# Stand-alone Field-Configurable Memory Architectures

Steven J.E. Wilton
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, B.C., Canada V6T 1Z4
(604) 822-2872
stevew@ee.ubc.ca

Jonathan Rose
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada M5S 3G4
(416) 978-6992
jayar@eecg.toronto.edu

Zvonko Vranesic
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada M5S 3G4
(416) 978-5032
zvonko@eecg.toronto.edu

**Abstract**

As the capacities of field-programmable gate arrays (FPGAs) and reconfigurable systems grow, they will be used to implement much larger circuits than ever before. These larger circuits often require significant amounts of memory. Efficient architectural support for memory in FPGAs and reconfigurable systems is, therefore, essential. Since circuits have widely varying memory requirements, a key requirement of the memory architecture is that it be flexible enough to accommodate different memory shapes (widths and depths) as well as allowing different numbers of independently-addressed user memories. In this paper, we describe a parameterized family of such configurable memory architectures that can be used as a stand-alone memory or embedded into an FPGA. This paper also examines the effects of varying various architectural parameters on the device's chip area, access time, and overall flexibility.

**Keywords**

Configurable Memory, FPGAs, Reconfigurable Systems

## I. Introduction

With recent dramatic improvements in integrated circuit technology, more transistors than ever are available to IC designers. The impact of improving process technology is very evident in the evolution of field-programmable gate arrays (FPGAs) and FPGA-based reconfigurable systems (boards and multi-chip modules). In the past, such systems have been primarily used to implement small logic subcircuits, but as these devices grow, FPGAs and reconfigurable systems are being used to implement much larger circuits. An important difference between these larger circuits and the smaller subcircuits that have traditionally been implemented on reconfigurable devices is that the larger circuits often contain significant amounts of memory. This means that efficient architectural support for memory in next-generation FPGAs and FPGA-based systems is critical.

A key requirement of this memory support is the ability to accommodate various numbers, sizes, and shapes of memories. Table I shows several large example circuits. We refer to the memory requirements of a circuit as that circuit's *user memory configuration*. A single memory within a user memory configuration is referred to as a *user memory*. As the table indicates, each application requires a different number of memories, and the memories have different widths and depths.

In this paper, we describe an architecture for field-configurable memory that is flexible enough

TABLE I

Example systems.

| System | User Memory Configuration |
|---|---|
| Viterbi decoder | three 28x16, one 28x3 |
| Graphics Chip | eight 128x22, two 16x27 |
| Neural Networks Chip | 16x80, 16x16 |
| Translation Lookaside Buffer | two 256x69, 16x18 |
| Fast Divider | 2048x56, 4096x12 (ROM) |
| Communications Chip #1 | two 1620x3, two 168x12, two 366x11 |
| Communications Chip #2 | six 88x8, one 64x24 |
| Communications Chip #3 | one 192x12 |

to implement the storage requirements of a wide variety of such circuits. Such an architecture can be used in two ways. First, it can be used as a stand-alone device, in which the memory pins are connected directly to I/O pads. Such a chip would be valuable in reconfigurable systems consisting of FPGAs, memory devices, and interconnect [1], [2], [3], [4], [5]. These systems have been used to achieve significant speedups in many compute-intensive applications (compared to software implementations), as well as being used as an efficient medium for logic emulation. In many of these systems, user memories are packed into standard off-the-shelf memory chips. These user memories typically have a different size or aspect ratio than the physical memory devices. When this happens, either memory is wasted, or the user memories must be time-multiplexed onto the memory chips [6]. By providing memory resources that can better adapt to the requirements of circuits, more efficient circuit implementations are possible.

There are similarities between our architecture and a general-purpose CPU memory system that can provide byte, word, or long-word data to a CPU depending on the type of memory access being performed. The CPU sees a single bank of memory that is either 8 bits, 16 bits, or some other width. There are two significant differences between this and a memory required in a reconfigurable system. First, as evidenced in Table I, the memories in application circuits tend to require many widths besides the traditional powers-of-two. The architecture presented in this paper can match the memory widths of user memories much better than can a CPU memory system. Second, user applications often require separate banks of memory that must be accessed simultaneously. If a single large memory is used, only one access can be satisfied at a time, often leading to significant performance degradation. In our architecture, many user memories can be implemented, and these user memories can be accessed simultaneously.

The architecture presented in this paper can also be embedded into an FPGA to provide on-chip configurable memory. On-chip memory has a number of advantages over off-chip memory: it reduces the system cost by decreasing the number of chips required to fully implement a system, it often allows for faster clock rates since external pins (and board-level traces) need not be driven with each memory access, and it frees I/O pins that would otherwise be devoted to address and data connections. FPGAs with significant amounts of on-chip memory are available from several vendors. The Altera FLEX 10K architecture contains between three and twelve *embedded array blocks*, each of which contains a 2Kbit array [7]. Other FPGAs with on-chip memory include the Actel SPGA, the Actel 3200DX, the Xilinx 4000 family of devices, and the Lucent Technologies ORCA FPGAs [8], [9], [10], [11]. There has been little published work that compares and evaluates configurable memory architectures. The extension of the architecture described in this paper to on-chip applications is discussed in [12], [13].

In [14], a specific architecture for a field-configurable memory was proposed and its implementation was described. In this paper, we identify the architectural parameters of that device, and explore their effects on the chip area, access time, and the ability of the device to implement different user memories. This paper significantly extends the analysis given in [15].

## II. Configurable Memory Architecture Family

The configurable memory of [14] consists of four arrays, each with a configurable aspect ratio, that can be programmably connected together. Fig. 1 illustrates a generalization of that architecture consisting of $B$ bits divided among $N$ identical memory arrays, each with a single data port. The ability to implement many different numbers and shapes of user memories is achieved in two ways: by allowing the user to configure the effective data width of each array (trading width for depth), and by allowing the user to combine arrays to implement larger memories.

The ability to configure the effective data width of each array is provided by the L1 data mapping blocks. Each array has an associated L1 data mapping block, and each can be configured independently. Section II-A describes this mapping block in more detail.

The ability to combine arrays to implement larger memories is provided by the L2 data and address mapping blocks. These blocks programmably connect arrays to each other and to the external bidirectional data and unidirectional address buses. These buses are of a fixed width, and are connected directly to I/O pads. The user can access memory only through these buses. We assume that a single bus can be used for only one user memory at a time, meaning that if a memory does not use the entire bus, the rest is wasted. This assumption simplifies the design of the L2 data mapping and address mapping blocks; the structure of these blocks is the focus of Section II-B.

The parameters used to characterize each member of this architectural family are summarized in Table II. For an architecture with a given number of bits, $B$, there are many choices for the number of arrays ($N$), the number of external data and address buses ($M$ and $Q$), and the nominal and set of allowable effective data widths ($W_{\mathrm{nom}}$ and $W_{\mathrm{eff}}$). Each of these parameters affects the speed and area of the device, as well as its ability to implement different user memory configurations. As an example, an architecture with more, smaller arrays (higher $N$), will suffer in terms of area and speed due to the array and mapping block overhead, when compared to an architecture with fewer, larger arrays (lower $N$). The former architecture, however, will be more flexible, since the number of ways in which the arrays can be combined to implement user memories is greater. Similar tradeoffs exist for the other parameters. All of these tradeoffs must be considered when designing a configurable memory. In Section IV we vary these parameters and measure the effects on area, speed, and flexibility of the device. In the remainder of this section, we describe the details of the general architecture of Fig. 1.

Note that we assume the data lines are bidirectional. Therefore, the architecture can be used to implement either RAM's or ROM's. If the device is used to implement a ROM, the data pins are only used as inputs.

### *A. L1 Data Mapping Block*

Each of the $N$ arrays has a nominal width of $W_{\mathrm{nom}}$ and depth of $B/(NW_{\mathrm{nom}})$. As indicated above, the user can configure the effective data width of each array independently using an L1 data mapping block (there is one L1 data mapping block per array). The mapping block consists of a grid of programmable switches, as shown in Fig. 2(a). Each switch, indicated by a circle in the diagram, programmably connects a vertical track (data line from the memory array) to a horizontal track (data line from the L2 data mapping block).

Note that not every intersection in Fig. 2 contains a switch. The switch pattern determines

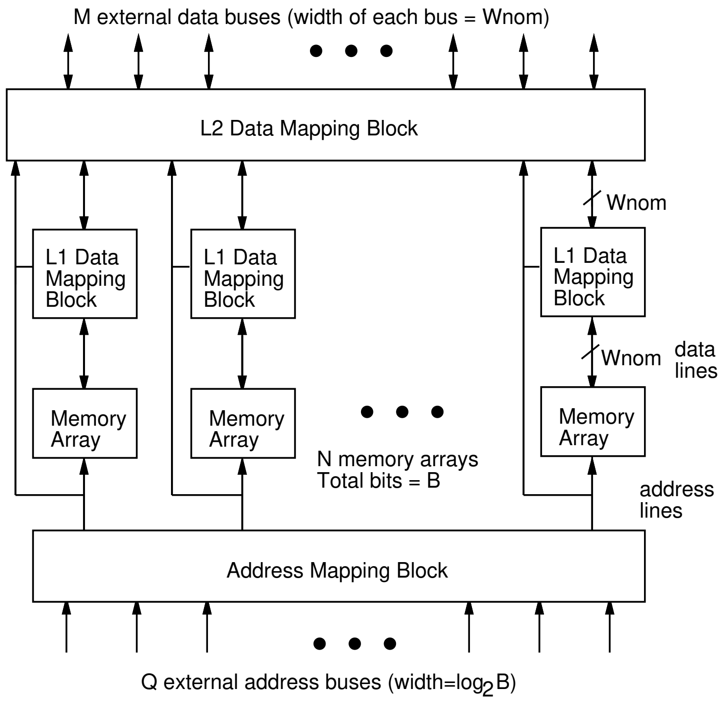



Fig. 1. General architecture for a stand-alone configurable memory.

the capability of the L1 data mapping block. In the example, the switch pattern is sufficient to allow the user to configure the array to be one of $\frac{B}{N}$x1, $\frac{B}{2N}$x2, $\frac{B}{4N}$x4, or $\frac{B}{8N}$x8. Fig. 2(b) shows two sets of switches, $A$ and $B$, that are used to implement the $\frac{B}{4N}$x4 configuration. One of the memory address bits is used to determine which set of switches, $A$ or $B$, is turned on. Each set of switches connects a different portion of the memory array to the bottom four data lines.

The mapping block in this example is capable of implementing all power-of-two widths between 1 and $W_{\mathrm{nom}}$. In general, the L1 data mapping block capability is quantified by the parameter $W_{\mathrm{eff}}$; this parameter indicates the set of allowable data widths for each array. The relationship between $W_{\mathrm{eff}}$ and the switch pattern is as follows. The set of horizontal tracks to which each vertical track $i$ can be connected is:

$$T(i) = \{i \bmod j \;:\; j \in W_{\mathrm{eff}}\} \qquad (1)$$

Each of the possible connections represents one programmable switch.

By removing every second switch along the bottom row of the block in Fig. 2(a), a faster and smaller mapping block could be obtained. The resulting mapping block would only be able to provide an effective data width of 2, 4, or 8 ($W_{\mathrm{eff}} = \{2, 4, 8\}$), however, meaning that the resulting architecture would be less flexible. The impact of removing L1 data mapping block

TABLE II

Architectural parameters.

| Parameter | Meaning |
|---|---|
| $B$ | Total bits |
| $N$ | Number of arrays |
| $M$ | Number of external data buses |
| $Q$ | Number of external address buses |
| $W_{\text{nom}}$ | Nominal data width of each array |
| $W_{\text{eff}}$ | Set of allowable effective data widths of each array |

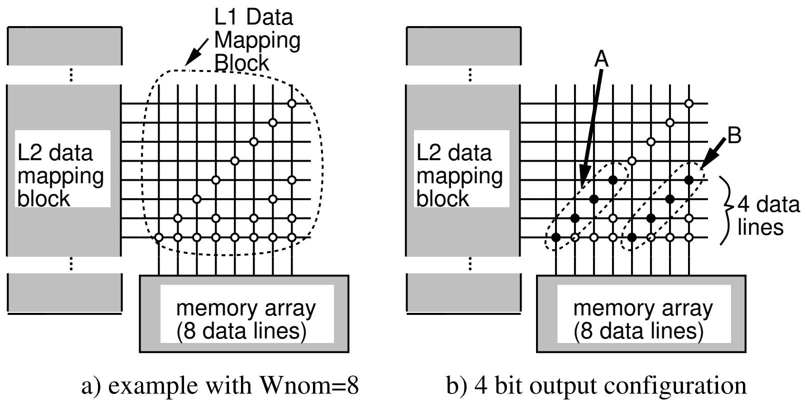
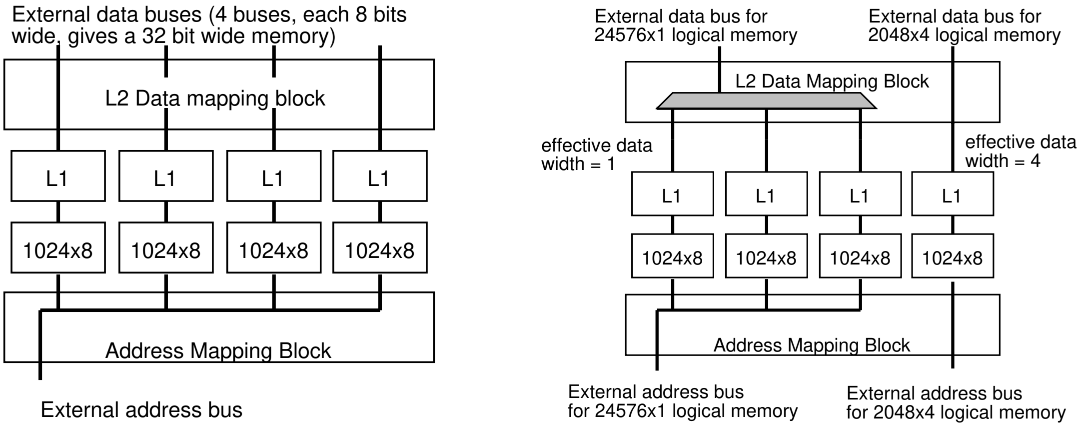



Fig. 2. L1 data mapping block.

switches on area, speed, and flexibility will be examined in Section IV-B.

Note that we have only considered widths that are a power-of-two. It is very difficult to create an L1 mapping block in which the number of data lines incident to each array is not a multiple of all the allowable effective widths. In general, however, this is not necessary. As will be explained below, arrays can often be combined to implement larger memories; when arrays are combined, odd data widths can be created. For example, a user memory that is three bits wide can be created by combining three arrays, each configured as "x1". Alternatively, a three bit wide memory can be implemented by using a single array in the "x4" mode; in this case, one bit per word is wasted.




a) 1024x32 b) 24Kx1 and 2048x4

Fig. 3. Two example mappings.

### *B. L2 Data Mapping Block and Address Mapping Block*

Memory flexibility is also obtained by allowing the user to combine arrays to implement larger memories. Fig. 3(a) shows how four 1024x8 arrays can be combined to implement a 1024x32 user memory. In this case, a single external address bus is connected to each array, while the data bus from each array is connected to separate external data buses (giving a 32-bit data width). Each L1 data mapping block connects 8 array data lines directly to the 8 L1 outputs.

Fig. 3(b) shows how this architecture can be used to implement a configuration containing two user memories: one 24Kx1 and one 2048x4. The three arrays implementing the 24Kx1 memory are each configured as 8192x1 using the L1 data mapping block, and each data line is multiplexed to a single external data line using pass transistors. Two address bits control the pass transistors; the value of these address bits determine which array drives (or is driven by) the external data line. The 2048x4 memory can be implemented using the remaining array, with the L1 block configured in the "by 4" mode.

The extent to which arrays can be combined depends on the flexibility of the L2 data and address mapping blocks. The address mapping block must be flexible enough that each array that is used in a single user memory can be connected to the same external address bus. The L2 data mapping block must be flexible enough to combine arrays in two ways:

1. Arrays can be combined "horizontally" to implement wider user memories. In this case, each array must be connected to its own external data bus. An example of this is shown in Fig. 3(a).

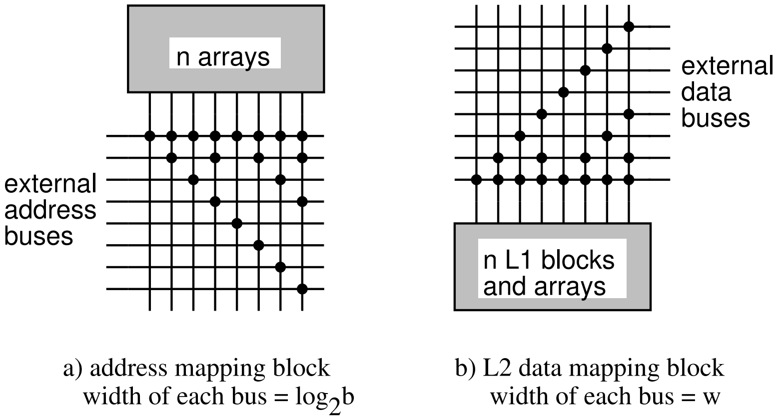



Fig. 4. Level 2 data and address mapping block topology ($N = M = Q = 8$).

2. Arrays can be combined "vertically" to implement a deeper user memory. In this case, the data buses of all arrays that implement this user memory must be multiplexed onto a *single* external data bus. An example of this is shown in Fig. 3(b).

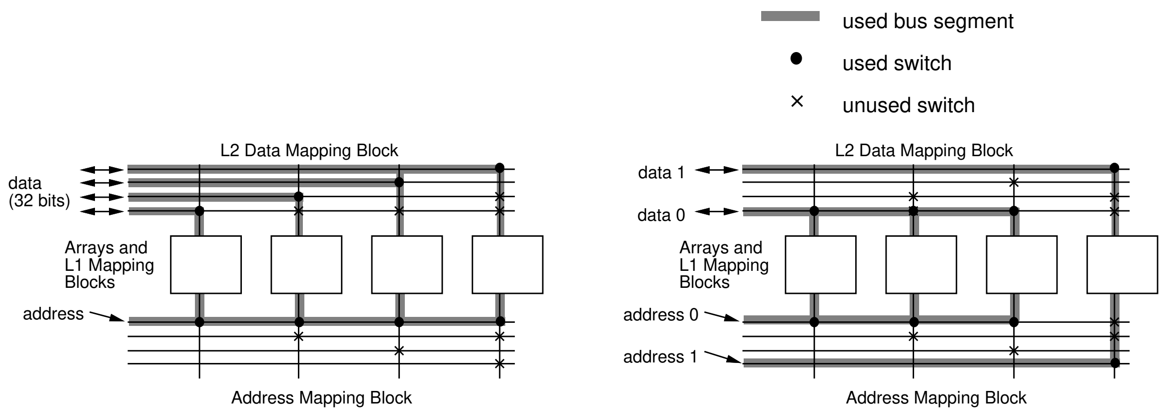
Of course, some user memories can be implemented by combining arrays both horizontally and vertically.

The topology of the switches in the L2 data mapping block and the address mapping block determines the flexibility of these blocks. If both of these mapping blocks are fully populated, meaning any external bus (both address and data) can be connected to any array, a very flexible architecture would result. Unfortunately, the switches take chip area and add capacitance to the routing lines, resulting in a larger and slower device than is necessary. The switch topologies shown in Fig. 4 provide a compromise between speed and flexibility. In this figure, each dot represents a set of switches controlled by a single programming bit, one switch for each bit in the bus ($W_{\mathrm{nom}}$ in the L2 data mapping block and $\lceil log_2 B \rceil$ in the address mapping block). This pattern is similar to that of the L1 data mapping block described by Equation 1. In the L2 data mapping block, the set of buses to which array $i$ can be connected ($0 \leq i < N$) is:

$$B(i) = \{ i \bmod 2^j \ : \ 0 \leq j \leq log_2 M \} \tag{2}$$

The pattern in the address mapping block is the same with $M$ replaced by $Q$. Although in the examples of Fig. 4, $M = Q = N$, the same formula applies for non-square mapping blocks. Note that this equation assumes that $M$ and $Q$ are powers of two, which is the case in all architectures considered in this paper. In Sections IV-D and V, this mapping block topology will be compared to other possible patterns.

Fig. 5 shows how a mapping block defined by Equation 2 with $N = M = Q = 4$ can be used




a) 1024x32 b) 24Kx1 and 2048x4

Fig. 5. Implementation of two examples in Fig. 3.

to perform the connections required in the examples of Fig. 3.

It is important to note that all the bits of a bus are routed as a unit through the L2 data and address mapping blocks. Thus, a single programming bit can be used to control each set of $W_{\mathrm{nom}}$ switches in the L2 data mapping block and $\lceil log_2 B \rceil$ switches in the address mapping block. This results in a significant savings in area (compared to a device in which each switch has its own programming bit), but reduces the flexibility of the architecture. In the example of Fig. 5(b), only one bit of the bottom data bus is used; the others are wasted and can not be used for other user memories.

In addition to address and data lines, write enable signals are required for each array. The write enable lines can be switched in the L2 mapping block just as the data lines are. In order to correctly update arrays for effective widths less than the nominal width, we assume that the arrays are such that each column in the array can be selectively enabled. The address bits used to control the L1 mapping block can be used to select which array column(s) are updated [14].

### *C. Wide Mapping Blocks*

For architectures with $N > 8$, the horizontal buses become heavily loaded and will begin to dominate the memory access time. To reduce this effect, the two-stage structure shown in Fig. 6 is assumed for wide mapping blocks. This diagram shows the address mapping block; the L2 data mapping block is similar, with each inverter replaced by the bidirectional driver in Fig. 7. The optimal number of sub-buses that each bus should be broken into depends on the width of the original mapping block.

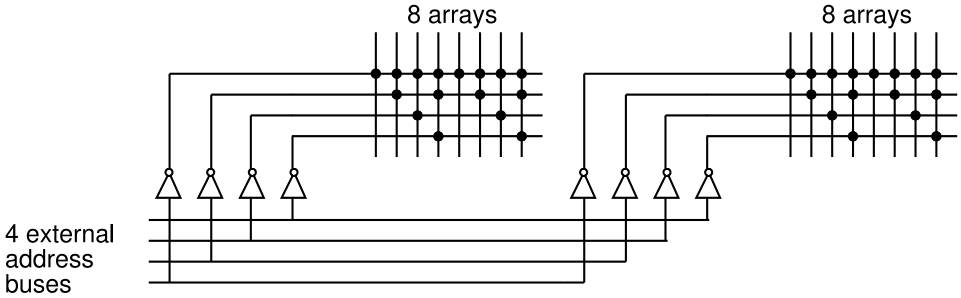



Fig. 6. Two-stage address mapping block.

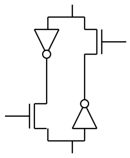


Fig. 7. Bi-directional switches in two-stage L2 data-mapping block.

## III. Experimental Methodology

The configurable memory architecture described in the last section forms the framework for a very flexible memory that can be used in a wide variety of applications. In order to create a good configurable memory, however, it is crucial to understand how each of the architectural parameters in Table II affects the chip area, memory access time, and overall device flexibility. In the remainder of this paper, we explore these tradeoffs by varying the array size, data bus width, and the L1 and L2 mapping block structures.

Fig. 8 illustrates the experimental methodology employed. For each architecture under study, we attempt to "implement" many user memory configurations using custom CAD algorithms. Each of these implementation attempts either succeeds or fails; we count the number of failures and use the count as a flexibility metric. The architecture with the fewest failures is deemed the most flexible. In addition to the flexibility results, we also obtain and compare area and access time estimates for each architecture using detailed analytical models.

The source of the benchmark memory configurations, the CAD algorithms, and the analytical models are discussed below; more details can be found in [12].

### *A. Source of Benchmark User Memory Configurations*

In order to obtain meaningful results, it is important that we use enough benchmark circuits to thoroughly exercise each architecture. In FPGA architectural studies, it is common to use 5

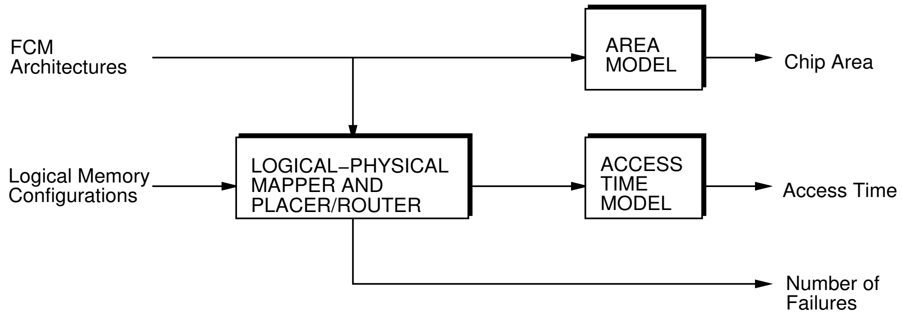



Fig. 8. Methodology for stand-alone memory experiments.

to 20 "real" circuits as benchmarks [16], [17], [18], [19], [20]. This works well for these studies; since each of these circuits contains hundreds (or thousands) of logic blocks, we can be confident that the architecture under study is thoroughly exercised using only a few circuits. Most memory circuits, however, only contain a few user memories. Thus, to thoroughly exercise a configurable memory architecture, we need hundreds (or thousands) of benchmark circuits to achieve the same level of confidence.

Unfortunately, we were unable to gather such a large number of complete circuits. Instead, we generated them stochastically. We have developed a circuit generator that stochastically creates realistic benchmark circuits; we extracted the memory configurations from the circuits, and used them as benchmark configurations in our experiments. Although stochastic circuit generators have been used and described elsewhere [21], [22], ours is the first that generates circuits containing both logic and memory.

It is crucial that the generated memory configurations are realistic. We ensure this by basing the generator on the results of a detailed circuit analysis. The next subsection briefly outlines the analysis, while the following subsection describes how we use the analysis results to ensure our stochastically generated memory configurations are realistic. A full explanation of both the analysis and generation is given in [12].

### A.1 Structural Analysis of Circuits with Memory

This analysis is based on 171 circuits containing a total of 268 user memories. Data regarding these circuits was obtained from several sources: recent conference proceedings, recent journal articles, local designers at the University of Toronto, a major telecommunications company, and a customer study conducted by Altera [23]. All circuits contributed equally to the gathered data.

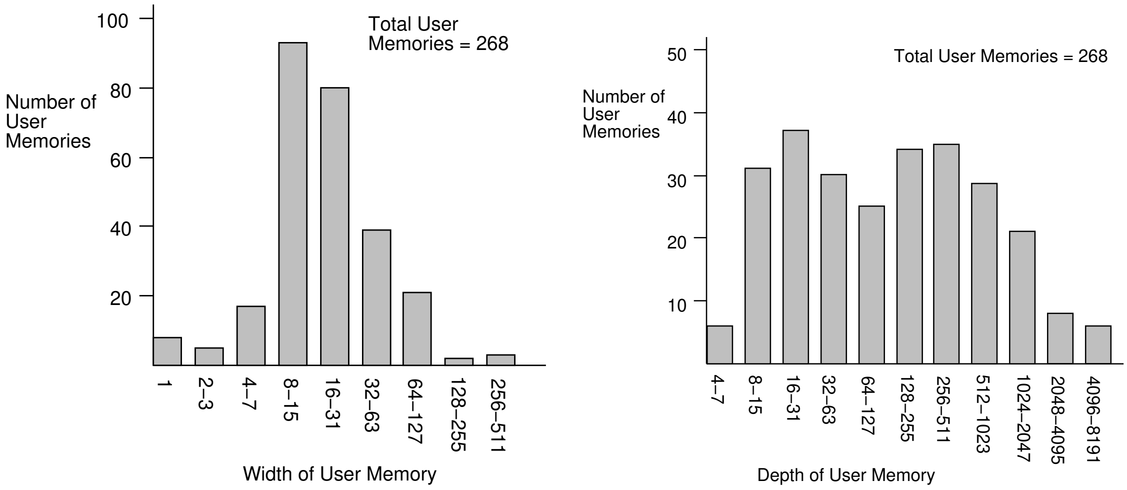



Fig. 9. Distributions of user memory widths and depths.

Although we were unable to obtain netlists for the circuits, we could gather several key memory parameters.

As an example of the data gathered during the analysis, Fig. 9 shows the width and depth profiles of the user memories in our sample circuits. As the graphs show, the proportion of memories with depths in each power-of-two-interval between 8 and 2048 is roughly constant, while the memory width distribution peaks at about 8, and falls off quickly below 8 and above 16. Another interesting measurement not reflected on the graph is that in 69% of the widths and 74% of depths are a power of two.

In the same way, results were obtained regarding the number of memories in each circuit, and correlations between memory parameters among the memories of each circuit. We also gathered information about how the memories were connected to logic, but this information was not required for the study presented in this paper.

One item we were not able to consider was the temporal behavior of the memory access patterns. In many circuits, not every memory must be accessed every clock cycle. In such circuits, efficient implementations are possible by time-multiplexing the pins and/or arrays onto the physical resources. Unfortunately, we did not have this temporal information for any of our circuits. The extension of the results along this dimension is an area of future work.

### A.2 Circuit Generation

In order to ensure that the circuits from the circuit generator are realistic, we closely based the generation on the results of the circuit analysis. The distributions in Fig. 9, along distributions regarding the number of memories in circuits and correlations between memory parameters within circuits, were used as probability distributions during the generation. This means that, for example, most circuits contain memories of widths between 8 and 16, but there is a small probability of generating a circuit that uses wider or narrower memories. The algorithm for choosing the memory parameters is described in detail in [12].

### *B. CAD Algorithms*

Implementing a user memory configuration on a physical configurable memory architecture requires the following steps:

*logical-to-physical mapping:* mapping the user memory configuration to a "netlist" of arrays, and determining the aspect ratio of each array. A valid netlist contains at most $N$ arrays and requires at most $M$ data buses and $Q$ address buses. If such a netlist can not be found, the user memory configuration can not be implemented using the given architecture.

*placement and routing:* assigning physical arrays to implement each array in the netlist, and connecting the physical arrays to the external buses using the mapping blocks.

Heuristic algorithms that provide near-optimal solutions to both of the above problems were developed and are described below. For more details, see [12].

#### B.1 Logical-to-Physical Mapping

The goal of the logical-to-physical mapping algorithm is to map the user memory configuration to a "netlist" of arrays, and to determine the aspect ratio of each array. In our heuristic algorithm, we only search for solutions in which all arrays making up a single user memory use the same aspect ratio. For each user memory, we consider all possible aspect ratios and calculate the number of arrays, data buses, and address buses required. These leads to up to $|w_{\mathrm{eff}}|$ possible implementations of each user memory. Then, by considering all user memories in the configuration simultaneously, we find an implementation of the entire configuration that requires a total of $N$ or fewer arrays, $M$ or fewer data buses, and $Q$ or fewer address buses. In the worst case, this would require checking $|w_{\mathrm{eff}}|^N$ combinations, but in practice, most potential implementations for each user memory can be eliminated without considering the other user memories.

Occasionally, more than one valid solution is found. Without information regarding the location of the switches in the L2 data mapping block and the address mapping block, it is impossible to ascertain which of the valid solutions (if any) will result in a successful placement/routing. To avoid making an arbitrary decision at this point, *all valid solutions* are passed to the placer/router. The placer/router is then free to use which ever logical-to-physical mapping results in a successful place and route. This is in contrast to most technology mapping algorithms for logic circuits, in which only one valid mapping is passed to the placer/router.

### B.2 Placement and Routing

The purpose of the place and route algorithm is to assign physical arrays to implement each array in the netlist, and to connect the physical arrays to the external buses using the mapping blocks. Although any solution to the logical-to-physical mapping algorithm is guaranteed to use $N$ or fewer arrays, $M$ or fewer data buses, and $Q$ or fewer address buses, there is no guarantee that the solution(s) found from the previous stage will be routable because of the limited mapping block flexibility. The limited mapping block flexibility also, in many cases, dictates the placement of the memory arrays and the assignment of the external buses. Thus, it is essential to perform the placement and routing tasks simultaneously.

Consider an exhaustive algorithm which considers all combinations of array placement and external bus assignment. For each combination, the algorithm would determine whether the mapping blocks are capable of making the required connections. Such an algorithm is infeasible for large architectures; the worst-case complexity is $O(N! * M! * Q!)$.

Instead, we have created a heuristic algorithm, which processes each user memory sequentially, beginning with the one that requires the fewest number of arrays. This arrays implementing this memory are placed in such a way that they can be connected to the least flexible available address and data buses. This is repeated for each user memory, until either all memories have been placed (success) or until an array can not be connected to the required number of external buses (failure). In [12], the performance of this heuristic is compared to that of the exhaustive algorithm; for only 0.35% of the test cases did the exhaustive algorithm find a solution that the heuristic algorithm missed.

## *C. Area and Access Time Models*

To perform meaningful area and access time comparisons, we used detailed analytical models. The access time model, which was modified from a detailed cache access time model [24], is used to estimate the read access time of the memory. The read access time includes the delay of the address mapping block, the delay of the memory array, and the delay of the data mapping block.

Note that, in our architecture, some of the address lines are decoded and used to drive switches in the data block. We assume that the delay of these decoders is less than the delay of the actual memory array itself. Thus, by the time the data has been read from the memory, and is available at the sense amps, the switches in the data mapping block have settled.

The array delay model contains terms for the delays due to the decoder, word lines, bit lines, column multiplexors, and sense amplifiers. The networks in the mapping blocks were modeled using RC-based techniques from [25]. The number of switches encountered in the mapping blocks is different for each external bus; this potentially causes skew between the paths into and out of memory. For each memory implementation, we find the delay of the longest path through the mapping blocks, and use them when calculating the memory access time.

In our model, we assume a logic 0.8um CMOS process. Clearly, the delays would be different for different processes; in more advanced processes, a higher proportion of the overall delay is likely to be attributed to the mapping blocks (because of the long wires in these blocks). We have also assumed specific driver sizes in the mapping blocks; the driver sizes were chosen to give reasonable delays across the range of mapping block sizes. If these driver sizes were to change, however, the delay of the mapping blocks would change as well. The effects of varying these technology parameters is an area of future work.

The area model was based on measurements obtained from a configurable memory implementation [14]. To obtain a degree of process independence, area measurements are given in *memory bit equivalents* or *mbe's*; one mbe is equal to the size of one memory cell in an SRAM array (1 mbe = 0.6 rbe in [26] $\approx 250um^2$ in a 0.8um CMOS process).

Both the area and access time models are described in more detail in [12].

## IV. Experimental Results

In order to create a flexible but efficient configurable memory architecture, it is vital to understand how the architectural parameters from Table II affect the flexibility, access time, and chip area of the resulting device. This section presents experimental results that give insight to the effects of these parameters.

In our experiments, we concentrate on two architectures; the nominal parameter values for these architectures are shown in Table III. The nominal switch patterns for the L2 data and address mapping blocks are given by Equation 2. In the following subsections, we vary the number of arrays ($N$), the capability of the L1 mapping block ($W_{\text{eff}}$), the number of data buses

TABLE III

Nominal Parameter Values for Experimental Architectures.

| Parameter | Value |
|---|---|
| $B$ | 8Kbits |
| $N$ | 8 |
| $M$ | 4 |
| $Q$ | 4 |
| $W_{\text{eff}}$ | {1,2,4,8} |

a) Architecture 1

| Parameter | Value |
|---|---|
| $B$ | 64Kbits |
| $N$ | 16 |
| $M$ | 8 |
| $Q$ | 8 |
| $W_{\text{eff}}$ | {1,2,4,8,16} |

b) Architecture 2

($M$), and the L2 mapping block switch patterns, and examine the effects on the device area, speed, and flexibility.

*A. Number of Basic Arrays*

One of the key architectural parameters is the number of arrays, $N$. This subsection examines the impact of changing $N$ while keeping the total number of bits constant.

The motivation for increasing $N$ is flexibility. Since each array can be connected to at most one address bus at a time, it can only be used to implement one user memory. If a user memory does not use the entire array, the remaining bits are wasted. This is especially a problem when implementing configurations with many small user memories. As an example, an architecture with four 1-Kbit arrays can implement at most four user memories, no matter how small they are. An architecture with eight 512-bit arrays, however, can implement configurations with up to eight user memories if there are sufficient data buses.

We explored the flexibility as a function of $N$ for the two architectures. As shown in Table III, the first contains 8 Kbits of memory and four address and data buses, while the second contains 64 Kbits of memory and eight address and data buses. In each case, $N$ was varied from its minimum value (it does not make sense to have $N < Q$) to 64. We stochastically generated 100,000 user memory configurations, and used the CAD tools to attempt to map each memory configuration to each architecture. Fig. 10 shows plots of the proportion of the configurations that could *not* be implemented using each architecture as a function of $N$. In each graph, the configurations that could not be mapped are broken into three categories:

1. The lower dashed line indicates the mapping attempts that failed because the logical-to-

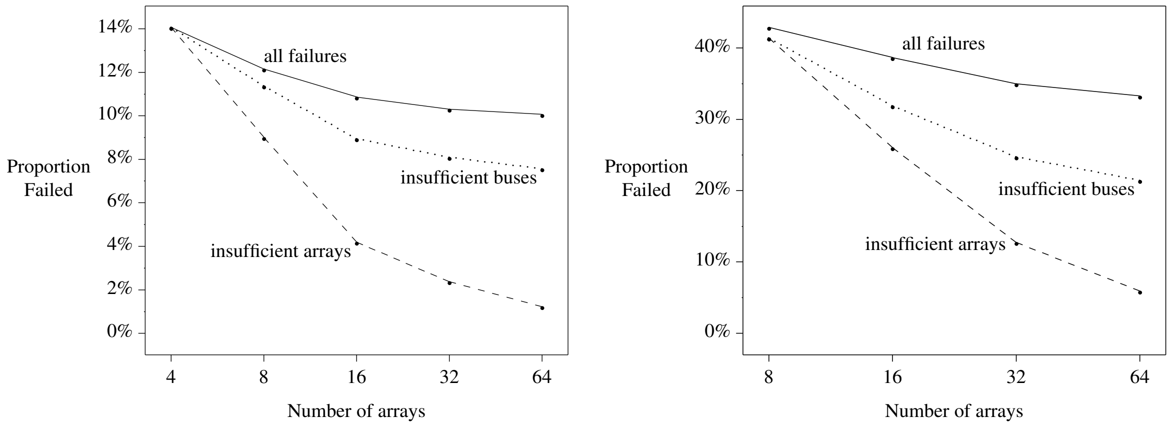
physical mapper could not find a mapping using $N$ or fewer arrays.

2. The distance between the dashed and dotted lines indicates the configurations for which the logical-to-physical mapper could not find a mapping requiring $M$ or fewer external data buses.
3. The distance between the dotted line and the solid line indicates the configurations that failed in the placer/router stage; that is, the switches in the L2 mapping block and address mapping blocks were not sufficient to perform the required mapping connections.

Configurations that fail for more than one of the above reasons are classified into the first appropriate failure category.

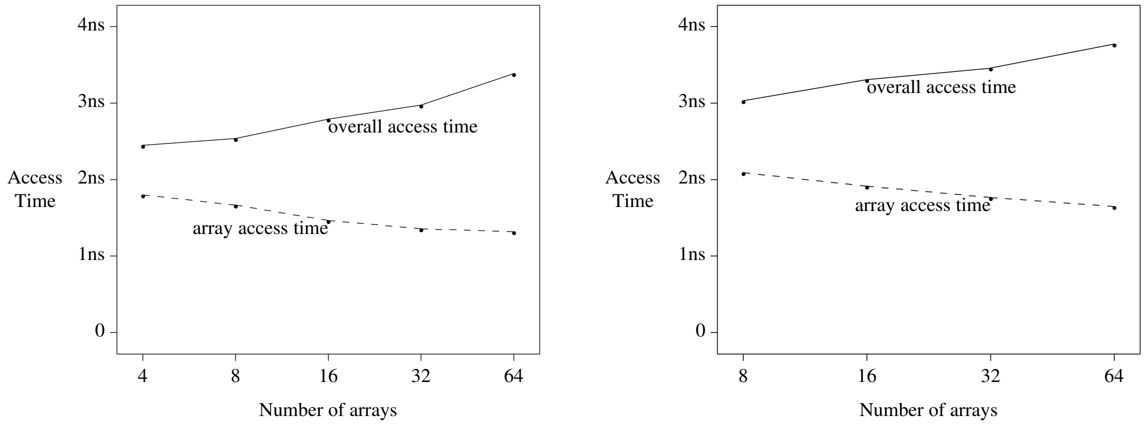
As the two graphs show, for low values of $N$, the predominant cause of failures is that the logical-to-physical mapper can not find a mapping requiring $N$ or fewer arrays. As $N$ increases, the blame shifts to an insufficient number of external data buses or an insufficient number of switches in the mapping blocks. The total number of failures, however, drops as $N$ increases. For the first architecture, the number of failures is reduced by 28% over the range of the graph, while for the second architecture, the number of failures is reduced by 22%.

Fig. 11 shows how the memory access time is affected by $N$ for the same two architectures. The access time is broken into two components: the array access time and the delay due to the mapping blocks. For each value of $N$ we tested both one and two level mapping blocks; for the two level blocks we varied the number of sub-buses in the lower level. The mapping block that resulted in the minimum access time was chosen.



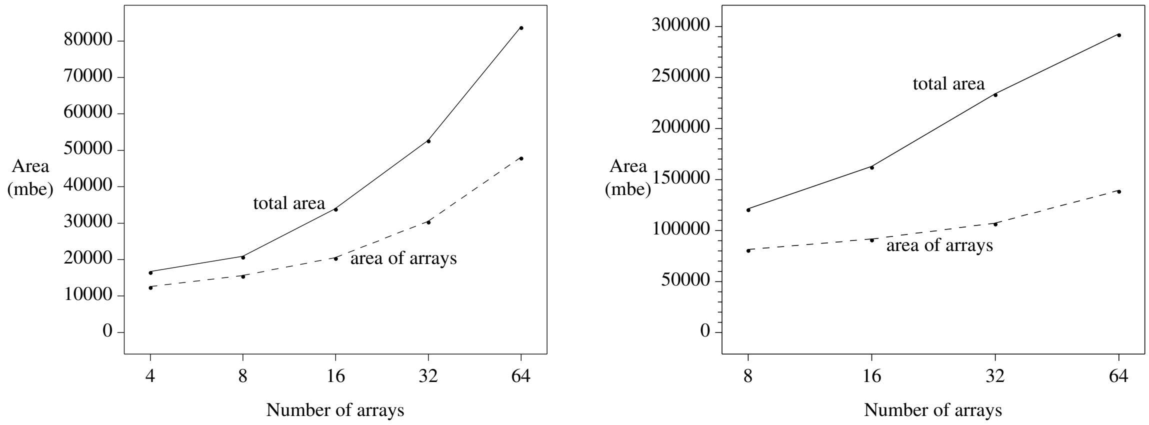

a) $B = 8K, M = Q = 4, W_{\text{eff}} = \{1, 2, 4, 8\}$ b) $B = 64K, M = Q = 8, W_{\text{eff}} = \{1, 2, 4, 8, 16\}$

Fig. 10. Number of failures as a function of number of arrays ($N$).




a) $B = 8K, M = Q = 4, W_{\text{eff}} = \{1, 2, 4, 8\}$ b) $B = 64K, M = Q = 8, W_{\text{eff}} = \{1, 2, 4, 8, 16\}$

Fig. 11. Access time as a function of number of blocks ($N$).

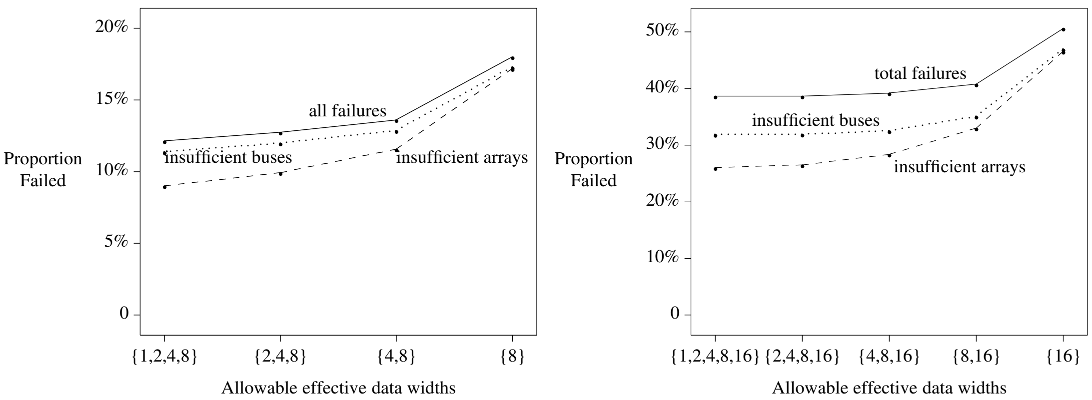



a) $B = 8K, M = Q = 4, W_{\text{eff}} = \{1, 2, 4, 8\}$ b) $B = 64K, M = Q = 8, W_{\text{eff}} = \{1, 2, 4, 8, 16\}$

Fig. 12. Chip area as a function of number of arrays ($N$).

As $N$ increases, each array gets smaller; these smaller arrays have lower access times as shown by the dotted lines in Fig. 11. This is overshadowed, however, by an increase in the delay due to the mapping blocks; as $N$ increases, these mapping blocks get larger, causing the overall access time to increase. For the smaller architecture, the access time increases by about 38% over this range, while for the larger architecture, it increases by about 24%.

Finally, Fig. 12 shows the chip area required by each architecture as a function of $N$, again broken into two components: the area of the arrays (along with their support circuitry) and the area due to the mapping blocks. As $N$ increases, the arrays get smaller, meaning the overhead due to the array support circuitry increases. The mapping blocks also get larger as $N$ increases.




a) $B = 8K, N = 8, M = Q = 4$ b) $B = 64K, N = 16, M = Q = 8$

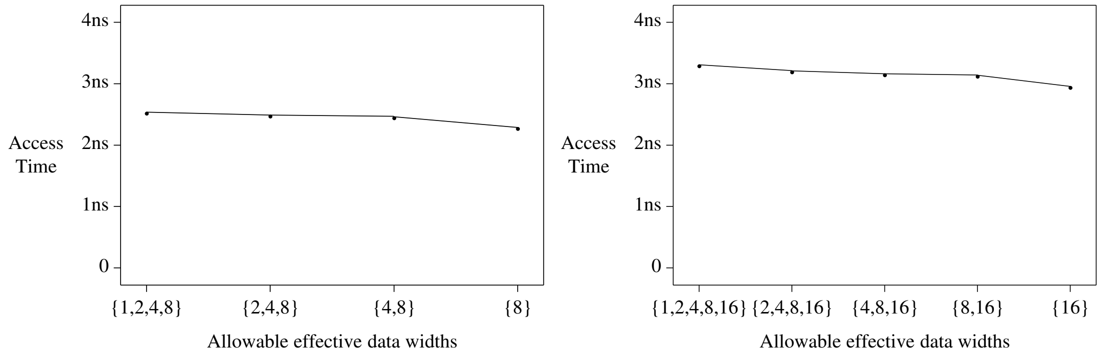
Fig. 13. Number of failures as a function of L1 mapping block capability.

Combining the access time, area, and flexibility results, a good choice for the smaller architecture is approximately $N = 8$ or $N = 16$, while for the second architecture, $N$ should be slightly larger. Intuitively, the optimum choice for $N$ depends heavily on the number of address buses, since the number of address buses dictates the maximum number of user memories that can be implemented on the architecture, which in turn, influences the number of arrays required. Thus, we would expect that if the number of address buses is increased beyond 8, the optimum number of arrays would increase as well.

### *B. L1 Mapping Block Capability*

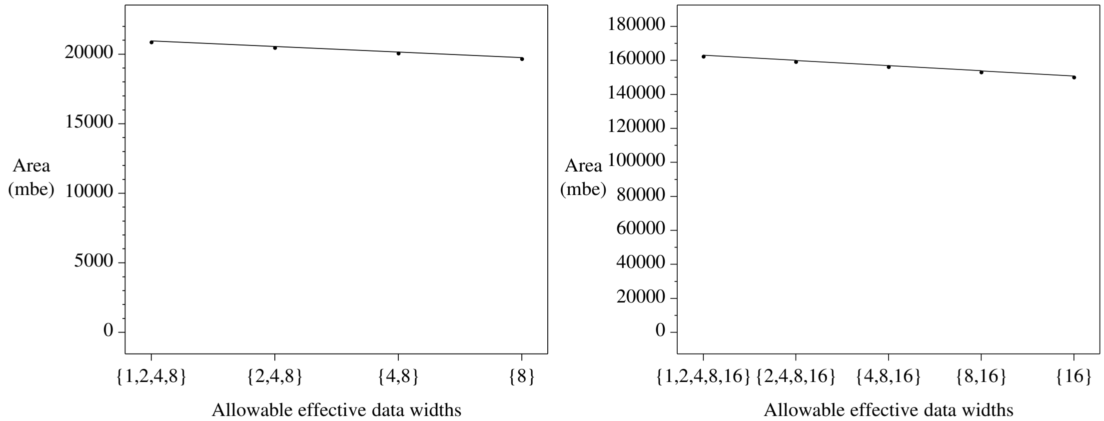
The effective data width of each array can be set by configuring the L1 data mapping blocks. In the previous set of results, it was assumed that the set of effective output widths, $W_{\text{eff}}$, consisted of all powers-of-two between 1 and the nominal array width $W_{\text{nom}}$. Section II-A discussed how a faster, but less flexible architecture could be obtained by removing some of the capability of the L1 data mapping block. In this section, we investigate the effects of changing the minimum effective data width (smallest value of $W_{\text{eff}}$).

Intuitively, the higher the minimum data width, the less flexible the L1 mapping block, and hence, the less flexible the architecture. As shown in Fig. 13, this is indeed the case; decreasing the L1 block flexibility causes the logical-to-physical mapper to fail more often. The difference is more noticeable in the smaller architecture. Recall that we are only including configurations that use between 75% and 100% of the available bits; configurations aimed at the larger architecture are more likely to have wide data widths, meaning an L1 block that can be configured in the "by




a) $B = 8K, N = 8, M = Q = 4$ b) $B = 64K, N = 16, M = Q = 8$

Fig. 14. Access time as a function of L1 mapping block capability.




a) $B = 8K, N = 8, M = Q = 4$ b) $B = 64K, N = 16, M = Q = 8$

Fig. 15. Chip area as a function of L1 mapping block capability.

1" or "by 2" configuration is less important.

Figs. 14 and 15 show that the speed and area benefit of the simpler L1 mapping blocks is small. For both architectures, the access time of a memory employing the least flexible L1 mapping block is only 10% less than that for a memory employing the most flexible mapping block. The area requirements of a memory with the least flexible mapping block are only 6% and 7% less than if the most flexible block was used for the two architectures considered. Thus, we conclude that, especially for small architectures, a flexible L1 mapping block is most appropriate.

### *C. Data Bus Granularity*

In Section IV-A, the number of failures due to insufficient arrays was reduced by increasing the array granularity (creating more, but smaller, arrays). Unfortunately, most of the gain was lost because the architecture did not contain enough external data buses. In this section, we

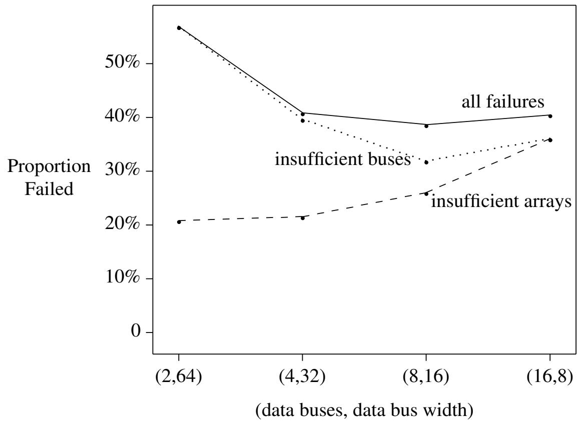



Fig. 16. Failures as a function of data bus width: $B = 64K, N = 16$, 128 data pins.

vary the number of external data buses in an attempt to reduce this effect.

In order to perform fair comparisons, we fix the total number of data pins. In the results in this section, there are 128 data pins; we examine architectures in which these 128 pins are broken into 2, 4, 8, and 16 buses (of 64, 32, 16, and 8 bits respectively). In order to concentrate on the data bus effects, we fix the number of arrays at 16 (each containing 4 Kbits) and the number of address buses at 8 (this is the second architecture from Table III).

Fig. 16 shows the failure rates for the three architectures. As the number of data buses is increased, the failures due to insufficient buses is reduced, as expected. This is offset, however, by an increase in the number of failures due to insufficient arrays. The increase in failures due to insufficient arrays is because as the data width decreases, each array becomes less capable, and thus, more of them are needed to implement wider user memories. For example, in the 16-bus architecture, each bus is only 8 bits wide. This means that the maximum data width (highest value in $W_{\text{eff}}$), and hence the maximum number of data bits that can be extracted from each array, is 8. Thus, a 32-bit wide user memory will require at least four arrays, regardless of the memory's depth. This is in contrast to the 4-bus architecture, in which 32 bits can be extracted from each array. The same wide user memory on this architecture would require only one array (as long as the depth is less than 128). Overall, the number of failures drops by 29% over the range of the graph.

Fig. 17 shows the area and delay results for the same set of architectures. There is an area advantage for architectures with more buses; a narrower bus means fewer sense amplifiers are required in each array. A narrower bus also means a less complex L1 mapping block. These two

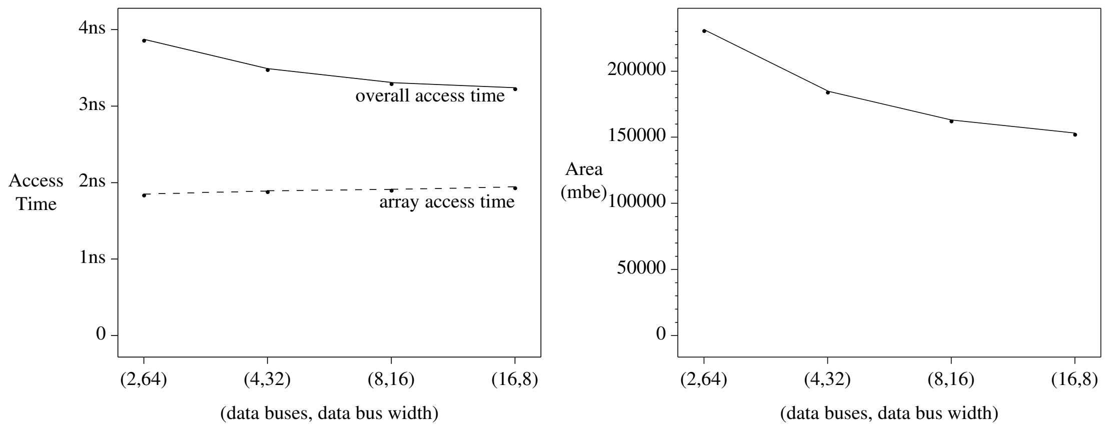



Fig. 17. Delay/area as a function of data bus width: $B = 64K, N = 16$, 128 data pins.

factors offset the fact that the architectures with more external data buses have more complex L2 mapping blocks. Over the range of the graph, the access time drops 16% and the area drops 34%. Thus, we conclude that, in this architecture, either 8 or 16 data buses is the best choice.

## *D. L2 Switch Patterns*

Until now, all results have assumed that the L2 mapping block and address mapping block employ the switch pattern defined by Equation 2. In this section, we compare this pattern to three other patterns; the four patterns are shown in Fig. 18. For each pattern, an equation giving the set of buses to which each array can connect is also given. Intuitively, the more switches within the mapping blocks, the more flexible the overall architecture will be, but at the expense of slower user memory implementations and more chip area.

Table IV shows the flexibility, delay, and area results for the smaller architecture (eight 1-Kbit arrays and four address and data buses). The column labeled "P&R Failures" shows the proportion of all configurations that failed *during* the place-and-route algorithm (clearly, the switch pattern has no effect on the logical-to-physical mapper, so configurations that failed during that phase are not included in the table). Pattern 1 does not provide sufficient flexibility; 85% of the configurations could not be mapped. Pattern 2, which is the pattern assumed in all other results in this section, can be used to implement all but 7.8% of the configurations. The area and delay penalties when moving from pattern 2 to 3 are small (0.3% and 0.7% respectively). With pattern 3, however, every configuration could be mapped. Thus, for this architecture, pattern 3 seems to be the best choice.

Table V shows the results for the larger architecture (16 arrays and 8 address and data buses).

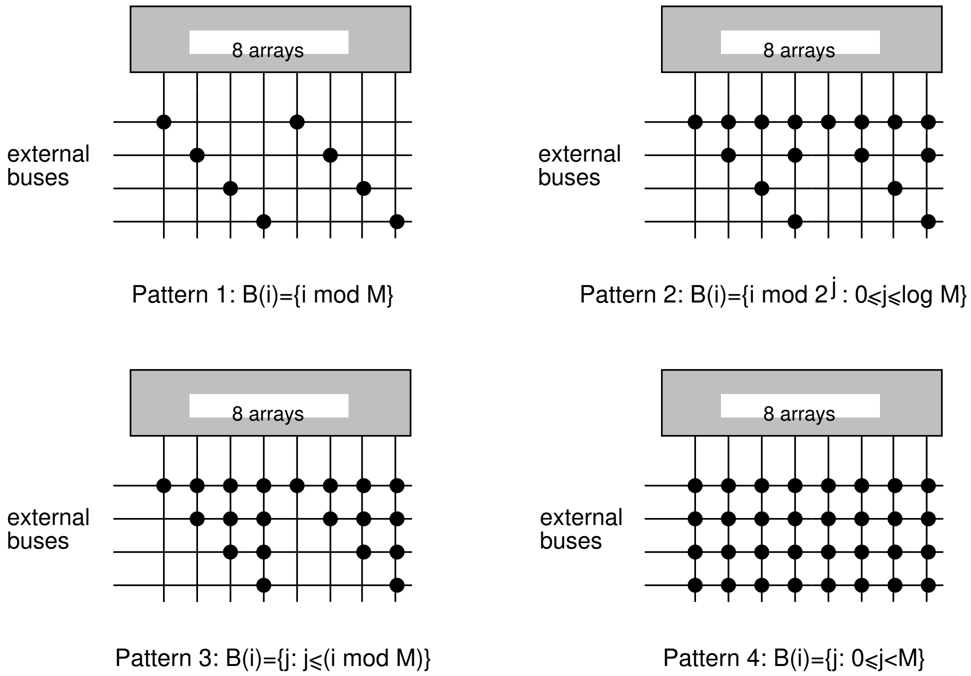



Fig. 18. L2 data block/address block switch patterns considered.

TABLE IV

L2 SWITCH PATTERN RESULTS $(B = 8K, N = 8, M = Q = 4, W_{\mathrm{EFF}} = \{1, 2, 4, 8\})$.

| Pattern | P&R Failures | Total Delay | Total Area |
|---|---|---|---|
| 1 | 84.6% | 3.53 ns | 20841 |
| 2 | 7.79% | 4.06 ns | 20966 |
| 3 | 0% | 4.09 ns | 21029 |
| 4 | 0% | 4.20 ns | 21216 |

TABLE V

L2 SWITCH PATTERN RESULTS $(B = 64K, N = 16, M = Q = 8, W_{\mathrm{EFF}} = \{1, 2, 4, 8, 16\})$.

| Pattern | P&R Failures | Total Delay | Total Area |
|---|---|---|---|
| 1 | 68.7% | 4.13 ns | 137153 |
| 2 | 6.84% | 5.29 ns | 163039 |
| 3 | 0.06% | 5.50 ns | 189050 |
| 4 | 0% | 5.81 ns | 189924 |

TABLE VI

SUMMARY OF ARCHITECTURAL RESULTS.

| Parameter | Area | Access Time | Failures | Best Choice | |
|---|---|---|---|---|---|
| | | | | Arch. 1 | Arch. 2 |
| as $N$ increases | $\uparrow$ | $\uparrow$ | $\downarrow$ | 8 | 16 |
| as $\lvert w_{\text{eff}} \rvert$ increases | approx const. | approx const. | $\downarrow$ | $\{1, 2, 4, 8\}$ | $\{1, 2, 4, 8, 16\}$ |
| as $M$ increases | $\downarrow$ | $\downarrow$ | min. at 8 | - | 8 |
| as num. L2 switches increases | $\uparrow$ | $\uparrow$ | $\downarrow$ | From Eq. 2 | |

For this architecture, the delay penalty from moving from pattern 2 to 3 is 4%, and the area penalty is 17%. The dramatic increase in area is because more sub-buses are required; each of the sub-buses needs a second-level driver. If the number of sub-buses was held constant, the access times for patterns 3 and 4 would be considerably larger than they are. From the results in the two tables of this section, we conclude that the base mapping block (pattern 2) is the best choice, since it is flexible enough to perform most required mappings, but at a lower area and speed cost than patterns 3 and 4 (especially for large architectures).

### *E. Summary of Architectural Experiments*

Table VI summarizes the results of the architectural experiments. In the table, a $\uparrow$ indicates that the corresponding metric (area, access time, or number of mapping failures) increases as the indicated parameter increases, while a $\downarrow$ indicates that the corresponding metric decreases. The final two columns show the best choice for each of the parameters for the two architectures in Table III.

## V. EXTERNAL PIN ASSIGNMENT FLEXIBILITY

The results in the previous section assumed that the place/route algorithms are free to assign any I/O signal to any external pin. In practice, there are many applications in which this is unacceptable. For example,

- In large systems where the configurable memory is one of many chips mounted on a PC-board, once the board has been constructed, the pin assignments of all chips are fixed. If future system changes involve reconfiguring the memory, the new pin assignments must be

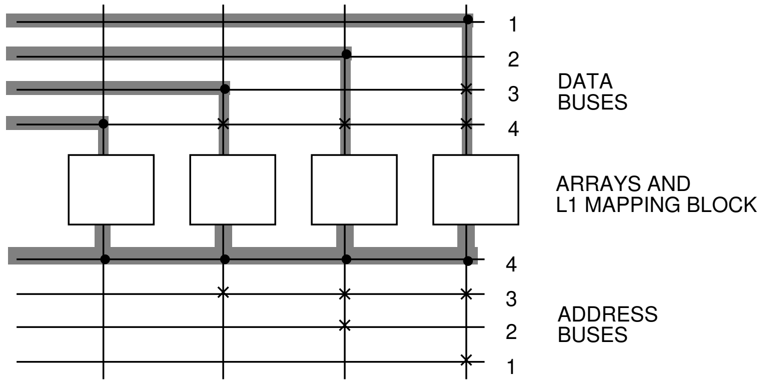



Fig. 19. A single user memory that uses 4 data buses and 1 address bus.

the same as the old assignments.

- In reconfigurable systems which contain FPGAs and configurable memories, although each chip is configurable, the interconnections between these chips are often fixed. If there is little flexibility in the memory pin assignments, severe restrictions will be placed upon the system-level partitioner, placer, and router.

In order to evaluate the viability of the configurable memory in these applications, a measure of pin assignment flexibility is required. This section defines a metric for this flexibility, and evaluates this flexibility for the architecture described earlier.

In any memory, there is complete functional flexibility in the pin assignments within a single address or data bus; i.e. all address pins are equivalent, as are all data pins. In this section, we are concerned with a higher level of flexibility: flexibility in the assignment of buses. As an example, consider Fig. 19. In this example, a single user memory that requires four data buses is implemented. There is no flexibility at all in the address bus assignment; only address bus 4 has enough pins to connect to all four arrays. There is complete flexibility in the data bus assignments, however, since each bus need only connect to one array. If two data buses are swapped, the arrays implementing the corresponding data fields can also be swapped, guaranteeing a successful implementation.

Consider a user configuration requiring $s$ data buses and $z$ address buses mapped onto an architecture with $M$ data buses and $Q$ address buses ($M \geq s$ and $Q \geq z$). If the address mapping block contains switches at every horizontal/vertical bus interconnection, the number of possible address bus assignments is

$$A_{a,max} = \left( \begin{array}{c} Q \\ s \end{array} \right) s! = \frac{Q!}{(Q-s)!}$$

Similarly, if the L2 data mapping block contains switches at every intersection, the number of possible data bus assignments is

$$A_{d,max} = \left( \begin{array}{c} M \\ z \end{array} \right) z! = \frac{M!}{(M-z)!}$$

Combining these two results gives the number of possible bus assignments given complete mapping block flexibility:

$$A_{max} = A_{a,max} * A_{d,max} = \frac{Q!M!}{(Q-s)!(M-z)!}$$

For architectures with mapping blocks that do not contain switches at every intersection, not all $A_{max}$ bus assignments will lead to a valid implementation. Averaging the ratio of the number of valid bus assignments to $A_{max}$ over many user memory configurations gives the *bus assignment flexibility* of a particular architecture. The maximum value of an architecture's bus assignment flexibility is 1.

In order to evaluate the bus assignment flexibility of our architectures, we used an exhaustive algorithm that iteratively steps through all possible address bus assignments and determines whether a valid mapping is possible. Because of the heavy computational requirements of this algorithm, we were limited to studying an architecture with 4 arrays, 4 data buses, and 4 address buses. Nonetheless, this simple architecture provides insight into the behaviour that we expect out of larger configurable memories.

Using the above procedure, we measured the bus assignment flexibility of the base architecture to be 0.0829. In other words, on average, only 8.3% of all possible bus assignments result in valid implementations. For many applications, this is unacceptable.

In order to increase the bus assignment flexibility, we can add switches to the L2 data mapping block and address mapping block. The mapping block considered until now contains 8 intersections with switches and 8 intersections without switches (see Fig. 20). Consider adding switches (a switch-set, since each wire in the bus needs its own switch) to a single intersection. Since there are 8 intersections without switches, there are 8 locations where we can add this switch-set. We experimentally tried each, and the mapping blocks that gave the best results are in Fig. 21 (both

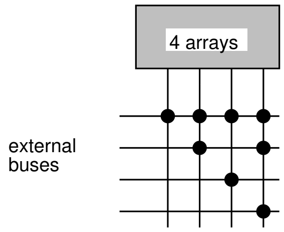



Fig. 20. Base switch pattern for address and L2 data mapping blocks.

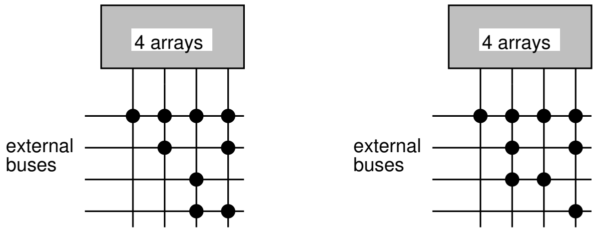



Fig. 21. Best switch patterns if one switch-set is added.

blocks shown are equivalent). If either of these mapping blocks are used, the bus assignment flexibility approximately doubles to 0.157. The area and speed overhead in adding one extra set of switches is small, so clearly, for many applications, either of the mapping blocks in Fig. 21 results in a better architecture.

We can go further and add switches to two intersections. There are 28 ways these two switch-sets can be added; the best combination is shown in Fig. 22. This mapping block gives an bus assignment flexibility of 0.217.

This process was repeated for 3,4,5,6,7, and 8 switch sets. The graph in Fig. 23 gives the bus assignment flexibility for each number of switch sets. For each point, all possible switch patterns were considered, and the best chosen. As the graph shows, the curve is roughly linear, meaning there is no point beyond which adding switches gives little benefit. Thus, we conclude that in order to get a bus assignment flexibility of close to 1, the mapping block should be fully populated.

Note that there are two points on the graph that stand out: 3 and 6 switch sets. Fig. 24 shows the mapping block patterns corresponding to these points. Fig. 24(a) is the first pattern which has two horizontal buses that can connect to all 4 arrays; Fig. 24(b) is the first pattern containing three horizontal buses that can connect to all 4 arrays. Logical memory configurations in which all arrays must be connected to each other are common; thus, the more buses on which this can be done, the higher the bus assignment flexibility.

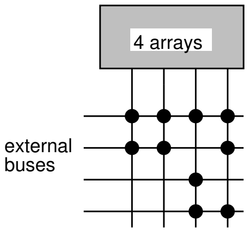



Fig. 22. Best switch pattern if two switch-sets are added.

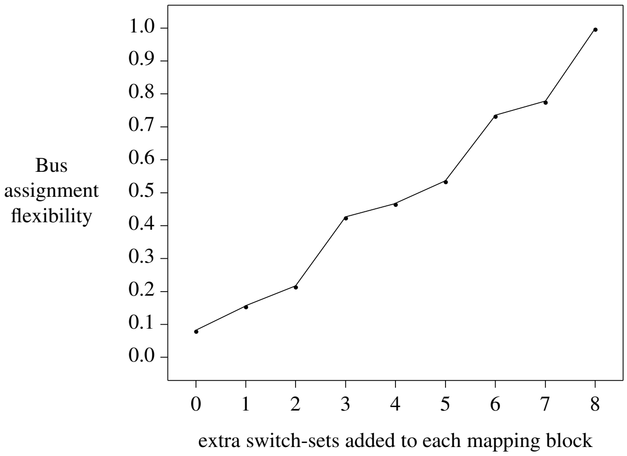
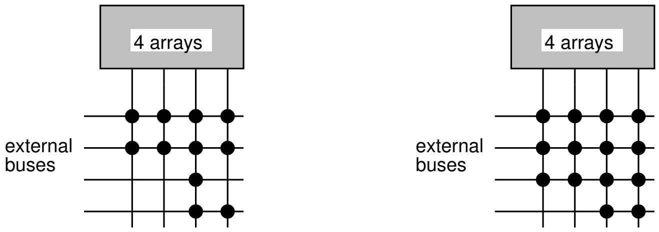



Fig. 23. Bus assignment flexibility results.

## VI. Conclusions

In this paper, we have described a stand-alone configurable memory architecture consisting of an interconnected set of arrays. There are two levels of configurability: the effective data width of each array and the interconnection between these arrays. Together, these result in an architecture that is flexible enough to implement a wide variety of user memory configurations.

We have also described a set of experiments aimed at determining how various architectural parameters affect the chip area, access time, and flexibility of the device. We have concentrated




a) Three switch-sets added b) Six switch-sets added

Fig. 24. Best switch patterns if 3 and 6 switch-sets are added.

on two architectures: one containing 8Kbits and the other containing 64Kbits. For the first architecture, we found that dividing the 8Kbits into 8 arrays, and connecting these arrays to each other and to the chip pins using 4 data buses is the optimum choice. In the second architecture, we found that dividing the 64Kbits into 16 arrays, and connecting these arrays and the I/O pins using 8 data buses results in the most efficient architecture. We have also presented a mapping block topology that provides near-perfect flexibility, assuming the mapping tools are free to make external pin assignments. If the device is to be used in an application in which I/O pins must be pre-assigned, however, a mapping block in which every bus can connect to every array is required.

Finally, although this paper has concentrated on stand-alone configurable memories, the architecture is suitable for inclusion as embedded memory resources within an FPGA. The most important issue in embedded memory resources is the interface between the memory and logic parts of the FPGA. If this interconnection is not flexible enough, much of the native flexibility of the configurable memory will be lost. On the other hand, an overly flexible interconnect will result in excessive overhead in terms of chip area and circuit speed. These issues are explored further in [12]. Embedding configurable memory arrays in an FPGA will likely have additional advantages; we expect that some logic circuits will map very nicely into memory arrays. The investigation of this possibility is an active area of future work.

## Acknowledgments

Financial support was provided by MICRONET, the Natural Sciences and Engineering Research Council of Canada, the Walter C. Sumner Memorial Foundation, and UBC's Centre for Integrated Computer Systems Research (CICSR).

## References

[1] J. M. Arnold and D. A. Buell, "Splash 2," in *Proceedings of the 4th Annual ACM Symposium on Parallel and Distributed Algorithms and Architectures*, pp. 316–324, 1992.

[2] P. Bertin, D. Roncin, and J. Vuillemin, "Programmable Active Memories: A Performance Assessment," in *Research on Integrated Systems: Proceedings of the 1993 Symposium*, MIT Press, 1993.

[3] D. E. van den Bout, J. N. Morris, D. Thomae, S. Labrozzi, S. Wingo, and D. Hallman, "Anyboard: An FPGA-based, reconfigurable system," *IEEE Design and Test of Computers*, pp. 21–30, September 1992.

[4] Altera Corporation, *Reconfigurable Interconnect Peripheral Processor (RIPP10) Users Manual*, May 1994.

[5] D. Galloway, D. Karchmer, P. Chow, D. Lewis, and J. Rose, "The Transmogrifier: the University of Toronto field-programmable system," in *Proceedings of the Canadian Workshop on Field-Programmable Devices*, June 1994.

[6] D. Karchmer and J. Rose, "Definition and solution of the memory packing problem for field-programmable systems," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 20–26, 1994.

[7] Altera Corporation, *Datasheet: FLEX 10K Embedded Programmable Logic Family*, July 1995.

[8] Actel Corporation, *Datasheet: 3200DX Field-Programmable Gate Arrays*, 1995.

[9] Actel Corporation, *Actel's Reprogrammable SPGAs*, 1996.

[10] Xilinx, Inc., *XC4000 Series (E/L/EX/XL) Field Programmable Gate Arrays v1.04*, Setpember 1996.

[11] AT&T Microelectronics, *Data Sheet: Optimized Reconfigurable Cell Array (ORCA) Series Field-Programmable Gate Arrays*, March 1994.

[12] S. J. E. Wilton, *Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memory.* PhD thesis, University of Toronto, 1997.

[13] S. J. E. Wilton, J. Rose, and Z. G. Vranesic, "Memory-to-memory connection structures in FPGAs with embedded memory arrays," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 10–16, February 1997.

[14] T. Ngai, J. Rose, and S. J. E. Wilton, "An SRAM-Programmable field-configurable memory," in *Proceedings of the IEEE 1995 Custom Integrated Circuits Conference*, pp. 499–502, May 1995.

[15] S. J. E. Wilton, J. Rose, and Z. G. Vranesic, "Architecture of centralized field-configurable memory," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 97–103, 1995.

[16] J. Rose, R. Francis, D. Lewis, and P. Chow, "Architecture of programmable gate arrays: The effect of logic block functionality on area efficiency," *IEEE Journal of Solid-State Circuits*, vol. 25, pp. 1217–1225, October 1990.

[17] J. Rose and S. Brown, "Flexibility of interconnection structures for field-programmable gate arrays," *IEEE Journal of Solid-State Circuits*, vol. 26, pp. 277–282, March 1991.

[18] S. Brown, G. Lemieux, and M. Khellah, "Segmented routing for speed-performance and routability in field-programmable gate arrays," *Journal of VLSI Design*, vol. 4, no. 4, pp. 275–291, 1996.

[19] D. Hill and N.-S. Woo, "The benefits of flexibility in lookup table-based FPGA's," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, pp. 349–353, February 1993.

[20] J. L. Kouloheris and A. E. Gamal, "PLA-based FPGA area versus cell granularity," in *Proceedings of the IEEE 1992 Custom Integrated Circuits Conference*, pp. 4.3.1–4.3.4, 1992.

[21] M. Hutton, J. Grossman, J. Rose, and D. Corneil, "Characterization and parameterized random generation of digital circuits," in *Proceedings of ACM/IEEE Design Automation Conference*, pp. 94–99, June 1996.

[22] J. Darnauer and W. W. Dai, "A method for generating random circuits and its application to routability measurement," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 66–72, Feb. 1996.

[23] K. Veenstra. private communications, 1995.

[24] S. J. E. Wilton and N. P. Jouppi, "CACTI: an enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 677–688, May 1996.

[25] M. M. Khellah, "Minimizing interconnection delays in array-based FPGAs," Master's thesis, University of Toronto, 1994.

[26] J. M. Mulder, N. T. Quach, and M. J. Flynn, "An area model for on-chip memories and its application," *IEEE Journal of Solid-State Circuits*, vol. 26, pp. 98–106, Feb. 1991.