HLS Support for Unconstrained Memory Accesses

Girish Venkataramani, Tiberiu Chelcea and Seth Copen Goldstein
{girish,tibi,seth}@cs.cmu.edu
Computer Science Department
Carnegie Mellon University

Abstract

A major constraint in high-level synthesis (HLS) for large-scale ASIC systems is memory access patterns. Typically, most state-of-the-art HLS tools severely constrain the kinds of memory references allowed in the source, requiring them to have predictable access patterns or requiring dependencies between them to be statically determinable. This paper shows how these constraints can be eliminated.

We present an analysis infrastructure that can be used within any HLS toolflow for synthesizing circuits from high-level abstractions, such as ANSI-C, where no assumptions are made about either memory access latencies or about dependencies between memory references, i.e., arbitrary pointer aliasing are allowed. Our solution starts with a generic framework for building a dependence-aware, fully distributed, although often conservative, memory-access network (MAN) for a given memory-dependence graph. Then, we propose a suite of optimizations to customize the MAN for the given specification. All these techniques guarantee memory coherency. Experimental results on Mediabench benchmarks, show that such an approach succeeds in maintaining high levels of parallelism, while ensuring memory coherency. The optimizations succeed in lowering the synchronization overhead by as much as 4x.

1 Introduction

High level synthesis (HLS) tools have shown great potential for generating highquality circuits in a timely fashion by bridging the semantic gap between highlevel abstractions and gatelevel implementations. As design time becomes critical, and the increase in design productivity continues to lag behind the increase in design complexity [1], HLS tools can play a central role in delivering highperformance, lowpower, largescale ASICs from abstract, complex behavioral specifications [3].

In general, HLS tools have shown to be promising at extracting finegrained parallelism in synthesizing highperformance circuits for purely dataflow abstractions. However, largescale applications with numerous memory references continue to present an obstacle due to two main reasons:

- Highperformance memory systems result in variable memory access latency. Even the simplest hierarchical memory system will have different times for a cache hit and a cache miss. Hence, a synthesis flow cannot statically schedule these memory accesses.
- Even stateoftheart pointer alias analyses can statically disambiguate only about 60% of all memory dependencies in C programs [14]. Hence, the circuit must support some dynamic synchronization framework to guarantee memory coherency at all times.

Many HLS tools deal with the first issue by constraining the memory system so that all accesses take a fixed latency [32, 22]. For the second issue, most HLS tools restrict themselves to specifications in which all memory dependencies are statically explicit (for example, pointer aliasing is disallowed in SystemC [20]). The motivation of this paper is to introduce techniques to allow unconstrained memory access dependencies to occur in the source specifications for HLS tools, and further to support highperformance hierarchical memory systems. We present a framework that can be embedded within any HLS flow. This can help in expanding the applicationdomain space for tools like SystemC.

At the core of this framework is a simple memory access network (MAN) that provides reliable, pipelined, arbitrated access to shared memory resources. The key feature of the MAN is a synchronization framework that guarantees that no memory dependency (including statically unknown ones) is ever violated — in other words, memory coherency is always maintained. Of course, this guarantee involves a synchronization overhead at runtime, and several conservative design decisions may artificially increase the overhead. Thus, the paper also introduces optimization techniques that infer, from the source program, certain properties of the memory dependencies to reduce the synchronization overhead. These techniques were implemented within the CASH HLS toolflow [7, 29]. CASH starts with unannotated, unrestricted ANSIC programs and produces gatelevel implementations of fully asynchronous (selftimed) circuits.

The novel research contributions described in this paper are:

- A unified framework featuring a MAN architecture to synthesize memory coherent, gatelevel circuits from HLS specifications exhibiting variable latency memory operations, and statically unknown memory dependencies.
- Memory coherency requires synchronization checkpoints at various junctures in the circuit. We present MAN optimizations to eliminate unnecessary synchronization.
- With applications that exhibit high levels of memory parallelism, the MAN can quickly get congested. We present a flow-control technique to minimize MAN congestion.

The paper first describes the input specification requirements in Section 2, and goes on to illustrate how a simplistic MAN architecture can be built for this in Section 3. Various optimizations to enhance the MAN are presented in Section 4. We examine related work, present results and conclude in Sections 5,6 and 7 respectively.

```
int A_arr[100], B_arr[100];
int foo(int* ptr, int index, int offset) {
  int result = A_arr[index] +  // lod1
               B_arr[index];   // lod2
  if (index)
    result += *(ptr+offset);   // lod3
  else
    *ptr = result;             // str1
  return result;
}
```

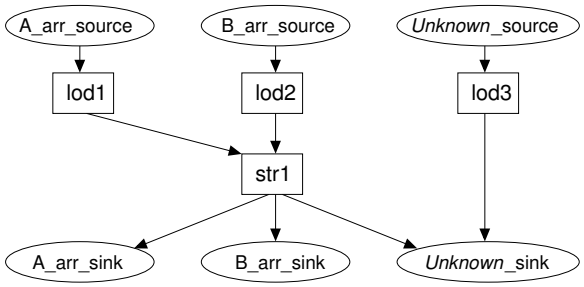


Figure 1: *A trivial example demonstrating explicit memory dependency representation*

2 Dependency Representation

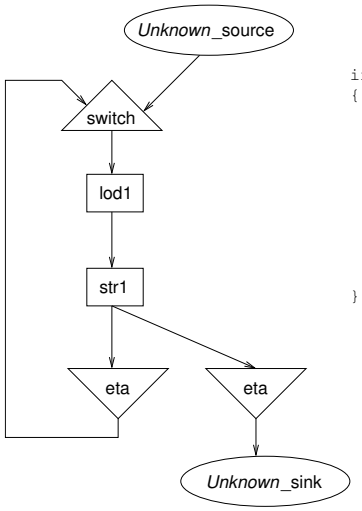
The starting point for building and optimizing our proposed memory access network is an input specification which explicitly encodes dependencies, called *may dependencies*, between memory references. A *may dependency* exists between any two references that cannot be proven to be independent.

We use the notion of location sets [33], which represents a unique position within a block of storage. Typically, each scalar memory location is assigned to its own location set, while all entries within an array are assigned to a single location set [33, 25]. Alias analysis then unravels false dependencies, and assigns memory accesses to unique location sets. In the worst-case, when nothing can be disambiguated, there will exist a single location set representing the entire memory block.

The input specification is a flow-graph in which nodes represent unique memory accesses in the source program and edges represent (synchronization-)*tokens*. The tokens indicate that two accesses are assigned to the same location set, i.e. there exists a may dependency between them. Thus, the flow-graph explicitly represents a partialordering of memory accesses through these tokens. At runtime, the execution semantics of a node is equivalent to dataflow execution semantics [2]. Each edge either holds a token or is *empty*. The memory access is executed only after it receives tokens along all its input edges. After accessing memory, the token is released along all its output edges. Thus, memory accesses are dynamically scheduled, and we claim that this is a necessary requirement in any synthesis framework supporting memory access dependencies that can only be dynamically disambiguated.

Figure 1 shows a trivial example of this representation. There are three location sets, arrays `A_arr` and `B_arr`, and the rest of the memory block, represented as $Unknown$. Special source and sink nodes in the graph represent the synchronization boundaries with the rest of the application. When execution starts, all sources contain tokens on their outputs. Execution finishes when all sinks have received tokens on their inputs.

The accesses, $lod1$ and $lod2$, reference different location sets (`A_arr` and `B_arr` respectively), and therefore the representation does not introduce any edge between them. On the other hand, we know nothing about the pointer, `ptr`, and hence it is associated with $Unknown$. To preserve memory coherency, we must assume that it can point to (or alias) either of the other two location sets. Thus, we may introduce tokens between each of $lod1$ and $lod2$, and each of $lod3$ and $str1$, and between $lod3$ and $str1$.




```
int foo(int* p, int* q, int n)
{
  int sum = 0, i;

  for (i = 0; i < n; i++) {
    int tmp = *p++;  // lod1
    *q++ = tmp;      // str1
    sum += tmp;
  }

  return sum;
}
```

Figure 2: *Memory-dependency representation in loops*

This is a valid, though conservative, memory-coherent specification. Further analysis reveals that $lod3$ need not synchronize with $lod1$ and $lod2$ since they are all memory-reads, and can be issued out-of-order. Similarly, we need not synchronize $lod3$ with $str1$ because they occur in the different branches of the `if-else` statement. But, we need to synchronize $lod1$ and $lod2$ with $str1$, as shown in Figure 1. Such analysis and elimination of redundant dependencies will improve the overall quality of the circuits, and should be performed before the techniques described in this paper are applied.

Finally, two constructs called SWITCH and ETA are used to model memory accesses in a loop, as shown in Figure 2. Inputs to a SWITCH are assumed to be mutually exclusive, and the execution semantics of this node just moves an active input to its output. ETA has a predicate and a data input. The execution semantics specifies that, when the predicate is `true`, the input is moved to the output; otherwise, no output is generated and the input is simply consumed.

Loop-entry points are represented using SWITCH nodes with two inputs - a token on the first input starts the execution of the first loop iteration, while a token on the second input implies the start of a new iteration. A loop-exit point is modeled using an ETA node. The node's data input is the token, and its predicate input represents the loop-exit condition.

This loop representation is illustrated in Figure 2. Notice that there exists a dependency between the accesses in one iteration to the same set of accesses in the next iteration, since the token has to flow around the loop. If it can be determined that the accesses in all loop iterations are independent, then code-motion (for loop-invariant accesses) or loop unrolling can be applied to break these false dependencies.

This completes our discussion of the input specification. This representation is not a complete one of the source application, since data as well as the predicate edges are missing. We assume that this representation will be used within a more complete and unified dataflow representation for the whole program, that mod-

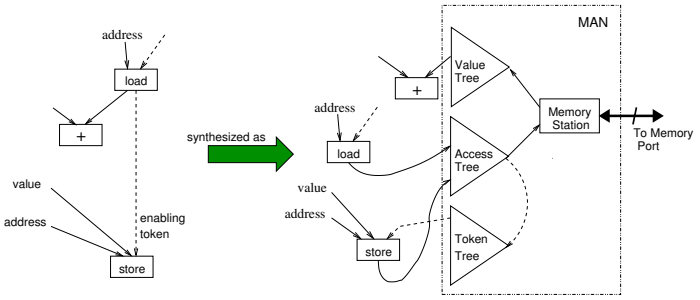


Figure 3: *(a) A simple input spec; and (b) the MAN architecture*

els all data-types. Pegasus [5] is one such complete representation, and several optimizations have been proposed to eliminate unnecessary token edges [6]. However, the token network specification is sufficient for the discussions in this paper.

3 MAN Architecture

This section describes how a given dependency-annotated input graph specification can be synthesized into a dependency-coherent memory access network (MAN). Without loss of generality, we assume that all accesses are to a single, shared memory block with a single memory port. The same optimization framework proposed in this paper can be applied to systems with multi-ported memories, or even multiple, distributed memories.

The MAN architecture must address three specific goals. It must (1) allow multiple, potentially concurrent accesses to the shared memory block; (2) allow data read from memory to be routed to their appropriate destinations; and (3) guarantee memory-coherency. The MAN allocates an input port for each static memory reference in the program. Hence, the MAN architecture must be highly scalable as the number of references can be numerous.

The proposed MAN architecture is fully distributed and pipelined. It is both scalable, since there are no global controllers, and provides for increased pipeline parallelism. Figure 3 shows the architecture of the MAN for a simple flow-graph, with only two memory operations with a memory dependency between them. The data read by the `load` from memory is to be used by an adder in the circuit.

The two accesses are connected to the $AccessTree$, which is a pipelined, arbitrated tree, whose tree elements are implemented as $Memmux$ elements, shown in Figure 4. Each $Memmux$ receives requests on $bus0$ and $bus1$ data bundles, which seek to gain access to the next level of the tree. The $Memmux$ arbitrates these using an asynchronous mutual-exclusion element [28]. The arbitration winner controls the Mux, which steers the appropriate data item to the internal register. In turn, the Mux activates the handshake controller (*HS Cntrl*), which performs the 4-phase handshaking between the tree children nodes and the parent node.

The $MemoryStation$ provides an interface to the shared memory port. It simply issues accesses once they exit the accesstree, and, for `load` accesses, it waits for the memory response and forwards the load-value to the $ValueTree$. This tree routes the values to the appropriate destinations (such as the adder in Figure 3). The valuetree, like the accesstree, is also pipelined, but the flow direction is reversed. Each tree-node element, called a $Memdemux$, is implemented as a handshake-demultiplexor, as shown in Figure 5.

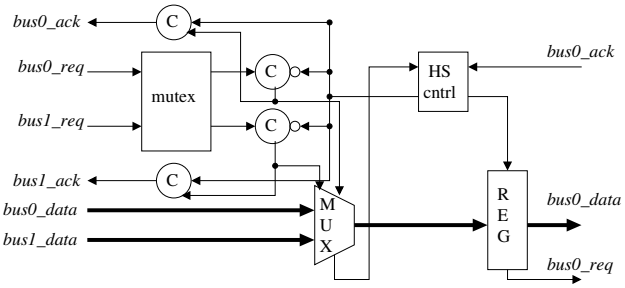


Figure 4: *Circuit-level implementation of the $Memmux$ element*

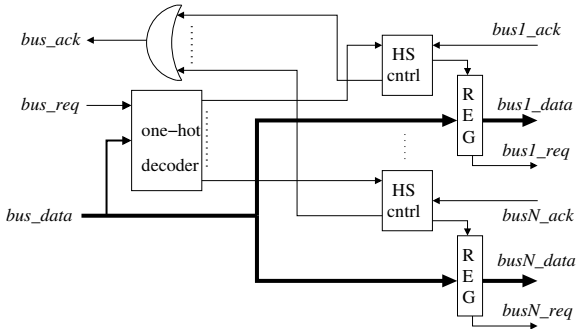


Figure 5: *Circuit-level implementation of the $Memdemux$ element*

Memory coherency can be achieved if an access releases its token to its dependents only when it is guaranteed that the dependents cannot overtake this access. Since the root of the access-tree represents a common point in the path of all accesses to memory and these accesses cannot overtake each other in the memorystation, the tokens can be safely released from the access-tree root. Once released, the token flows to its appropriate destination through the $TokenTree$. Functionally, the tokentree is identical to the valuetree.

The encoding of packets passing through these trees is straightforward. An accesstree packet has: (1) access-specific information like memory address, etc; (2) the path taken by its token through the tokentree; (3) value path, if the access is a `load`. The return tree packets carry path routing information, with the load-value being an additional field for the valuetree.

The return trees are parameterized by the tree-degree, and their construction is simple since their only function is to deliver packets to the appropriate destinations. The accesstree, on the other hand, not only delivers an access to the memorystation, but also releases the accesses' tokens to the tokentree. Decisions regarding when to perform the latter can significantly affect performance and is the topic of the next section.

4 Restructuring

A source of inefficiency in the MAN architecture is that memory coherency is enforced conservatively: an access releases its synchronization token only from the $AccessTree$ root, thereby synchronizing with every other access in the program. In reality, it needs to synchronize only with its token-dependents.

To understand this, consider a trivial example in Figure 6 where access A has only a single token-dependent, B. If they are both connected to the same $Memmux$ node, then this node becomes the common point on their paths, and A's token can be released

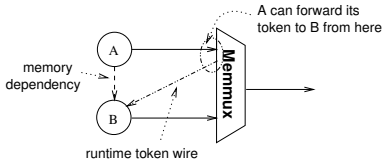


Figure 6: *By constructing subtrees of local dependence regions, early token release points can be found*

from here. In general, if there are N memory accesses in the program, then the token round-trip time (RTT) for A is reduced from $O(2\log(N)$ to just $O(1)$. Furthermore, A and B are mutually exclusive since B needs the token from A to fire; hence, this leaf-node can, in fact, be un-arbitrated, thereby reducing the critical path through $Memmux$.

This example shows that the RTT through the MAN can be greatly decreased if we take into account local regions of dependencies. We now present an algorithm to analyze memory dependence relations and to then construct an $AccessTree$ that reduces the token RTT, without sacrificing memory coherency.

4.1 SESE Regions

An important concept that enables our proposed optimization algorithm is Single Entry, Single Exit (SESE) regions [15]. Such a region has a single node through which all external edges enter the region, and has a single node through which all internal edges exit the region. Internally, however, there may be arbitrary fanout/fanin of edges, and can also include cycles. A SESE region has the property that for all its nodes except the entry node, the predecessors of each node are contained within this region. Therefore, if all the nodes in such a region belong to the same subtree (within the $AccessTree$), then the subtree root is a common point on their paths to memory. Except for the exit node, all accesses can release their tokens from this common point; the exit node, however, must synchronize with its dependents outside the SESE region.

For our analysis, we define two special types of SESEs: (1) a Linear SESE (LSESE) region is the longest chain of nodes with no fanout/fanin within them; (2) an Innermost SESE (ISESE) region is one that contains no other SESEs (including LSESEs) within it.

4.2 $Reduce$: AccessTree Construction Algorithm

Using SESEs, we have devised an algorithm, $Reduce$, that generates an application-aware $AccessTree$ for a given input graph, G. $Reduce$ is described in the pseudo-code presented in Figure 7 and is the focus of this section. $Reduce$ requires G to have unique $entry$ and $exit$ nodes. If a graph has multiple source (sink) nodes, then a pseudo-node $entry$ ($exit$) can be created as a predecessor (successor) of all source (sink) nodes. In addition, each node is associated with a tree pointer ($Tptr$) indicating which tree-node in $AccessTree$ releases the token for the access. Initially, all $Tptr$ pointers are $NULL$.

Next, we find all ISESE and LSESE regions in the graph, and create a local subtree for the nodes in the SESE. All accesses in an LSESE are mutually exclusive, and thus, the subtree for this is a simple un-arbitrated $Memmux$ with all LSESE nodes as inputs. For an ISESE, we first construct a balanced arbitrated tree, say t, using all the internal nodes of the region. Since the entry node,

```
Reduce(G) {
  Add pseudo entry/exit nodes, if necessary
  Initalize Tptrs of all nodes in G to 0.

  while (|G| > 1) {
    Find LS, the set of all LSESEs in G
    Find IS, the set of all ISESEs in G

    foreach s in LS {
      Let n = size of s
      Create un-arbitrated, n-input Memmux, r
      Connect all nodes in s to r
      Let Tptrs of all nodes (except exit) point to r
      Replace s in G with a new, unique node, x
    }
    foreach s in IS {
      Create binary, arbitrated Memmux tree, t, with all
       the internal nodes in s
      Create un-arbitrated, 3-input Memmux, r with inputs:
        entry(s), t, and exit(s) respectively
      Let Tptrs of all nodes (except exit) point to r
      Replace s in G with a new, unique node, x
    }
  }
}
```

Figure 7: *Pseudo-code of the $Reduce$ algorithm*

t, and the exit node are all mutually exclusive we connect these to a un-arbitrated 3-input $Memmux$, which forms the region's subtree root.

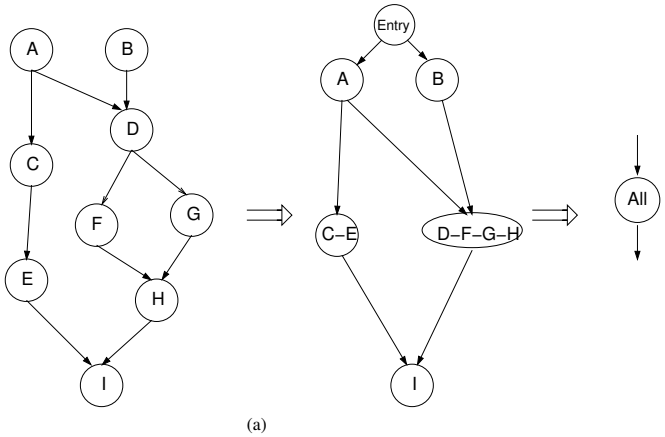
Once we form a subtree for a given SESE, say s, we update the $Tptr$ values of all nodes but the SESE exit to point to the subtree root. We then delete s from G, and replace it with a new (compound) graph node, say X, and we alias $Tptr(exit(s)) = Tptr(X)$. This SESE reduction is then repeated on the newly reduced graph. When X is later involved in a (higher-level) SESE reduction, we will be connecting the subtree associated with X to the (higher-level) tree, and $Tptr$ of X will be updated.

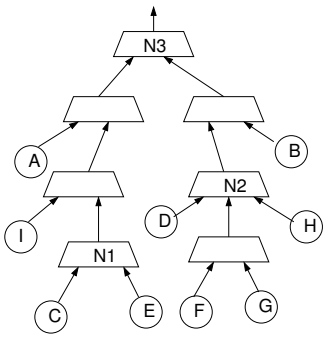
The progression of $Reduce$ is illustrated in Figure 8a. The initial graph is progressively reduced, until a single node is left in the graph. Since the original graph itself is one large SESE region, $Reduce$ will always converge. The final $AccessTree$ generated after reduce is shown in Figure 8b. The table in Figure 8c displays the values of $Tptr$ pointers corresponding to all accesses after the $Reduce$ completes. Notice that token RTT for most nodes have now been drastically reduced. The complexity of a single iteration of $Reduce$ is equivalent to the complexity of finding SESEs. The number iterations depends on the input graph structure, but typically converges in less than five iterations.

4.3 Loop Optimizations

Figure 9a shows a typical loop. In the first iteration, the loop accesses are dependent on (receive their tokens from) accesses that occur prior to entering the loop, the $Pred$ block. In subsequent iterations, all tokens remain within the loop itself. In the final iteration, accesses outside the loop (in the $Succ$ block) will be the recipients of last iteration's access tokens.

Loop accesses present a unique opportunity because we know that between any two given loop iterations, the memory dependencies are local to the loop region. However, a loop can never form a SESE region by itself since there always exists at least two loop exits — one for looping back to the loop-entry, and another to the $Succ$ block. A key observation, however, is that the latter exit is used only in the last loop iteration. To optimize for the common case, the loop is temporarily changed into a new graph,





Node	Tptr
A	N3
B	N3
C	N1
D	N2
E	N3
F	N2
G	N2
H	N3
I	N3

(c)

Figure 8: *An example of applying* $Reduce$ *on the graph in (a). (b) shows the resulting memory access tree; the table on the right shows* $Tptr$ *pointers for each access specifying where in the tree, this access can release its token*

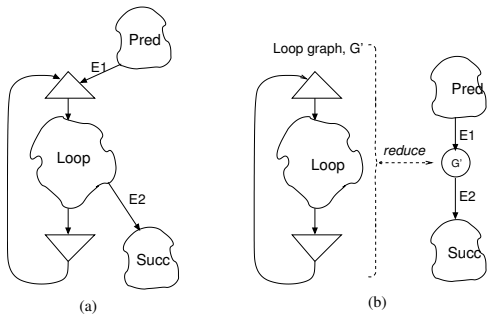


Figure 9: *Optimizing* $Reduce$ *in loops*

G' by deleting all exits to the $Succ$ block; thus, the only exit from the loop region is now the back-edge. Now, $Reduce$ will consider G' to be SESE region and optimize the resulting subtree independent of its external context.

Next, the fully reduced loop node is inserted into the top-level graph by re-introducing the deleted exit edges as shown in Figure 9b. The resulting graph is then reduced as before. However, upon this second reduction, the loop accesses that emit tokens along the $Succ$ block exits, are now flagged as generating tokens from two points in the $AccessTree$ — one from the reduction of G' and the other from the reduction of the top-level graph. The first (*local*) token synchronizes with all accesses within the loop, while the second (*global*) token is used only in the last iteration and is used to synchronize with dependents in the $Succ$ block. In this manner, loop accesses are accelerated through localized synchronization.

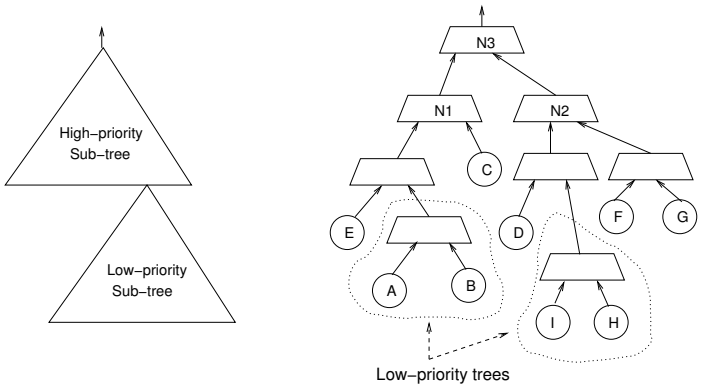


Figure 10: *For two groups of high- and low-priority nodes, the tree construction can be biased as shown in (a). (b) shows the result of applying this optimization on the access tree generated in Figure 8 to optimize nodes C, F and G.*

4.4 Priority-based Construction

A problem with $Reduce$ is that accesses in SESE regions reduced earlier on end up at the bottom of the $AccessTree$ (for example, *C-E* and *D-F-G-H* in Figure 8). This is not a problem for token RTT, however, if any of these accesses is a `load`, then the path of the load-value round-trip will include the entire depth of the $AccessTree$. If the access is critical, then overall performance can degrade.

We solve this problem by assigning priorities to nodes during the subtree construction phase of $Reduce$. Currently, the algorithm employs a simple, greedy heuristic that only considers two levels of priorities at a time — $High$ and Low. Independent subtrees are formed for each priority class, and finally, the Low priority subtree is subordinated to a leaf of the $High$ priority subtree. Figure 10a illustrates idea.

Since a node involved in the subtree construction could itself be a smaller subtree, the heuristic can be applied hierarchically, which provides for more flexibility in selecting the accesses to be optimized. For example, in Figure 8b, if we assign high-priority to accesses *C, E, D, F, and G*, then the resulting $AccessTree$ is shown in Figure 10b. Most importantly, this heuristic preserves the $Tptr$ of the smaller subtrees involved in the construction.

4.5 Sequencers

In case of high memory parallelism, the congestion in the MAN usually results in performance penalties due to two reasons - (a) high cost of arbitration, and (b) the turn-around time of the handshake in $Memmux$, which is usually hidden, now falls on the critical path. Therefore, as an optimization, entire arbitrated subtrees are replaced by "sequencer" stages, which, unlike a *Memmux* stage, services its inputs in strict order and does not need arbitration. In addition, while it is expensive to implement arbitrated *Memmux*es with more than two incoming accesses, a sequencer stage can easily be extended to multiple inputs. With this optimization, the critical path for the accesses serviced by the sequencer is reduced from $\ln(N)$ down to 1 stage.

However, not any sub-tree can be replaced by a sequencer, since (a) deadlocks may be introduced, and (b) performance may degrade when a critical access is connected to a port serviced

amongst the last. Currently, our MAN optimization step uses sequencers only for a set of accesses that have (a) common sources, which will avoid deadlocks, and (b) have the same ASAP schedule, ensuring that performance is not sacrificed.

5 Related Work

HLS tool support for applications with memory references can be classified into the following four broad categories:

- Memory Size Estimation and Mapping: A vast body of work examines the ideal sizing of memory modules in order to customize them to the particular application's needs [30, 23, 34, 17]. De Micheli [25] described how data structures in ANSI-C can be allocated into separate memories. In particular, they present an implementation of the `malloc/free` constructs in C used for dynamic memory allocation.
- Memory redundancy elimination: has been the focus of many efforts attempting to improve memory bandwidth and reduce unnecessary memory accesses. Kolson [18] described techniques based on Tree Height Reduction to consider memory access latencies and redundancies in forming a schedule. Recent work by Stitt [27] shows how words recently read from memory can be reused.
- Access ordering and access scheduling: A huge body of work in HLS systems addresses the problem of static scheduling in memory-intensive applications [9, 32, 13, 24, 22]. Most of these efforts use a control-data flow graph (CDFG) like specification as their input, where memory references are explicitly marked (that is, statically disambiguated). They differ in the static scheduling algorithm used, and may even assume that memory accesses incur fixed latencies [32, 22]. There are also some efforts that consider memory access scheduling and memory allocation in conjunction [21, 26].
- Tool support: A number of C-like toolflows [8, 10, 11, 31, 16, 12] like System-C [20] define synthesizable subsets of C, but they all require static memory reference disambiguation.

The first two research directions described above are orthogonal to the focus of our work, and can be used in conjunction with our techniques. A commonality in the last two is that they all perform static scheduling and rely on the fact that memory references can be statically disambiguated.

Our work differs from all of the above in that our proposed HLS techniques support input specifications in which memory references cannot be statically determined. Our input spec uses an explicit memory dependency representation, which also becomes a runtime synchronization construct. Hence, all memory accesses are *dynamically scheduled* once their dependencies have been dynamically disambiguated. To our knowledge, we are the first to propose HLS techniques to handle these concepts.

6 Experimental Results

This section evaluates the quality of the MAN architecture, and the optimizations applied to it. We have implemented the techniques described in this paper within the CASH toolflow [29]. The input to this toolflow is ANSI-C programs, which are automatically synthesized into fully asynchronous, gate-level circuits using a [180nm/2V] standard-cell library. All results reported in this section are extracted from post-layout estimations.

Benchmark	Kernel	Static Refs.	Tok RTT Ratio	Val RTT Ratio
adpcm_d	adpcm_decoder	9	623.00	1.12
adpcm_e	adpcm_coder	10	579.00	1.30
gsm_d	Short_term_synthesis_filtering	6	1.93	1.34
gsm_e	Short_term_analysis_filtering	5	1.94	1.05
jpeg_d	jpeg_idct_islow	68	1.23	1.14
jpeg_e	jpeg_fdct_islow	32	2.43	1.00
mpeg2_d	idctcol	35	0.78	1.00
mpeg2_e	dist1	43	1.21	1.03
pgp_d	mp_smul	7	1.22	1.00
pgp_e	mp_smul	7	1.23	1.00
Geometric Mean (GM)			4.74	1.09

Table 1: *List of the benchmark kernels synthesized and the number of static memory accesses within each kernel. The RTT ratios specify the ratio of the weighted average of dynamic RTTs in the base MAN to the weighted average of dynamic RTTs in MAN built using* $Reduce$.

Our benchmarks are the most frequently executed kernels from the Mediabench suite [19]. The second column of Table 1 lists these kernels and the third column lists the number of static memory references present in them. The latter is the number of input ports to the accesstree, and gives us a feel for its complexity. For all these kernels, our technique achieves its design goal of synthesizing a memory coherent access network which allows for high memory level parallelism (MLP), which is measured as the number of in-flight accesses. In other words, MLP measures the number of accesses that have entered the accesstree but whose tokens have not yet emerged from the tokentree. Our results show that the MAN can sustain high MLP, e.g., 32 in the `mpeg2_e` kernel.

MAN Optimizations. Table 1 shows the impact of using the restructured MAN as described in Section 4, by evaluating the effect on the token and value RTTs. Token (value) RTT is defined as the time difference between when an access is inserted into the accesstree, and when its token (load-value) emerges from the tokentree (valuetree). Using simulation traces, we computed the weighted average of the dynamic token and value RTTs over all accesses. The fourth and fifth column of Table 1 shows the ratio of these averages for the basic MAN versus a restructured MAN. A value greater than one implies the token (value) RTT is better (i.e., shorter) after restructuring.

The table suggests that we are able to reduce the the token RTT (and hence the synchronization overhead) by a factor greater than 4x, on average. This is especially true with the `adpcm` kernels, which contain loops (and hence SESE regions) with a single access. This means that the sub-tree consists of a single node, and the synchronization token for the access never needs to enter the MAN. Hence token RTT is zero. The improvements for the value RTT comes from assigning high-priority to loop accesses as discussed in Section 4.4. However, these improvements are modest,

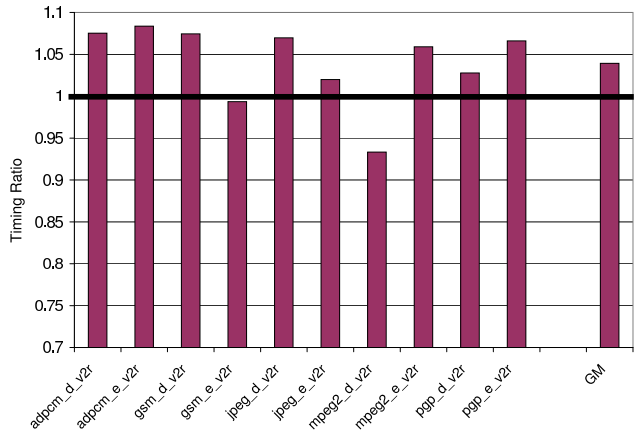


Figure 11: *Evaluation of MAN optimizations*

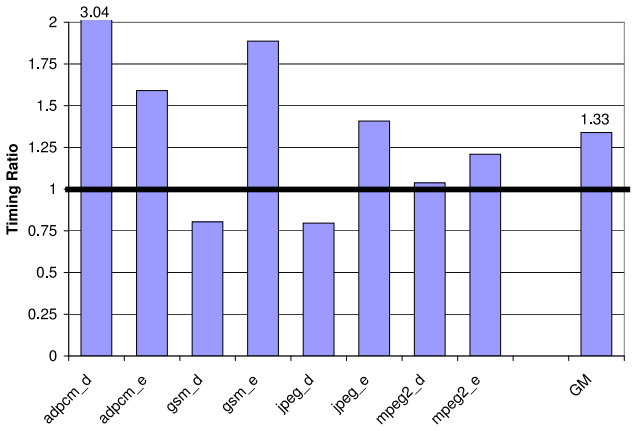


Figure 12: *Performance comparison with superscalar*

because MAN restructuring mainly strives to reduce synchronization overhead, but may inadvertently worsen the value RTT.

Figure 11 shows the impact on the kernel performance when using the restructured MAN. The graph shows the ratio of the kernel execution latency of the base MAN construction versus the optimized one. In most cases (8 out of 10), performance improves with the optimized MAN.

For two benchmarks, the base construction works better. The `gsm_e` kernel contains a two-deep nested loop, with two accesses in the outer loop, and the rest of the accesses within the inner loop. The greedy heuristic used in the priority-based construction works poorly since, in our current framework, it only prioritizes inner loop accesses; our simulations indicate that the outer loop accesses are on the critical path, which is now worsened.

The `mpeg2_d` kernel loses by about 8% and is a more instructive example. Here, there are no loops in the kernel. Since our priority heuristic only classifies loop accesses as high-priority, the *Reduce*-based construction ends up being priority-less. This shows that a more nuanced priority-based *Reduce* construction is vital. Without this information, *Reduce* may result in overall performance degradation.

The sequencer synthesis technique, on the other hand, is a local optimization, and is bound to improve the critical path when it can be applied. It boosts the performance of the `mpeg2_e` and `jpeg_d` kernels, which are the only ones that present an opportunities. In the former, two sets of 16 accesses can use sequencers, and in the latter two sets of 8 accesses can be sequenced, resulting in a reduction of 3/2 stages, respectively, for each sequenced access.

Based on these results, we can draw two conclusions:

- The *Reduce*-based accesstree construction technique relies heavily on which accesses are tagged for optimization. The current greedy heuristic is insufficient in some cases, and employing more sophisticated analysis to reveal critical accesses would greatly enhance *Reduce*. Finding critical accesses is really orthogonal to the *Reduce* algorithm, and we are currently working on incorporating some scheduling-based analysis techniques.
- Table 1 shows that *Reduce* results in significant reductions in the dynamic RTTs; however, this does not always translate to equivalent improvements in overall performance in Figure 11. Analysis of the results shows that these reductions in the RTTs have the effect of shifting the critical sections to new regions in the circuit, which then become the bottleneck to performance. To that end, the optimizations have succeeded. However, to attain overall improvement, it is imperative that we use the MAN construction techniques in concert with other circuit optimization techniques, that can then deliver a global improvement in performance.

MAN vs Superscalar. The optimizations presented in this paper improve on the basic performance of the MAN. To provide the reader with a feeling for the overall effectiveness of the resulting circuits we compare the performance of the automatically synthesized circuit to that of a superscalar processor running the identical program. Despite the fact that the superscalar is general purpose, the comparison is illustrative of two very different approaches to accessing memory. In the superscalar, there is a single-point of access for all memory operations, the load-store queue (LSQ). Thus, there is no arbitration. Furthermore, the LSQ performs a host of aggressive optimizations such as store forwarding, and load-store re-ordering to minimize the cost of memory access dependencies. On the other hand, the ASIC implementations have many points of access to memory, and one of the fundamental functions of the MAN is to arbitrate amongst these accesses, while enforcing all memory dependencies. With a single-ported memory in both cases, the cost of performing a memory access in the MAN-based ASIC is thus much more expensive than in a superscalar core. In many ways, the LSQ and the MAN are two extremes in the continuum of hardware support for unconstrained memory accesses.

Figure 12 compares the execution latency of the C kernels in Wattch [4] simulations of a 4-wide, out-of-order execution core versus the execution latency of the same kernels in post-layout simulations of ASICs generated by our toolflow with all MAN optimizations enabled. The factors influencing performance in both systems are varied; CASH takes advantage of data-parallelism through spatial layout of the circuit, while the superscalar core relies on techniques like register renaming, prediction and speculation to uncover parallelism. Previous limit studies of the MAN [7] show that it quickly becomes a bottleneck in memory-intensive applications due to its excessive cost of accessing memory. Despite this, our results show that an ASIC with an optimized MAN still delivers competitive performance (on average 33% better than the core). Of course, the superscalar core is more flexible since the same core can execute all kernels.

7 Conclusions

This paper presents techniques to support unconstrained memory accesses in a HLS flow. Given an input graph representation that explicitly specifies may-dependencies, we can synthesize a distributed memory access network (MAN) architecture to provide access to and from memory. The architecture is scalable and maintains memory coherency at all times. Comparison with aggressive memory coherency techniques used in superscalar cores show that MAN is effective since it keeps the cost overhead of distributed memory access low.

We have also presented some analyses and optimizations that improve upon the basic MAN architecture by reducing excessive synchronization and by accelerating the application for the common-case. While some optimizations like sequencer-synthesis are local optimizations and should always provide benefits, others like $Reduce$-based $AccessTree$ construction relies heavily on good criticality information. Although these optimizations generally improve performance by themselves, our analysis shows that there is more to gain if they are applied in concert with other circuit optimizations.

8 Acknowledgments

We want to thank the reviewers for their helpful and thorough comments. This research is funded in part by the National Science Foundation under Grants No. CCR-0224022 and No. CCR-0205523, by DARPA under contracts N000140110659 and 01PR07586-00, the Semiconductor Research Corporation and an equipment grant from Intel Corporation.

References

[1] International technology roadmap for semiconductors (ITRS). http://public.itrs.net/Files/2003ITRS/SysDrivers200.pdf, 2003.

[2] T. Agerwala and Arvind. DataFlow systems. *Computer*, 15(2):10–30, February 1982.

[3] Arvind, R. Nikhil, et al. High-level synthesis: An essential ingredient for designing complex asics. In *ICCAD*. IEEE Press, 2005.

[4] D. Brooks, V. Tiwari, et al. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *ISCA*. ACM SIGARCH / IEEE, June 2000.

[5] M. Budiu and S. C. Goldstein. Pegasus: An efficient intermediate representation. Technical Report CMU-CS-02-107, Carnegie Mellon University, May 2002.

[6] M. Budiu and S. C. Goldstein. Optimizing memory accesses for spatial computation. In *International ACM/IEEE Symposium on Code Generation and Optimization (CGO)*, pages 216–227, March 23–26 2003.

[7] M. Budiu, G. Venkataramani, et al. Spatial computation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 14–26, October 2004.

[8] C Level Design, http://www.cleveldesign.com/. *C2HDL*.

[9] G. Corre, E. Senn, et al. Memory accesses management during high level synthesis. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 42–47. ACM Press, 2004.

[10] CoWare, http://www.coware.com/. *N2C*.

[11] Frontier Design, http://www.frontierd.com/. *A—rt Builder*.

[12] A. Ghosh, J. Kunkel, et al. Hardware synthesis from c/c++. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, page 82. ACM Press, 1999.

[13] P. Gupta and A. C. Parker. Smash: a program for scheduling memory-intensive application-specific hardware. In *ISSS '94: Proceedings of the 7th international symposium on High-level synthesis*, pages 54–59. IEEE Computer Society Press, 1994.

[14] M. Hind and A. Pioli. Evaluating the effectiveness of pointer alias analyses. *Science of Computer Programming*, 39(1):31–55, January 2001.

[15] R. Johnson, D. Pearson, et al. The program structure tree: Computing control regions in linear time. In *SIGPLAN Conference on PLDI*, pages 171–185, June 1994.

[16] T. Kambe, A. Yamada, et al. A c-based synthesis system, bach, and its application (invited talk). In *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 151–155. ACM Press, 2001.

[17] P. G. Kjeldsberg, F. Catthoor, et al. Storage requirement estimation for optimized design of data intensive applications. *ACM Trans. Des. Autom. Electron. Syst.*, 9(2):133–158, 2004.

[18] D. J. Kolson, A. Nicolau, et al. Integrating program transformations in the memory-based synthesis of image and video algorithms. In *ICCAD '94: Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 27–30. IEEE Computer Society Press, 1994.

[19] C. Lee, M. Potkonjak, et al. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Micro-30*, pages 330–335, 1997.

[20] S. Y. Liao. Towards a new standard for system level design. In *CODES*, pages 2–6. ACM Press, 2000.

[21] C.-G. Lyuh and T. Kim. Memory access scheduling and binding considering energy minimization in multi-bank memory systems. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 81–86. ACM Press, 2004.

[22] G. Mittal, D. C. Zaretsky, et al. Automatic translation of software binaries onto fpgas. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 389–394. ACM Press, 2004.

[23] S. Y. Ohm, F. J. Kurdahi, et al. A comprehensive estimation technique for high-level synthesis. In *ISSS '95: Proceedings of the 8th international symposium on System synthesis*, pages 122–127. ACM Press, 1995.

[24] J. Park and P. C. Diniz. Synthesis of pipelined memory access controllers for streamed data applications on fpga-based computing engines. In *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, pages 221–226. ACM Press, 2001.

[25] L. Séméria, K. Sato, et al. Synthesis of hardware models in C with pointers and complex data structures. *IEEE trans. on VLSI*, 2001.

[26] J. Seo, T. Kim, et al. An integrated algorithm for memory allocation and assignment in high-level synthesis. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 608–611. ACM Press, 2002.

[27] G. Stitt, Z. Guo, et al. Techniques for synthesizing binaries to an advanced register/memory structure. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 118–124. ACM Press, 2005.

[28] I. Sutherland. Micropipelines: Turing Award Lecture. *Comm. of the ACM*, 32 (6):720–738, June 1989.

[29] G. Venkataramani, M. Budiu, et al. C to asynchronous dataflow circuits: An end-to-end toolflow. In *International Workshop on Logic synthesis (IWLS)*, pages 501–508, June 2004. (full paper).

[30] I. M. Verbauwhede, C. J. Scheers, et al. Memory estimation for high level synthesis. In *DAC '94: Proceedings of the 31st annual conference on Design automation*, pages 143–148. ACM Press, 1994.

[31] K. Wakabayashi. C-based synthesis experiences with a behavior synthesizer, cyber. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, page 83. ACM Press, 1999.

[32] W. Wang, T. K. Tan, et al. A comprehensive high-level synthesis system for control-flow intensive behaviors. In *GLSVLSI '03: Proceedings of the 13th ACM Great Lakes symposium on VLSI*, pages 11–14. ACM Press, 2003.

[33] R. P. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, 1995.

[34] Y. Zhao and S. Malik. Exact memory size estimation for array computations without loop unrolling. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 811–816. ACM Press, 1999.