomura, Y. Aimoto, A. Shibayama, Y. Yabe, and M. Yamashina. "An Embedded DRAM-FPGA Chip with Instantaneous Logic Reconfiguration". In *IEEE Symposium on VLSI Circuits*, Kyoto, Japan, 12-14 June 1997. pp. 55–56.

[87] T. Nagai, K. Numata, M. Ogihara, et al. "A 17-ns 4-Mb CMOS DRAM". *IEEE Journal of Solid State Circuits*, 26(11):1538–1543, November 1991.

[88] K. Nakamura, K. Takeda, H. Toyoshima, et al. "A 500MHz 4Mb CMOS Pipeline-Burst Cache SRAM with Point-to-Point Noise Reduction Coding I/O". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 6-8 February 1997. pp. 406–407.

[89] M. Nakamura, T. Takahashi, T. Akiba, et al. "A 29ns 64Mb DRAM with Hierarchical Array Architecture". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 15-17 February 1995. pp. 246–247.

[90] H. Nambu, K. Kanetani, K. Yamasaki, et al. "A 1.8ns Access, 550MHz 4.5Mb CMOS SRAM". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 5-7 February 1998. pp. 360–361.

[91] T. Namekawa, S. Miyano, R. Fukuda, et al. "Dynamically Shift-Switcehd Dataline Redundancy Suitable for DRAM Macro with Wide Data Bus". In *IEEE Symposium on VLSI Circuits*, Kyoto, Japan, 17-19 June 1999. pp. 149–152.

[92] I. Naritake, T. Sugibayashi, Y. Nakajima, et al. "A 12ns 8MB DRAM Secondary Cache for a 64b Microprocessor". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 15-17 February 1999. pp. 420–421.

[93] M. Oskin, F.T. Chong, and T. Sherwood. "Active Pages: A Computation Model for Intelligent Memory". In *Proceedings International Symposium on Computer Architecture*, Barcelona, Spain, 27 June – 1 July 1998. pp. 192–303.

[94] M. Oskin, J. Hensley, D. Keen, et al. "Exploiting ILP in Page-Based Intelligent Memory". In *Annual IEEE/ACM International Symposium on Microarchitecture*, Haifa, Israel, 16-18 November 1999. pp. 208–218.

[95] L. Paris, J. Benzreba, P. De Mone, et al. "A 800MB/s 72Mb SLDRAM with Digitally-Calibrated DLL". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 15-17 February 1999. pp. 414–415.

[96] H-C. Park, S-K. Yang, M-C. Jung, et al. “A 833Mb/s 2.5V 4Mb Double Data Rate SRAM”. In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 5-7 February 1998. pp. 356–357.

[97] D.A. Patterson, T. Anderson, N. Cardwell, et al. “A Case for Intelligent RAM: IRAM”. *IEEE Micro*, 17(2):34–44, March/April 1997.

[98] S. Perissakis, Y. Joo, J. Ahn, A. DeHon, and J. Wawrzynek. “Embedded DRAM for a Reconfigurable Array”. In *Symp. VLSI Circuits*, Kyoto, Japan, 17-19 June 1999. pp. 145–148.

[99] J.M. Rabaey. “Reconfigurable Processing: The Solution to Low-Power Programmable DSP”. IEEE International Conference on Acoustics, Speech and Signal Processing, `http://infopad.EECS.berkeley.edu/research/reconfigurable/slides/icassp97.pdf`, 21-24 April 1997.

[100] N.K. Ratha, A.K. Jain, and D.T. Rover. “An FPGA-based Point Pattern Matching Processor with Application to Fingerprint Matching”. In *Proceedings, Converence on Computer Architectures for Machine Perception*, Como, Italy, 18-20 September 1995. pp. 394–401.

[101] R. Razdan and M.D. Smith. “A High-Performance Microarchitecture with Hardware-Programmable Functional Units”. In *Annual IEEE/ACM International Symposium on Microarchitecture*, San Jose, CA, USA, 30 November – 2 December 1994. pp. 172–180.

[102] W.M. Regitz and J. Karp. “A Three-Transistor Cell, 1024-Bit, 500ns MOS RAM”. In *IEEE International Solid-State Circuits Conference*, Philadelphia, PA, USA, 18-20 February 1970. pp. 42–43.

[103] M. Rencher and B.L. Hutchings. “Automated Target Recognition on Splash 2”. In *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, USA, 16-18 April 1997. pp. 192–200.

[104] UC Berkeley BRASS research group. “SCORE project”. `http://brass.cs.berkeley.edu/SCORE`.

[105] UC Berkeley IRAM research group. `http://iram.cs.berkeley.edu/`.

[106] A. Roberts. "Direct Rambus: The New Memory Standard". In *Microprocessor Forum*, 1997.

[107] T. Saeki, Y. Nakaoka, M. Fujita, et al. "A 2.5ns Clock Access 250MHz 256Mb SDRAM with a Synchronous Mirror Delay". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 8-10 February 1996. pp. 374–375.

[108] S. Santhanam. "StrongARM SA110: A 160MHz 32b 0.5W CMOS ARM Processor". In *Hot Chips*, Stanford, CA, USA, 18-20 August 1996. pp. 119–130.

[109] T. Sato, M. Nagamatsu, and H. Tago. "Power and Performance Simulator: ESP and its Application for 100 MIPS/W Class RISC Design". In *IEEE Symposium on Low Power Electronics*, San Diego, CA, USA, 10-12 October 1994. pp. 46–47.

[110] K. Sawada, T. Iizuka, Y. Uchino, and T. Kobayashi. "A 72k CMOS Track-Free Gate Array with Embedded 1 Mbit DRAM". *Electronics and Communications in Japan, Part 2 (electronics)*, 73(5):66–73, May 1990.

[111] S.Cadambi et al. "Managing Pipeline-reconfigurable FPGAs". In *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, USA, 1997.

[112] "International Technology Roadmap for Semiconductors". Technical report, Sematech, 1999. `http://www.itrs.net/1999_SIA_Roadmap/Home.htm`.

[113] T. Shimizu, J. Korematu, M. Satou, et al. "A multimedia 32b RISC microprocessor with 16Mb DRAM". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 8-10 February 1996. pp. 216–217.

[114] S. Shiratake, K. Tsuchida, H. Toda, et al. "A Pseudo Multi-Bank DRAM with Categorized Access Sequence". In *IEEE Symposium on VLSI Circuits*, Kyoto, Japan, 17-19 June 1999. pp. 127–130.

[115] D. Speck. "The Mosaic Fast 512K Scalable CMOS dRAM". In *Advanced Research in VLSI*, 1991, pp. 229–244.

[116] T. Sugibayashi, I. Naritake, S. Utsugi, et al. "A 1Gb DRAM for File Applications". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 15-17 February 1995. pp. 254–255.

[117] T. Sugibayashi, T. Takeshima, I. Naritake, et al. "A 30ns 256Mb DRAM with Multi-Divided Array Structure". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 24-26 February 1993. pp. 50–51.

[118] I.E. Sutherland. "Micropipelines". *Communications of the ACM*, 32(6):720–738, June 1989.

[119] Chameleon Systems. `http://www.chameleonsystems.com`.

[120] Enhanced Memory Systems. "16Mbit ESDRAM Design Guide". `http://www.edram.com/Library/appnotes/an-119.pdf`.

[121] O. Takahashi, S. Dhong, M. Ohkubo, et al. "1 GHz Fully-Pipelined 3.7ns Address Access Time 8k x 1024 Embedded DRAM Macro". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 7-9 February 2000. pp. 396–397.

[122] Y. Takai, M. Fujita, K. Nagata, et al. "A 250Mb/s/pin 1Gb Double Data Rate SDRAM with a Bi-Directional Delay and an Inter-Bank Shared Redundancy Scheme". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 15-17 February 1999. pp. 418–419.

[123] S. Takase and N. Kushiyama. "A 1.6GB/s DRAM with Flexible Mapping Redundancy Technique and Additional Refresh Scheme". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 15-17 February 1999. pp. 410–411.

[124] T. Kimura K. Takeda, Y. Aimoto, et al. "64Mb 6.8ns Random ROW Access DRAM Macro for ASICs". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 15-17 February 1999. pp. 416–417.

[125] S. Tanoi, Y. Tanaka, T. Tanabe, et al. "A 32-Bank 256Mb DRAM with Cache and TAG". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 16-18 February 1994. pp. 144–145.

[126] Toshiba. "ASIC Products". `http://www.toshiba.com/taec/nonflash/components/asic.html`.

[127] Triscend. `http://www.triscend.com`.

[128] TSMC. "Embedded DRAM Brief". `http://www.tsmc.com/e-html/dram-e.htm`.

[129] W. Tsu. "Hierarchical Synchronous Reconfigurable Array". Master's thesis, UC Berkeley, 1999.

[130] W. Tsu, K. Macy, A. Joshi, et al. "HSRA: High-Speed, Hierarchical Synchronous Reconfigurable Array". In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, USA, 21-23 February 1999. pp. 125–134.

[131] A.J. van de Goor. *Testing Semiconductor Memories: Theory and Practice.* Wiley, 1991.

[132] J.E. Vuillemin, P. Bertin, D. Roncin, et al. "Programmable Active Memories: Reconfigurable Systems Come of Age". *IEEE Transactions on VLSI Systems Systems*, 4(1):56–69, March 1996.

[133] T. Wada, S. Ohbayashi, H. Sato, et al. "A 500 MHz Pipelined Burst SRAM with Improved SER Immunity". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 15-17 February 1999. pp. 196–197.

[134] G.K. Wallace. "The JPEG Still Picture Compression Standard". *Communications of the ACM*, 34(4):30–44, April 1991.

[135] J. Walrand and P. Varaiya. *High Performance Communication Networks.* Morgan Kaufmann, 1999.

[136] S.A. White. "Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review". *IEEE ASSP Magazine*, 6(3):4–19, July 1989.

[137] S.J.E. Wilton. *Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memory.* PhD thesis, University of Toronto, 1997.

[138] S.J.E. Wilton, J. Rose, and Z.G. Vranesic. "Memory/Logic Interconnect Flexibility in FPGAs with Large Embedded Memory Arrays". In *IEEE Custom Integrated Circuits Conference*, San Diego, CA, USA, 5-8 May 1996. pp. 144–147.

[139] R.D. Wittig and P. Chow. "OneChip: An FPGA Processor with Reconfigurable Logic". In *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, USA, 17-19 April 1996. pp. 126–135.

[140] J. Woodfill and B. Von Herzen. "Real-Time Stereo Vision on the PARTS Reconfigurable Computer". In *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, USA, 16-18 April 1997. pp. 201–209.

[141] Xilinx. "Virtex Field Programmable Gate Array Databook". `http://www.xilinx.com/partinfo/ds003.pdf`.

[142] Xilinx. "XC4000E and XC4000X Series (EX/XL) Field Programmable Gate Array Databook". `http://www.xilinx.com/partinfo/4000.pdf`.

[143] T. Yabe, S. Miyano, K. Sato, et al. "A Configurable DRAM Macro Design for 2112 Derivative Organizations to be Synthesized Using a Memory Generator". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 5-7 February 1998. pp. 72–73.

[144] T. Yamauchi, L. Hammond, and K. Olukotun. "The Hierarchical Multi-Bank DRAM: A High-Performance Architecture for Memory Integrated with Processors". In *Advanced Research in VLSI*, Ann Arbor, MI, USA, 15-16 September 1997. pp. 303–319.

[145] A. Yamazaki, T. Fujino, K. Inoue, et al. "A 56.8 GB/s 0.18μm Embedded DRAM Macro with Dual Port Sense Amplifier for 3D Graphics Controller". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 7-9 February 2000. pp. 394–395.

[146] H-J. Yoo, K-W. Park, C-H. Chung, et al. "A 150MHz 8-Banks 256M Synchronous DRAM with Wave Pipelining Methods". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, 15-17 February 1995. p. 250.

[147] C. Zhao, U. Bhattacharya, M. Denham, et al. "An 18Mb, 12.3GB/s CMOS Pipeline-Burst Cache SRAM with 1.54GB/s/pin". In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, February 15-17 1999. pp. 200–201.

[148] J.F. Ziegler, M.E. Nelson, J.D. Shell, et al. "Cosmic Ray Soft Error Rates of 16-Mb DRAM Memory Chips". *IEEE Journal of Solid State Circuits*, 33(2):246–252, February 1998.