Appears in *2002 IEEE Symposium on Field-Programmable Custom Computing Machines, April 2002.*

Mobile Memory: Improving memory locality in very large reconfigurable fabrics

Rong Yan and Seth C. Goldstein
Carnegie Mellon University
{yanrong,seth}@cs.cmu.edu

Abstract

As the size of reconfigurable fabrics increases we can envision entire applications being mapped to a reconfigurable device; not just the code, but also the memory. These larger circuits, unfortunately, will suffer from the problem of a growing memory bottleneck. In this paper we explore how mobile memory techniques, inspired by cache-only memory architectures, can be applied to help solve this problem. The basic idea is to move the memory to the location of the accessor. Using both an analytical model and simulation we investigate several different memory movement algorithms. The results show that mobility can, on average, decrease memory latency 2x; which translates into speedup of about 15%.

1 Introduction

As reconfigurable fabrics grow in capacity and speed we can start to think about executing entire applications on them. In the past, FPGAs may have contained the code and some temporary memory for a kernel, or maybe even the code for an entire application, but never the code and memory for an entire application. However, with the advent of million-gate FPGAs that also include substantial memory resources we can begin to think of using this rich resource to overcome the processor-memory gap. Furthermore, some researchers are beginning to look at alternative implementation technologies which will produce multi-billion gate FPGAs [4]. Recent work has shown that such systems will be able to store entire applications and data sets on a single die.

When looking at such a system the main obstacle to high performance is memory latency [4]. Even in the best of all worlds, where memory addresses are known at compile time and all the memory can fit on a single chip, the latency to access memory from one part of the chip to another imposes a substantial delay. In this paper we examine how technology borrowed from shared memory multiprocessors can be applied to very large reconfigurable fabrics to reduce the cost of memory accesses. Specifically, we look at whether techniques used in cache-only memory architectures [3] (COMA) can be applied to reconfigurable devices to reduce memory latency when entire applications are mapped to the reconfigurable device.

The motivation for this work came from looking at the behavior of programs compiled to a reconfigurable fabric with 10^{10} gates [4]. Each program was converted into a circuit and mapped to the reconfigurable device. Furthermore, the memory was also mapped to the reconfigurable device. Figure 1 shows an example of a very small program mapped to such a device. Each square represents a cluster of the underlying device. The shading of the squares indicates how much of the area is taken up by instructions. A white square contains only code; a dark square contains only data. The edges show communication patterns. The edge width is the logarithm of the number of messages sent across the edge. The edge color indicates the mix of types of messages: dark edge indicates memory reads only, while lighter edges indicates control transfers, with intermediate shadings for edges which carry mixed traffic. Despite the graph being very small, it exhibits some typical features for all our programs, like the big "stars": code regions which touch most of the memory of the program. "Stars" are bad, because there is no way to place all adjacent nodes close to the star's center node; some have to be remote. "Hot" memory locations, which are touched by a lot of basic blocks, are less common. This lead us to wonder if we couldn't improve the total performance by dynamically moving the memory closer to the location where it was accessed. In other words, to treat all the memory as a COMA system, moving memory to accessors on demand.

We investigate several different methods for implementing mobile memory systems. We find, that there is a potential to significantly speed-up applications with simple methods. In the next section of the paper we describe some of the work done in COMA systems; the ancestors to our mobile memory system. In Section 3 we define the problem and describe the different techniques that we investigate. In Section 4 we develop an analytical model to determine the best possible behavior that we can expect

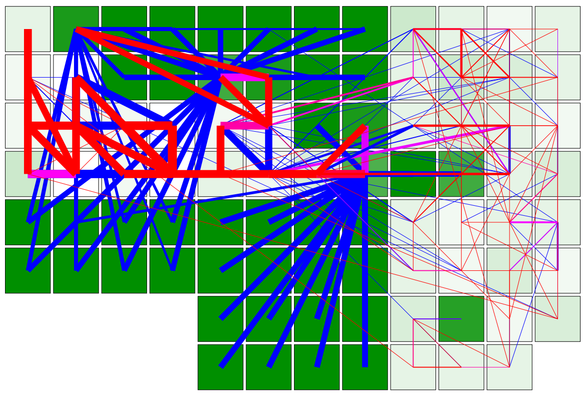
Figure 1: *A placed graph for g721_e.*

from mobile memory systems. In Section 5 we evaluate the different techniques using various simulations.

2 Background

The work which most inspired our study occurred around a decade ago in the context of multiprocessor systems. Cache-Only Memory Architectures (COMA) [3, 6, 5] were introduced as an alternative to message passing and shared memory architectures. In a COMA, virtual pages have no physical "home", and main memory acts as a huge cache. The hardware transparently migrates and replicates the data to the current accessor, adapting dynamically to the memory access pattern and hopefully placing the data closer to the subsequent accessors. As an alternative of COMA, Verghese [11] has augmented NUMA-RC machines with additional operating system support for dynamic page migration, aimed at reducing the remote memory access latencies.

Although these techniques are aimed at multiprocessors they have much in common with the very large reconfigurable fabrics we are considering. In fact, if we treat each little piece of code on our fabric as a processor, then we implement, on a single fabric a massively parallel system. However, very large reconfigurable fabrics have many advantages over multiprocessor systems: a more flexible topology, faster signalling, more available bandwidth, and the ability to create custom memory controllers.

2.1 Compilation

We treat very large reconfigurable fabrics (VLRFs) as a mixture of configurable logic blocks, memory arrays, and routing. We are necessarily abstract because our aim is to create a compilation framework for these fabrics that scale to large applications and large fabrics. The compilation process starts by partitioning the application into a collection of threads. Each thread is a sequence of instructions ending in a split-phase operation. An operation is deemed to be a split-phase operation if it has an unpredictable latency. For example, memory references and procedure calls are all split-phase operations. Thus, each thread, similar in spirit to a Threaded Abstract Machine (TAM) thread [2], communicates with other threads asynchronously using split-phase operations. This partitioning allows the CAD tools to concentrate on mapping small isolated netlists. Furthermore, this model, which we call the split-phase abstract machine (SAM) model, has all the mechanisms required to support thread-based parallelism.

Unlike a traditional thread model, where a thread is associated with a processor when executing, each SAM thread will be a custom "processor." While it is possible for a thread to be complex and load "instructions" from its local store, the intention is that it remains fairly simple, implementing only a small piece of a procedure. This allows the threads to act either in parallel or as a series of sequential processes. It also reduces the number of timing constraints on the system.

The SAM model is a simplification of TAM which supports multithreading. However, unlike in TAM, we use our model only as a way of partitioning large sequential programs in space on VLRFs. So, while SAM can support parallel computation, a parallelizing compiler is not necessary. The performance of this model rests on the ability to create custom processors. Later, as the compiler technology becomes more mature, the inherently parallel nature of the model can be exploited.

The SAM model explicitly hides many important details. For example, it neither addresses dynamic routing of messages nor allocation of stacks to the threads. Once an application has been turned into a set of cooperating SAM threads it is mapped to a more concrete architectural model which takes these issues into account. The mapping process will, when required, assign local stacks to threads, insert circuits to handle stack overflow, and create a network for routing messages with runtime computed addresses. For messages with addresses known at compile time it will route signals directly.

2.2 Target fabric

Once an application has been partitioned into a collection of SAMs, we estimate the area and simulate the applications using the simplest SAM model, i.e., only one thread executes at a time. We perform no special optimizations aimed at reconfigurable fabrics, e.g., we have not pipelined loop bodies, etc. We assume a 2-d mesh for communication and do not add overhead for routers. We define one unit area of our fabric to be the size required to implement a 32-bit adder. We then assume that it takes the same amount of area to implement 32 bits of memory. We intentionally underestimate the amount of space required

Sequence 1: **a a a a** b **a a a a a a a a a a** b **a a**
Migratory Access Pattern
Sequence 2: **a b c a b c** d **c b a a b c** e **a b c**
Group Access Pattern
Sequence 3: **a b c d e f g a g j k c d j a k c**
Unpredicatable Access Pattern

Figure 2: *Classification of Memory Access Pattern*

to implement each memory word. We use the extra space to hold the circuits needed to access the memory; essentially amortizing the cost of the memory units over all the memory words. Floating point units, multipliers, etc. are all scaled appropriately. We group one hundred units into a cluster. In Figure 1 each square is a cluster.

We assume that the underlying fabric can execute an elementary 32-bit operation (e.g., add, or) in one cycle. Write operations are asynchronous and take one cycle. In other words, no acknowledgment is returned when the write completes. We distinguish two kinds of read operations: local and remote. A local read takes place completely within one cluster. A remote read crosses clusters. Local reads take one cycle. Remote reads take one cycle to initiate and then time proportional to the round trip Manhattan distance between the accessor cluster and the cluster of the memory location.

3 Mobile Memory

Mobile memory aims at reducing memory latency by exploiting locality at runtime. A traditional approach to reducing memory locality is to introduce caches which are close to the processors and duplicate the contents of the main memory. We do away with the main memory and utilize the reconfigurable fabric to make all of memory a set of distributed caches. However, we do not ever duplicate memory. By avoiding memory duplication we eliminate the cost of maintaining coherency. The question we set out to ask is whether this approach, moving memory, but never duplicating it, is fruitful in the context of very large reconfigurable fabrics.

An example where mobile memory by itself is not beneficial is when the memory access pattern has a ping-pong character to it. For example, when two processors, A and B, alternate accesses to a particular memory location. Memory access patterns have been well studied [12, 1, 10]. Memory access patterns can be classified into one of three types: migratory, group, or unpredictable.

Migratory access pattern: occurs when a memory location is accessed primarily by a single accessor. This pattern is shown in the first sequence in Figure 2, where processor 'a' makes most of the memory accesses to a particular location. This pattern is the best candidate for mobile memory since the cost of migrating the memory to a processor is recouped by repeated accesses from that same processor.

Group access pattern: occurs when a memory location is accessed by a small group of processors. Sequence 2 in Figure 2 exhibits the group access pattern. If only two processors are involved then this devolves into the ping-pong access pattern. For this pattern migrating memory may or may not be beneficial.

Unpredictable access pattern: When the memory reference string is not one of the above two types we call it unpredictable. When the memory references are unpredictable, migrating the memory to a particular accessor will not provide any benefit.

As we will see later, most memory locations exhibit either migratory or group access patterns. We thus design our mobile memory system around these two patterns.

3.1 Policies

A mobile memory system has three main design axes: when, where, and how much.

When determines when to move the memory. What triggers the migration. In all of the systems we analyze, the trigger is the reference by a remote processor. One could imagine other triggers based on prefetching or the behavior of neighboring memory.

Where determines the destination of a memory migration. The optimal destination is not necessary any of the previously accessors. In the case of the unpredictable access pattern it may be a completely new location. In the case of the group access pattern it may be the centroid of the group of accessors. Without omniscience we must use the historical access pattern to make a decision about the future access pattern. We examine several different heuristics and also analyze how much history should be kept for choosing the best destination location.

How much refers to how much memory should be moved at once. Since many programs exhibit both temporal and spatial memory locality it can be beneficial to move groups of memory words together. We briefly examine moving more than a single word.

We now present three simple, but effective, policies for mobile memory. In the following, N is the amount of history we keep.

Greedy The greedy policy moves the memory so that it is co-resident with the most recent accessing processor. This policy preforms well if the memory access pattern is migratory. Otherwise, it performs poorly for two reasons. First, it moves memory too often. Second, it can actually increase the total latency by moving memory away from future accessors.

N-best The N-best policy keeps track of the last N accessors. It tries to work well for both the migratory and group access patterns. Under this policy the memory is moved to the processor, among the last N, that minimizes the total memory access time assuming the access pattern were to repeat itself. This policy devolves into the greedy policy if $N = 0$. As N increases this policy tends to migrate memory less, reducing unnecessary memory movement, but slowing down the benefit for the migratory access pattern.

Centroid The centroid policy moves memory to the centroid of the previous N accesses. Unlike the N-best policy it does not require that the memory be colocated with one of the previous accessors. This policy is likely to move the memory more often than N-best, but to move it far less each time. As with the N-best policy, when $N = 0$ this degrades into the greedy policy.

3.2 Costs

Mobile memory does not come for free. We divide the cost of implementing mobile memory into five areas: statistics gathering, decision support, directories, movement, and pressure.

In order to implement the N-best and Centroid policies, we must keep track of the previous N accessors. This data can probably not be kept on a per word basis. Instead we group together all the words in a cluster and have them share the same history information. Thus, we actually implement a system where the destination of an individual memory word is a function of the cluster's N last accessors, not the word's last N accessors. The source for history information is another interesting issue. Memory words can collect the history information in different ways. Three possible history information collection methods are investigated:

Home indicates each memory word uses the history information of its home cluster, and updates the home cluster's history when the memory word is moved.

New-Cluster indicates each memory word uses the history of the cluster to which it is moved. It updates the destination cluster's history with the current accessor when the word is moved.

Copy-History The moved word brings the history of its current cluster with it to the cluster to which it moves. Otherwises this method is the same as the New Cluster method.

The decision support hardware is also shared by all the words in a cluster. For N-best the amount of hardware needed is prohibitively expensive. For the centroid policy the hardware scales linearly with the amount of history used. For a history of N accessors, $2(N+1)$ adders and 2 dividers are necessary.[1]

The cost of the directories scales with the amount of memory in the system. Furthermore, there is a latency cost when a memory word has moved, a processor will send a request to the memories' home location, only to have it forwarded to its current location. Turning a round-trip access time into the time to goto the directory, the new location, and then back to the accessor. To capture this cost, we assume each cluster maintains two directories. One is the home directory, which contains the information for memory words whose home location are this cluster. The other is the local directory that contains the memory words which is recently accessed by this clusters. The home directory cannot discard any of its entries at runtime, however, the local directory can be more flexible, because it can discard any data when necessary. We use First-In-First-Out(FIFO) algorithm to manage the replacement of local directory entries. Moreover, when the information of the accessee is missing in current local directory, the accessor will check the home directory and update the information of accessee into its local directory.

Actually moving the memory has a cost in providing enough bandwidth to accommodate extra information. When prefetching is used to move more than the requested amount of memory this requires more bandwidth.

Finally, there is the cost of making room for mobile memory. When memory is moved from one location to another there must be room at the target to accommodate the new memory. As the total amount of allocated memory in system increases, the memory pressure increases. At some point room will have to made for mobile memory by evicting memory back to its home location. One way to reduce the chances that eviction will take place is to reserve a portion of memory near all the processors to hold mobile memory. This will cause the entire circuit to grow increasing the latency for all operations.

4 Analytical Model

In this section, we derive two lower bounds. One which determines the lower bound on the number of cycles needed to perform memory accesses and the other which determines the total application speedup that could be expected as a function of memory access time. These lower bounds will help to evaluate our heuristics.

[1]If (N+1) is a power of 2, the dividers are unnecessary.

4.1 Minimal memory cycles

As discussed previously, it is not possible to completely eliminate all the memory latency in an application. For example, when a memory location is accessed with the ping-pong pattern the memory overhead is impossible to reduce using mobile memory. So, here we use an off-line algorithm to determine the minimum number of cycles needed to perform all the memory accesses in a program. We first examine the off-line algorithm for moving only the requested word, then we look at moving more than one word at a time.

In calculating the minimum we assume that there are no extra costs introduced by the mobile memory policy. In other words, the cost of a read is proportional to the round trip time plus one cycle to initiate the read. The extra costs of mobility are all zero, i.e., the cost of deciding where to move the memory, the cost of moving the memory, the cost of finding room for the memory are all zero.

We further decompose the problem into calculating for each memory word the minimal number of memory cycles needed to access that word. Considering a single memory word M, if it has been accessed n times in total during the runtime, its accessor string can be represented as $A_1, ..., A_n$. Thus, the problem can be reduced to this: minimize the total memory access cycles $\sum_t(Mem(A_t, M))$ for each memory word. We derive the minimum using the following dynamic programming algorithm:

1. In the following, we only consider the memory reference to a single memory word M. M can be moved to any cluster after each memory access. Each cluster is identified with a unique integer, where C_i refers to cluster *i*.

2. Traverse the memory access in the reference sequence. Define Mem_t^j as the minimal accumulated memory cycles for the first *t* accesses, if M is in cluster C_j after the access from A_t.

    (a) Iterate though all clusters to calculate the Mem_t^j. Suppose M is in cluster *i* before the access from A_t and prepared to move to cluster *j* afterwards. We assume the memory movement executes as follows: accessor A_t sends a signal to cluster *i*, M moves from cluster *i* to cluster *j*, and cluster *j* sends back the data to accessor A_t. Then we have

    $$Mem_t^j = min_i(Mem_{t-1}^i + Mem(A_t, C_i)$$
    $$+Mem(C_i, C_j) + Mem(C_j, A_t))$$

    where $Mem(x, y)$ is the number of cycles needed for memory access between x and y.

    (b) Let $Mem_t = min_j(Mem_t^j)$ be the minimal number of cycles needed to access M up to the access from A_t. Plus,Mem_n is the minimal number of cycles to access M.

We can get the minimal memory access cycles, Mem_{MIN} by adding minimal access cycles over all memory words together. As we will see in Section 5 our heuristics are often very competitive with this offline minimum. However, by increasing the amount of memory moved on each request, we can lower the total memory access time. A very loose lower-bound when n words are moved together is $Mem'_{MIN} = Mem_{MIN}/n$. This bound is not that useful, but shows that when you access more than one word at a time, you can improve on the offline algorithm.

4.2 Memory Access Speedup and Total Speedup

Here we examine how significant memory access time is to total performance. We approximate this by examining how much of the memory latency is on the execution's critical path. We call c the application's *critical communication ratio*, which is the ratio of memory accesses that are on the critical path. As c approaches one, the percentage of memory accesses on the critical path increase and thus reducing memory access cycles benefits the overall performance dramatically.

The model includes several parameters: MV, $BASE$, MIN are respectively total memory access cycles with a specified mobile memory policy, baseline memory cycles without mobile memory and the minimal memory cycles achievable by mobile memory. f is the optimization factor of memory access cycles, defined by $f = \frac{MV-MIN}{BASE-MIN}$. A ratio close to 0 indicates that the existing moving policy is good at exploiting reference locality and reducing large portion of memory latency.

Then we can write the memory access speedup and total speedup as

$$Speedup_{mem} = \frac{BASE}{MV} = \frac{BASE}{MIN + f*(MV - MIN))}$$

$$Speedup_{total} = \frac{Total_{BASE}}{Total_{MV}} = \frac{1}{(1-c) + c/Speedup_{mem}}$$

This model gives us more theoretical insight. f is the only parameter changed with different memory moving policies in the model. In the limit, when the memory access performance is fully optimized ($f = 0$), the total speedup reaches its lower bound $((1 - c) + c * MIN/BASE)^{-1}$. With larger memory ratio c and lower minimal distance MIN, this lower bound becomes lower and therefore memory mobility will be more profitable.

5 Results

In this section we present the results of our limit study on mobile memory in very large reconfigurable fabrics. We evaluate the performance of several different heuristics. For the most part we are optimistic and assume the cost of implementing mobile memory is insignificant.

5.1 Simulation Methodology

The basis for our simulations is a trace-based simulator developed to study very large reconfigurable fabrics. Applications are compiled using gcc (with -O2 optimization) and instrumented with ATOM [8] to produce a memory access trace along with the information of control transfer. Using this information the application is placed in space on a two-dimensional reconfigurable fabric. (See [4] for details.) We implement the different mobile memory algorithms in our simulator.

All applications were run to completion in our simulation. Table 1 presents the 12 applications used in this study and some of their overall characteristics. The programs are the ones from MediaBench [7] and SpecInt95 [9] that could be run in our simulator. The last column of Table 1 shows the application's critical communication ratio. The higher this number the more performance improvement we should expect.

In all the experiments presented below we collect history information on a per cluster basis. When a memory word moves to a new cluster, it can use the history of either the new clusters or its home cluster as its own. In most of our experiments the memory word will use the information from its home cluster. We first present data where the signal propagation time is 1 cluster per cycle. Later we show how slowing down the clock affects performance as we examine a system with a propagation time of 5 cycles per cluster. Finally, we assume that the hardware necessary to implement the algorithm is contained in the memory clusters. This is overly optimistic for N-best, and within reason for the greedy and centroid policies.

5.2 Basic Policies

To get a feeling of how the mobile memory performs, we can compare the results of Figure 3 and Figure 4. These two figures show the breakdown of execution time without mobile memory Figure 3 and when using greedy policy Figure 4. By exploiting more memory locality, mobile memory considerably reduces the portion of idle time and hides more memory latency. Consequently, mobile memory achieves better performance than the baseline.

Figure 5 shows the simulated memory access cycles for the different mobile memory policies and average performance over all the benchmarks . For each application

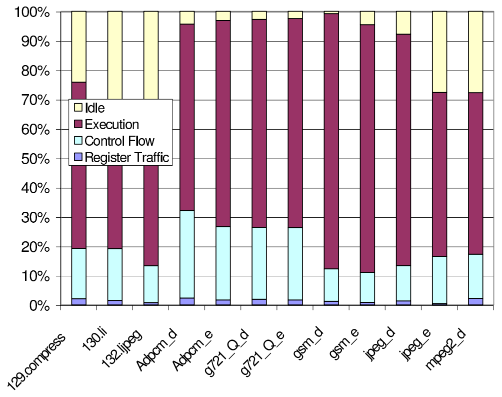

Figure 3: *Breakdown of execution time when no memory movement.*

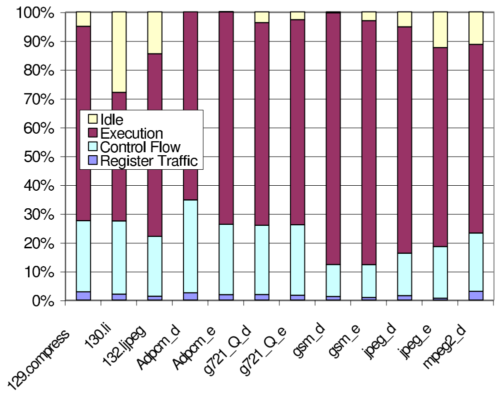

Figure 4: *Breakdown for execution time when greedy policy is used and communication delay is one cycle.*

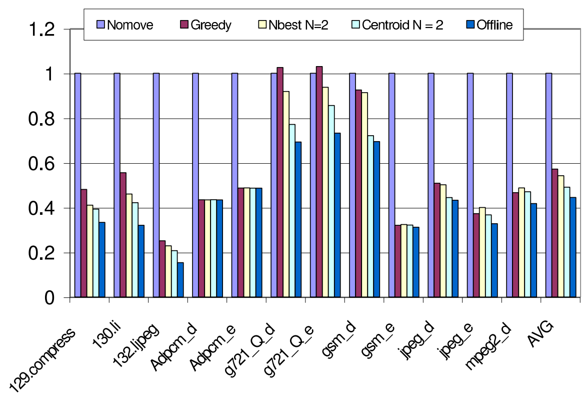

Figure 5: *Normalized memory access cycles for different policies (N = history depth)*

we compare five different policies: Nomove, indicates the base case where memory is not mobile, Greedy, indicates the greedy policy, N-best (N=2), indicates the N-best policy with a history of two, Centroid (N=2), is the centroid policy with a history of two, and offline; indicates the result of using the offline algorithm. Each bar in the figure represents the total access time normalized to the base case.

Benchmark	Source	# of inst.	# of Memory Op.	Critical ratio
Adpcm_d	Media	6070419	517719	0.13
Adpcm_e	Media	8224114	517719	0.06
jpeg_d	Media	6136507	1152922	0.16
mpeg2_d	Media	12797633	2119373	0.45
jpeg_e	Media	19759189	4011225	0.39
gsm_d	Media	85631813	8057881	0.07
129.compress	SPEC	56426146	12123830	0.45
g721_Q_e	Media	269339817	21722179	0.07
g721_Q_d	Media	254421686	22178927	0.05
130.li	SPEC	245501297	56895759	0.54
gsm_e	Media	3348820690	73954764	0.05
132.ljpeg	SPEC	1878366	332756525	0.52

Table 1: *Benchmark Information*

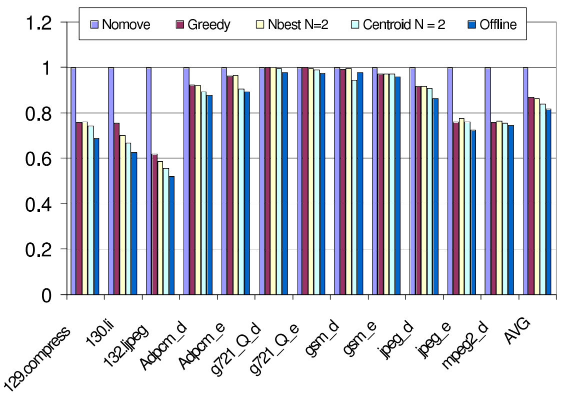

Figure 6: *Total execution cycles for different moving policies normalized to the base case. (N = history depth)*

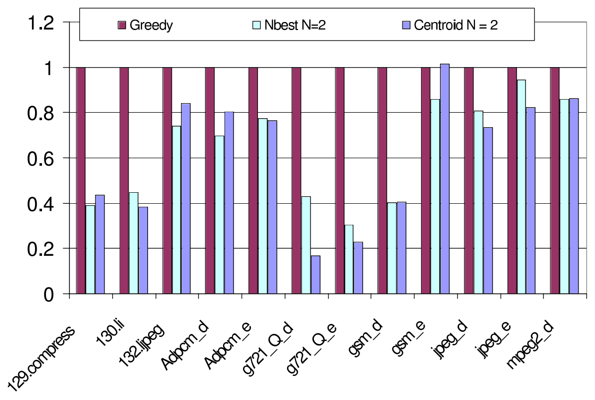

Figure 7: *Normalized memory moving distance for different moving policies. (N = history depth)*

Our first observation from Figure 5 is that mobile memory can dramatically reduce the memory access time. As seen in the figure all the mobile moving policies reduce memory access time in all but two cases. Among our policies, the Centroid policy is best as it reduces memory access time significantly. The greedy policy performs poorly for g721_Q_d and g721_Q_e as it suffers from thrashing problem generated by the group access pattern. The NBest and Centroid policies address this problem by being more conservative about moving memory and in the later case moving it to a better position.

We also observe from Figure 5 that our heuristics approach the offline algorithm. In six benchmarks, the greedy algorithm is only 1%-5% worse than the offline algorithm.

A reduction in memory access time translates to an improvement in performance as shown in Figure 6. Again, the bars are normalized to the base case. The total improvement in running time is dictated by both the critical ratio (shown in Table 1) and the improvement in memory access time. Thus, programs like mpeg2_d and 130.li with high critical ratios and substantial reductions in memory access times show dramatic improvements. While, gsm_d which has a large reduction in memory cycles has only a 2% reduction in the total running time because its critical communication ratio is only 0.07.

Figure 7 examines the relative amount of memory movement for the three heuristics. Notice that the N-best and Centroid polices move memory less often due to their conservative approach towards deciding whether to move memory.

In summary we see that the Centroid moving policy works quite well. It approaches the performance of the offline algorithm while keeping the total memory movement down. We now look at several different aspects of the Centroid policy.

5.3 Sensitivity to History

The amount of history information kept influences the performance of the algorithm in two ways. More history should lead to more accurate decision. However, it will require more hardware and impose additional costs on the memory system. Figure 8 shows how the effectiveness

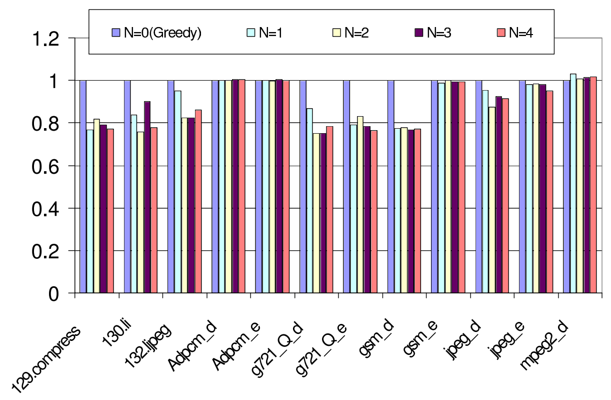

Figure 8: *Comparison of different amounts of history on the performance of the Centroid policy. Bars show the normalized cycle count. N = size of history.*

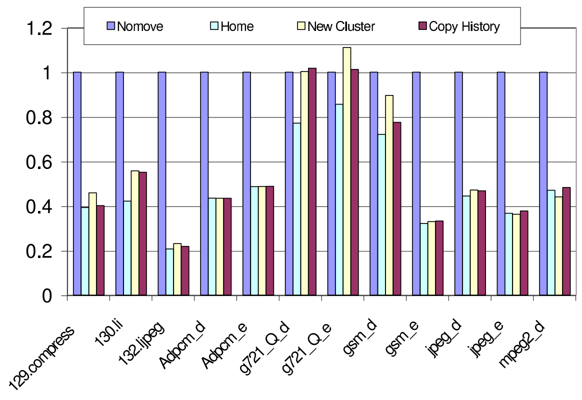

Figure 9: *Normalized memory access cycles when updating history with the information from different sources. All the memory words in a cluster share the same history. Centroid policy is used with a history of 2*

of the centroid policy changes as we vary the amount of history kept. Each bar in Figure 8 represents the memory access cycles normalized to the greedy policy. Interestingly, there is no apparent trend relating performance to history sizes. In fact, the performance of the policy seems relatively insensitive to the amount of history kept. For the following experiments we choose a history size of 2.

We also examine the effect of different information sources. Figure 9 compares how the different history information sources affect the performances. Each bar in Figure 9 represents the memory access cycles normalized to the baseline. Three possible history information collection methods are investigated: Home, indicates home policy, New Cluster, indicates new-cluster policy, and Copy History, indicates copy-history policy. In three of our benchmarks, 130li, g721_Q_d and g721_Q_e, New Cluster and Copy History produce poorer performance than Home, but in other benchmarks, these three have comparable performances. Although home policy has best performance, it have to obtain the home clusters' history for each memory movement , which hurts its performance.

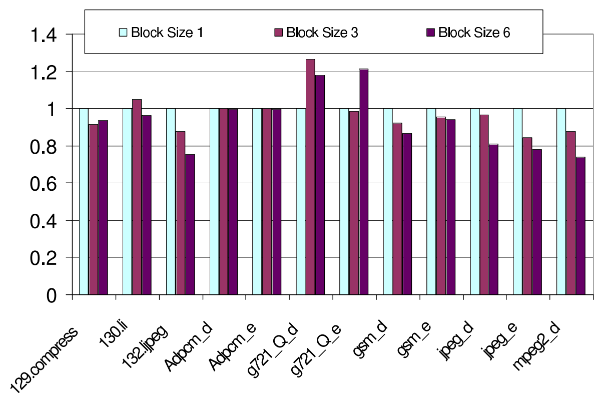

Figure 10: *Normalized cycle counts for different block sizes using the centroid policy.*

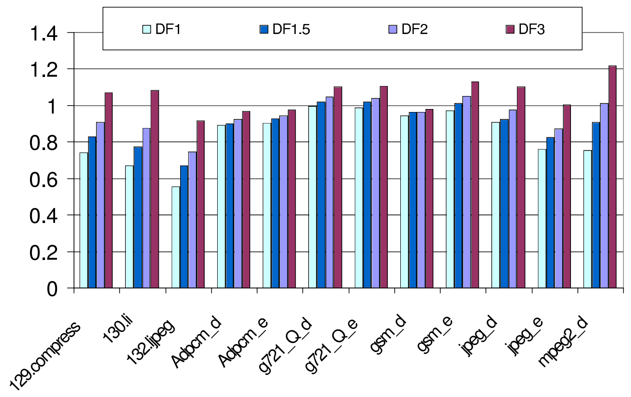

Figure 11: *The effect on total execution time when room for mobile memory is made by dilating the original graph. The dilation factor indicates the amount the graph is expanded in each dimension.*

5.4 Sensitivity to the granularity of movement

Until this point we have examined a system in which only the requested word is moved. Figure 10 shows the effect of moving the requested word and at the same time moving some adjacent words with it. As can be seen in the figure, increasing the block size can either improve or degrade the performance of the system depending on the spatial locality exhibited in the program.

5.5 Sensitivity to Implementation Cost

Mobile memory has an implementation cost that we thus far have not addressed. To get a feel for the effect of introducing extra hardware and making room for directories, and the moved memory we examine how the system performs when we expand the graph. Figure 11 shows how the Centroid policy behaves as we expand the graph in each dimension. In other words, the worst case situation we see here is when we expand the graph by a factor of 9 in size, 3x in each direction. We see that mobile memory

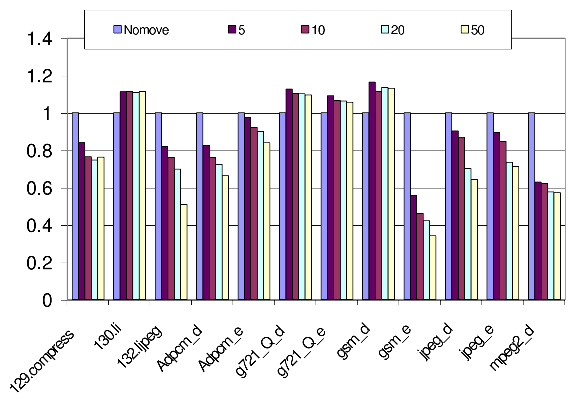

Figure 12: *The effect on memory access time with different sizes of local directory. Greedy policy is used.*

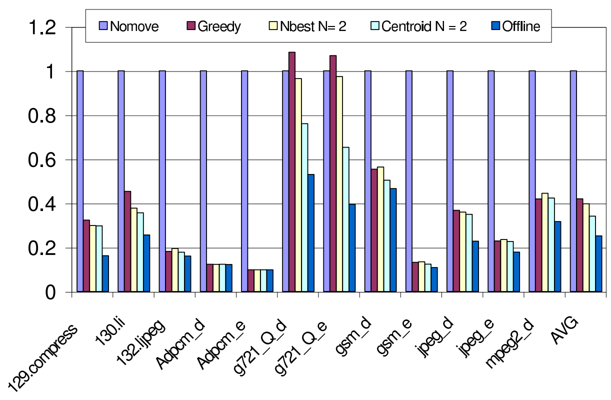

Figure 13: *Memory access cycles for different moving policies(N = history depth) (5 cycles / cluster)*

works well even when the amount of area is expanded by a factor of 4x. When it is expanded by a factor of 9x it outperforms the base case for five applications.

We also take a look at the effect of local directory sizes. Larger directories can increase the chance of finding the memory information in local directory, and reduces the latency caused by checking the home directory. Figure 12 shows how the greedy policy performs when we increased the size of local directory. Each bar is normalized by the baseline performance. Memory access cycles often go down when increasing the size of local directory. But some exceptions do happen, which can be explained away by the well-known "Belady anomalies" when using the First-In-First-Out algorithm to manage the directory. Note that most of the performance can be obtained with small directories.

5.6 Slowing down the clock

Figure 13 and Figure 14 shows the effect of reducing the signal propagation speed from 1 cycle per cluster to 5 cycles per cluster. The results indicate that mobile memory is even more important in this case as the critical ratio is higher, i.e., memory latency is more often on the critical path.

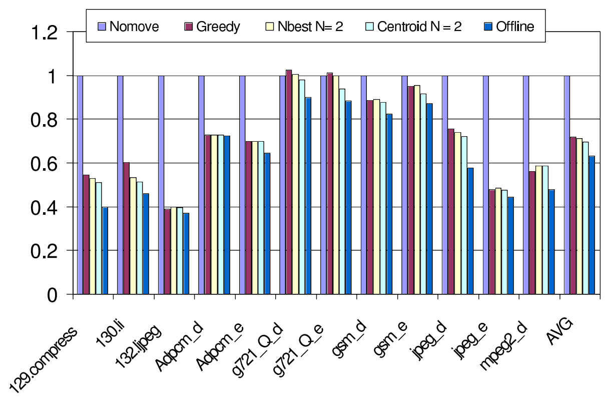

Figure 14: *Total execution cycles for different moving policies(N = history depth) (5 cycles / cluster)*

6 Conclusions

Very large reconfigurable fabrics are becoming a reality. With current offerings in the many millions of gates and chips expected in the near future with fifty million or more gates we will soon see entire applications mapped to reconfigurable devices. These devices will have the logic and memory necessary to implement mobile memory. In this paper we have shown that mobile memory has the potential to dramatically improve the performance of applications when they are mapped to very large reconfigurable fabrics. We determine that even a simple heuristic will always outperform a system without mobile memory and that the performance of these heuristics is quite close to that of the perfect off-line algorithm.

The limit study described in the this paper opens up a large area of research. We plan on developing a real implementation and more closely examining the costs of mobile memory. Furthermore, it appears that mobile memory in and of itself may not be sufficient for high performance. We plan on looking at how replication may be implemented along with mobile memory.

Finally, VLRFs have a significant advantage to multiprocessor systems in that the policy used at any one location can be tailored to the particular needs of that location. Thus, if the compiler can determine that a particular accessor, or memory location, needs to use mobility or replication, it can include that, and otherwise it can use a simpler system. In this way, the full power of reconfigurability can be brought to bear on the memory latency problem.

Acknowledgments

We thank Mihai Budiu for implementing the very large reconfigurable fabric simulation code. We would also like to thank other members of Phoenix group for their stimulating discussions and helpful hints. This work was supported in part by DARPA contract number N00014-01-1-0659.

References

[1] T. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 191–202, June 2001.

[2] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. TAM — a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, July 1993.

[3] Fredrik Dahlgren and Josep Torrellas. Ieee computer. *Journal of Parallel and Distributed Computing*, 32(6):72–79, June 1999.

[4] Seth Copen Goldstein and Mihai Budiu. NanoFabrics: Spatial Computing Using Molecular Electronics. In *Proceedings of the 28th International Symposium on Computer Architecture 2001*, 2001.

[5] E. Hagerstern, A. Landin, and S. Haridi. Ddm - a cache only memory architecture. *IEEE Computer*, 25(9):44–54, September 1992.

[6] Kendall Square Research Inc. Ksr technical summary. Technical report, Kendall Square Research Corporation, Waltham, MA, 1992.

[7] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Micro-30, 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, 1997.

[8] A. Srivastava and A. Eustace. a system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, 1994.

[9] Standard Performance Evaluation Corp. *SPEC CPU95 Benchmark Suite*, 1995.

[10] Simon C.Steely Susanne M.Balle. Analyzing memory access patterns of programs on alpha-based architectures. *Digital Technical Journal*, 9(4), 1997.

[11] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. OS support for improving data locality on CC-NUMA compute servers. Technical Report CSL-TR-96-688, Computer System Laboratory, Stanford University, 1996.

[12] Zhichen Xu, James R. Larus, and Barton P. Miller. Shared memory performance profiling. In *Principles Practice of Parallel Programming*, pages 240–251, 1997.