SOMA: A Tool for Synthesizing and Optimizing Memory Accesses in ASICs

Girish Venkataramani, Tiberiu Chelcea
Seth Copen Goldstein
Carnegie Mellon University
Pittsburgh, USA
{girish,tibi,seth}@cs.cmu.edu

Tobias Bjerregaard
TU Denmark
Lyngby, Denmark
tob@imm.dtu.dk

ABSTRACT

Arbitrary memory dependencies and variable latency memory systems are major obstacles to the synthesis of large-scale ASIC systems in high-level synthesis. This paper presents SOMA, a synthesis framework for constructing Memory Access Network (MAN) architectures that inherently enforce memory consistency in the presence of dynamic memory access dependencies. A fundamental bottleneck in any such network is arbitrating between concurrent accesses to a shared memory resource. To alleviate this bottleneck, SOMA uses an application-specific concurrency analysis technique to predict the dynamic memory parallelism profile of the application. This is then used to customize the MAN architecture. Depending on the parallelism profile, the MAN may be optimized for latency, throughput or both. The optimized MAN is automatically synthesized into gate-level structural Verilog using a flexible library of network building blocks. SOMA has been successfully integrated into an automated C-to-hardware synthesis flow, which generates standard cell circuits from unrestricted ANSI-C programs. Post-layout experiments demonstrate that application specific MAN construction significantly improves power and performance.

Categories and Subject Descriptors: B.5.2 [RTL Implementation]: Automatic Synthesis from ANSI-C; B.4.3 [I/O and Data Communications]: Topology
General Terms: Design, Performance, Experimentation.
Keywords: High-level synthesis, memory synthesis.

1. INTRODUCTION

When all data dependencies are statically explicit in the source specification, today's HLS tools have shown great promise in synthesizing high-performance circuits by extracting parallelism. However, large-scale applications with numerous memory references continue to present an obstacle in HLS for two main reasons: (1) In a hierarchical memory system, accesses may have variable latency, e.g. in the event of a cache miss. Hence, these cannot be statically scheduled. (2) More importantly, [10] finds that, on average, even the state-of-the-art pointer alias analysis can statically disambiguate only about 60% of all memory dependencies in C programs. Hence, the circuit must support a dynamic synchronization mechanism to guarantee memory consistency at all times. Many HLS tools deal with the first issue by imposing a fixed, worst-case (and often unacceptable) access time for memory operations [28, 18]. For the second issue, they restrict the input specs to statically explicit memory dependencies (for example, pointer aliasing is disallowed in System-C [16]).

The motivation behind this paper is to introduce techniques to allow unconstrained memory access dependencies in the source specification. To this end, we introduce SOMA, a framework for Synthesizing and Optimizing Memory Accesses. SOMA can be embedded within any HLS flow (like System-C, for example), thus expanding the capabilities of the tool. SOMA synthesizes a memory access network (MAN) that provides for pipelined, arbitrated access to shared memory resources, and for data items from memory to be reliably routed to their appropriate destinations. For dynamic memory dependency support, the MAN implements a synchronization mechanism that dynamically resolves dependencies, while sustaining high levels of memory parallelism.

The input to SOMA is a specification in which all potential memory dependencies are explicit (Section 3). A vital challenge in architecting the MAN (described in Section 4) is the construction of an optimal access network that routes potentially concurrent memory requests from numerous initiation points to a shared memory. We present a formal analysis framework to characterize the performance of different access network topologies for a given memory access concurrency profile. Using this information, we describe a heuristic to predict the application's dynamic memory parallelism profile and to finally construct an optimized MAN topology (Section 5). This topology is then automatically synthesized using our flexible library of network building blocks. As a demonstration of its use, SOMA has been integrated into CASH [25], an automated HLS toolflow that synthesizes unrestricted ANSI-C programs into pipelined, clockless circuits. However, the techniques described here can be easily adapted to other (e.g., synchronous) implementations as well. Post-layout experiments reveal the usefulness of our concurrency analysis, and the robustness of the heuristic in improving the MAN performance (Section 6).

2. RELATED WORK

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

1. 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 [26, 19, 30, 12]. Séméria [21] 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.

2. Memory redundancy elimination: Kolson [13] uses Tree Height Reduction to consider memory access latencies and redundancies in forming a schedule. Recent work by Stitt [23] shows

CODES+ISSS'05, Sept. 19–21, 2005, Jersey City, New Jersey, USA.

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

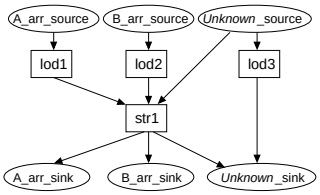


Figure 1: *An example demonstrating explicit memory dependency representation*

how words recently read from memory can be reused.

3. Access ordering and access scheduling: A huge body of work in HLS systems addresses the problem of static scheduling in memory-intensive applications [5, 28, 9, 20, 18]. Most of these efforts start with a control-data flow graph like specification, where memory references are explicitly marked (i.e. statically disambiguated). They differ in the static scheduling algorithm used, and may even assume that memory accesses incur fixed latencies [28, 18]. There are also some efforts that consider both memory access scheduling and memory allocation together [17, 22].

4. Tool support: A number of C-like toolflows [4, 6, 7, 27, 11, 8] define synthesizable subsets of C, but they all require static memory reference disambiguity. This limitation is also present in flows that start from System-C [16].

The first two categories 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 statically disambiguated memory references. Static scheduling of dynamically dependent memory operations results in overly conservative schedules, and hence none of the above techniques address this problem.

Our work differs from all of the above in that the proposed HLS techniques support input specifications in which memory references cannot be statically disambiguated. Our input specification explicitly represents potential memory dependencies, 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.

3. DEPENDENCY REPRESENTATION

The input specification to SOMA explicitly encodes *may dependencies* between memory references. A may dependency exists between any two references that cannot be proven to be independent. Alias analysis is used to eliminate false dependencies, and assigns memory accesses to unique location sets [29]. In the worst-case, when nothing can be statically 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*. A token between two accesses indicates that both *may be* assigned to the same location set at runtime. Thus, the flow-graph explicitly represents a partial-ordering of accesses through tokens. At runtime, the execution semantics of a memory access node is equivalent to dataflow execution semantics. A memory access is initiated only after it receives tokens along all its input edges. After accessing memory, tokens are released to all its successors. Thus, memory accesses are, in essence, dynamically scheduled, which is a necessary requirement in any synthesis framework supporting memory access dependencies that can only be dynamically disambiguated.

Figure 1 shows a simple example of this representation. In the C code, there are essentially three location sets, arrays `A_arr` and `B_arr`, and the rest of the memory block, represented as `Unknown`. Special source and sink nodes represent the synchronization boundaries with the rest of the application. The accesses, *lod1* and *lod2*,

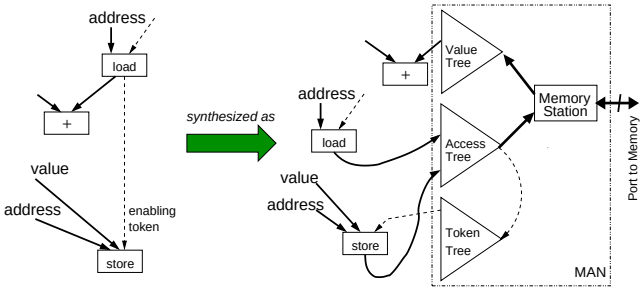


Figure 2: *An abstraction of the MAN architecture*

reference different location sets (`A_arr` and `B_arr` respectively), and therefore the representation does not introduce any edge between them. Nothing is known about `ptr`, and hence it is associated with *Unknown*. To preserve memory coherency, we must assume that it can point to either of the other two location sets. Closer examination reveals that *lod3* need not synchronize with *lod1* or *lod2* since they are all memory-reads. Similarly, we need not synchronize *lod3* with *str1* because they occur in different branches of the `if-else` statement. But, we need to synchronize *lod1* and *lod2* with *str1*. Elimination of redundant dependencies in this representation is important and has been addressed in [2], but is orthogonal to the focus of this paper.

4. MAN ARCHITECTURE

This section describes how a dependency-annotated token graph can be synthesized into a dependency-coherent memory access network (MAN). For ease of explanation, we assume that all accesses are to a shared, central, single-ported memory. The framework proposed in this paper can be extended to cover systems with multi-ported memories, or multiple, distributed memories.

The MAN architecture addresses three specific goals. It must (1) support multiple, potentially concurrent accesses to the shared memory; (2) route data read from memory to its appropriate destinations; and (3) guarantee memory consistency. The MAN implements an input port for each static memory reference in the program. Hence, it must be scalable as the number of references can be numerous. Figure 2 illustrates the architecture of the proposed MAN. It consists of three trees: an *access tree* routes requests to memory, arbitrating amongst concurrent accesses to provide sequential access to the *memory station*, which is the interface to memory; the *value tree* routes the memory responses (i.e. load-values) to their appropriate destinations, and the *token tree* routes synchronization-tokens, used for memory consistency. Dynamic synchronization is achieved by releasing the token associated with an access to its consumers, from the root of the access tree, since access-order from this point to memory can be guaranteed.

4.1 MAN building blocks

The MAN is characterized by dynamic scheduling, which implies that all communication within it and with the outside world is asynchronous in nature. At the circuit level, however, the MAN can be realized as a synchronous implementation, as a GALS architecture, or, as we have done in this work, as a clockless or self-timed implementation. Clockless circuits are characterized by the absence of a global synchronization signal. They are data driven, and all flow control is handled by local handshaking mechanisms. We make use of the 4-phase bundled-data protocol that uses control signals, request (req) and acknowledge (ack) [24], to implement the communication handshake.

In a single-ported monolithic memory, the construction of the value and token trees is simple since no congestion can occur. Each tree-node is implemented as a handshake-demultiplexer element, and the route to the destination is encoded in the data itself. The

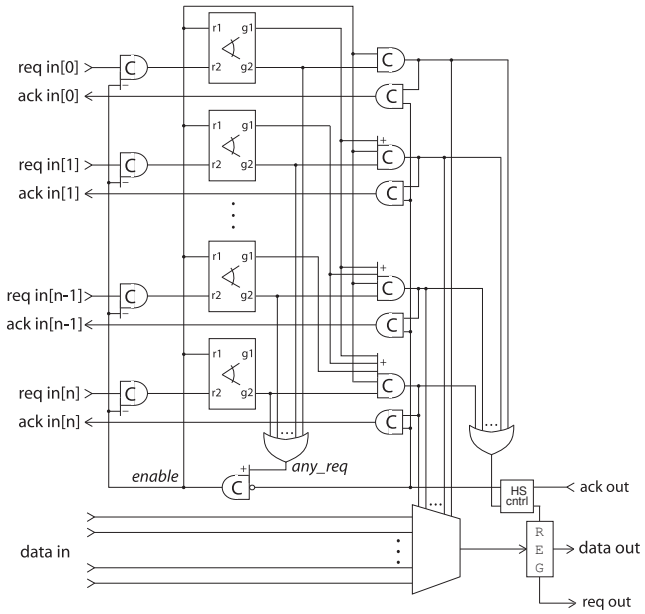


Figure 3: *Handshake-multiplexer implements an access tree node.*

access tree, on the other hand, is more complex since it must support arbitration between potentially concurrent accesses. Congestion among concurrent accesses degrades performance considerably, but the performance cost overhead can be minimized by customizing the tree topology to the application.

An access tree node is an arbitrating handshake-multiplexer as shown in Figure 3. The register at the output allows for pipeline-parallelism in the tree. Since the timing between the arrival of concurrent inputs is unknown in a clockless implementation, special circuitry is needed to arbitrate between these. The key control signal for this is *enable*. When input requests arrive, *any_req* causes *enable* to go high; this locks all signals to the right of the mutual exclusion elements [24] and starts the arbitration phase. The inputs are arbitrated according to a static priority. This design is based on [3], but is optimized for throughput, and first used in [1].

The access tree node is characterized by its *forward latency* and *cycle time*. The forward latency is the latency through the node when there is no congestion, while the cycle time is the minimum time between servicing two concurrent inputs. These parameters are functions of the number of inputs and the bit width. We can characterize the worst-case values of the forward latency (F_j) and cycle time (C_j) of a j-input multiplexer, for a given bit width, as:

$$F_j = \lambda + \beta \log_2 j \quad (1)$$

$$C_j = \tau + \gamma \log_2 j \quad (2)$$

The constants, λ and τ scale logarithmically with the bit width of the data path. The j-dependent term arises from the need to drive and merge internal signals that scale with the number of inputs. We have evaluated the constants (in terms of logic levels) to be $\lambda = 15 + \log_4 B$, $\tau = 22 + \log_4 B$, and $\beta = \gamma = 4.5$, where B is the bit width.

Note that (1) and (2) are analytical approximations based on the circuit architecture. In a real implementation the parameters do not follow such a smooth curve; e.g., when going from 8 to 9 inputs, the parameters may jump as an extra logic level is added in internal buffering. Nevertheless, functions (1) and (2) are of great use in understanding our heuristic, as will become clear in the following section.

5. TREE CONSTRUCTION

In this section, we first describe the cost functions used to evaluate the performance of a given access tree topology, and then use these results in a heuristic that constructs an application-specific tree. We limit the scope to balanced trees, as they are easier to formally reason about. Later, we will discuss the impact of creating asymmetric trees as well.

A balanced tree can be characterized by $[k, \{b_1, ..., b_k\}]$, where k is the number of levels in the tree, and b_i is the number of children at the i^{th} level, level 1 being the root. Hence, the number of inputs to the leaves is given by $N = \prod_{i=1}^{k} b_i$. Using (1), we define the forward latency of an uncongested tree, $T(k, b_1, .., b_k)$, as:

$$\begin{aligned} F_{tree} &= F_{b_1} + ... + F_{b_k} \\ &= (\lambda + \beta \log_2 b_1) + ... + (\lambda + \beta \log_2 b_k) \\ &= k\lambda + \beta \log_2 N \end{aligned} \quad (3)$$

5.1 Problem Analysis

Given an application with N memory accesses, we want to (a) determine the k and $\{b_1, ..., b_k\}$ parameters that define the optimal tree for this application, and to (b) assign the N accesses optimally to the leaf level. The goal is to minimize the performance cost of accessing memory through the tree.

Consider the trivial case where a single access makes its way through an uncongested tree. Then, the access-latency through the tree is F_{tree}. Now, assume that the next access is initiated after an interval of t_{next}, and that both accesses have separate routes through the tree, meeting only at the root. Then, the second access will reach the root t_{next} time units after the first access. If $t_{next} \geq C_{b_1}$ (the cycle time of the root node) then no congestion will occur between the two accesses, and the latency of the second access through the tree is also F_{tree}. The *total cost overhead* of accessing the memory through the tree, defined as the total time spent waiting for memory requests being routed, is given by $Cost = (F_{tree} + t_{next})$, if $t_{next} < F_{tree}$, and by $Cost = (2 \times F_{tree})$, if $t_{next} \geq F_{tree}$. In both cases, *Cost* is optimized by minimizing F_{tree}, and we call such accesses *mutually non-concurrent*.

If $t_{next} < C_{b_1}$ the delay of the second access through the tree becomes dependent on the cycle time C_{b_1} of the root node, because the accesses collide at the root. Such colliding accesses are said to be *mutually concurrent*. The total cost overhead is now given by $Cost = (F_{tree} + C_{b_1})$. A group of accesses are said to be mutually concurrent if $t_{next} < C_{b_1}$ for every two successive accesses. Under the assumption that the root is the bottleneck for mutually-concurrent accesses, a group of w such accesses has a total cost overhead of the forward latency through the tree for the first access, plus the cycle time of the root for the following $(w-1)$ accesses.

$$Cost = F_{tree} + (w-1) \times C_{b_1} \quad (4)$$

From (3), we see that F_{tree} is always smallest for a 1-level tree. However, this means that $b_1 = N$ in (4), resulting in an increased C_{b_1} compared with a multi-level tree. Hence, minimizing *Cost* involves a trade-off between the size of the root node and the depth of the tree. In Section 5.3, we will show how this trade-off is related to the application profile.

In the rest of this analysis, we divide all accesses in an application into groups of mutually concurrent accesses, a non-concurrent access forming a group of its own. Furthermore, we make the assumption that the separation between non-concurrent accesses is greater than the forward latency of the tree: $t_{next} \geq F_{tree}$. This simplifies the calculation of the total cost overhead of an application significantly, as it is now independent of t_{next}. The total cost can now be calculated by summing up (4) for each group of concurrent accesses $i \in \{1, 2, \ldots, n\}$.

$$Cost_{total} = \sum Cost_i \quad (5)$$

5.2 Tree Topology Selection

We will now analyze how we can use this $Cost$ model to find the optimal tree topology. The goal of this analysis is to determine how we can trim the search space for an application with N memory

references. We find an upper-bound for the number of tree levels k, for a given N, and we show that we do not need to look beyond some $k = k'$ to find the optimal tree topology. In a nutshell, we want to draw a relationship between N and k that defines a smaller search space.

We start by analyzing a $k = 1$ level tree. From (3), we see that this is optimal when most accesses are mutually non-concurrent with each other. We will now find a cross-over point $N = N_1$, for which this 1-level tree is no longer optimal. We assume that all accesses are mutually concurrent, as this corresponds to the worst case. An N-input, $k = 2$ level tree with r_2 inputs at the root will be more optimal than the 1-level tree, iff:

$$\begin{aligned} F_{tree2} + (N-1) \times C_{r_2} &< F_{tree1} + (N-1) \times C_N \\ F_{tree2} - F_{tree1} &< (N-1) \times (C_N - C_{r_2}) \\ 2\lambda + \beta \log_2 N - (\lambda + \beta \log_2 N) &< (N-1) \times (C_N - C_{r_2}) \\ \lambda &< (N-1) \times (C_N - C_{r_2}) \\ \lambda &< (N-1) \times \log_2(N/r_2) \end{aligned}$$

N_1, the lower-bound of N that satisfies this inequality, can be found if we use a lower-bound for r_2. Hence, by fixing $r_2 = 2$, we can solve for N to find N_1. This solution tells us that for upto N_1 accesses, we never need to search beyond a 1-level tree topology for optimality. Hence, we have bounded the search space for upto N_1 accesses. Now, let us determine N_2, the lower-bound of N for which the optimal tree is always 1 or 2 levels. For N concurrent accesses, a 3-level tree with r_3 inputs at the root is more optimal than the 2-level tree iff:

$$\begin{aligned} F_{tree3} - F_{tree2} &< (N-1) \times (C_{r_2} - C_{r_3}) \\ \lambda &< (N-1) \times (C_{r_2} - C_{r_3}) \end{aligned}$$

We solve for N to find N_2, and observe that the inequality will never be satisfied as long as $r_2 \leq r_3$. This means that the 2-level tree is always better than a 3-level tree. However, the $Cost$ model holds only under the crucial assumption that root node is always the bottleneck during any burst of concurrent access activity. There are two cases under which this assumption breaks down:

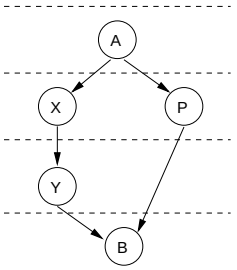
1. The lower-level node experiences congestion while the root is not busy. This, however, is controllable since it depends on how we assign accesses to leaves. As we show later on, careful assignment of accesses to leaves ensures that this condition is easily avoided.
2. If the entire 2^{nd} level (ie. the level below the root) is not fast enough to keep the root node busy, then the bottleneck is not in the root. For example, with an r_2-input root, we would need N/r_2 inputs at each node of the second level of a 2-level tree. The root for such a tree is not the bottleneck if:

$$r_2 \times C_{r_2} < C_{N/r_2} \quad (6)$$

The value of $N = N_2$ for which (6) is just satisfied determines the upper-bound of the search space for 1 and 2 levels. Hence, for any $N > N_2$, the search space would have to include 3-level trees. We can determine N_2 by using a lower-bound for $r_2 (= 2)$. We can easily extend these arguments to $k = 3$ and higher level trees, and determine the various $N = N_k$ that form the upper-bound in the search space for k.

Applying (2), with a bitwidth of $B = 64$ bits and $r_2 = 2$, in (6) we find $N < 375$. This implies that to keep a 2-input root node busy requires level-2 nodes with 187 or fewer inputs. Of course, the analytical model in (2) is an approximation, and the actual cycle times are affected by other factors outside the scope of this model, like wire loads. However, post-layout simulations show that a 32-input node has a small enough cycle time to keep a 2-input root node busy. The cycle time ratio, C_{32}/C_2, of these two nodes is measured in the simulation to be 1.85. This corresponds very closely with the cycle time ratio when computed in terms of logic levels as $cycle\_ratio = 52/28 = 1.86$, while (2) predicts $47.5/29.5 = 1.61$.

We have established that for any given k, there exists some upper-bound N_k, such that the optimal tree has k or fewer levels if the number of concurrent accesses is N_k. As a corollary, for a given N', we can also find an upper-bound k' for the search space.



(a)

Time Step	Accesses	Width
1	A	1
2	X P	2
3	Y	1
4	B	1

(b)

Figure 4: *(a) ASAP schedule creates the (b) Concurrency Table (C-Table): all accesses within a row are deemed mutually concurrent, while accesses from different rows are mutually non-concurrent*

5.3 Tree Construction Heuristic

This section describes a heuristic algorithm that explores the design space to find the best tree topology for a given application with N accesses. First, we need to classify the N accesses into groups of mutually concurrent memory accesses. Typically, a detailed execution schedule is necessary to find this concurrency information. However, since the circuits are dynamically scheduled, we can only predict a possible runtime schedule. A good estimate of the schedule requires detailed analysis of all communication patterns and control paths in not just the MAN, but also in the rest of the circuit. However, the token graph, described in Section 3, is an excellent starting point since any two accesses connected by an edge in this graph are guaranteed to be non-concurrent.

Given a token graph, the heuristic creates an ASAP [1] schedule as shown in Figure 4a. Using this schedule, we form a Concurrency Table (C-Table) in which all accesses scheduled in the same time-step, form a row of the C-Table (Figure 4b). Since they are scheduled concurrently, each row forms a group of mutually concurrent accesses. This approach to finding concurrency relations is accurate in most cases, eg. except for (X,P) and (Y,P), all other concurrency relations in the table are accurate.

Next, we compute the upper-bound tree-depth k for the given N as per our analysis in Section 5.2. We only need to search tree topologies with k or fewer levels to find the optimal tree. For a given tree-depth, we explore the number of inputs at each tree level using the values, $\{2, 4, 8, 16, 32\}$. We start by fixing the inputs of the root node, and then walk down the tree fixing the inputs at the lower levels. For each topology explored, we compute the cost overhead for a given row (with w accesses) in the C-Table,according to (4). The total cost overhead is computed according to (5), by adding up the costs of each row. Finally, we pick the tree topology with the lowest cost.

We now examine the time complexity of this heuristic. For a given tree-depth, if we choose the input-size of the root to be $b_1 = r_1$, then the choices at the next level are limited to N/r_1, and to $\frac{N}{r_1 . r_2}$ in the level after that and so on. Hence, for a given k, the complexity of the heuristic is $O(N^k)$. In practice, k is small (≤ 2 in our benchmarks). Using our analysis from the previous section and results from node simulations, we have observed that the cycle-time of an 8-input node is almost equivalent to that of a 2-input node. Since, a level of 32-input nodes is sufficient to keep both of these busy, for $k = 2$, we can support upto $N = 256$. Since, k scales logarithmically compared to N, the complexity of our algorithm is tractable.

Access-to-leaf assignment. As noted before, it is essential to assign accesses to leaves in such a way that concurrent accesses meet only at the root. This can be enforced if a balanced leaf-assignment

[1] ALAP was also tried, but does not perform as well.

Benchmark	Kernel	Static Refs	Total Top.	Heur. Better
gsm_e	Short_term_ analysis_filtering	5	12	85%
gsm_d	Short_term_ synthesis_filtering	6	12	100%
pgp_d	mp_smul	7	12	100%
pgp_e	mp_smul	7	12	100%
adpcm_e	adpcm_coder	10	17	74%
adpcm_d	adpcm_decoder	9	17	100%
jpeg_e	jpeg_fdct_islow	32	27	100%
mpeg2_d	idctcol	35	27	74%
mpeg2_e	dist1	43	27	93%
jpeg_d	jpeg_idct_islow	68	27	92%

Table 1: *List of the benchmark kernels synthesized and the number of static memory accesses within each kernel. The 4^{th} column shows the number of reference topologies explored and the 5^{th} shows the percentage of these that are inferior in overall performance in comparison with the heuristic*

is made for the accesses within a given C-Table row; all accesses within each row are distributed evenly amongst all leaves. As we show in Section 6, this heuristic sufficiently enforces our assumption that all congestion occurs at the root node. We also note that this strategy does not unduly increase wirelength, which is a concern in deep sub-micron technologies.

Criticality. The heuristic is easily modified to bias some accesses that are on the dynamic critical path of execution. If static analysis reveals criticality for the accesses, then this information can be used in three ways - (a) during topology selection, the total cost overhead of a C-Table is now computed as a weighted total, where a row's weight is the sum of the criticalities of accesses in the row; (b) the tree nodes support static priorities, which can directly be used to bias critical accesses during access-to-leaf assignment. (c) we can bias a critical access by building asymmetric trees, with shorter paths to the root for the critical accesses. However, since the number of tree levels is usually small, the benefit is limited.

6. EXPERIMENTAL RESULTS

This section evaluates the quality of our tree construction heuristic, and the assumptions that the heuristic makes. We have integrated SOMA into CASH [25], an HLS flow that automatically synthesizes clockless, standard-cell circuits in a [180nm/2V] technology, from un-annotated, unrestricted ANSI-C programs [25]. All results reported are extracted from post-layout simulations. Our benchmarks are the most frequently executed kernels from the Mediabench [15] suite. 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 *access tree* of the MAN, and gives us a feel for its complexity.

In order to evaluate our heuristic, we performed a set of reference experiments in which we construct simple, n-ary balanced trees. The accesses are randomly assigned to the leaves of the tree. To explore the design space, we varied n to be any of $\{2, 4, 8, 16, 32\}$. We compared the performance of these trees against the one constructed using our heuristic. In Table 1, the 4^{th} column shows the number of different reference topologies generated, and the 5^{th} column shows the fraction of these that were inferior to the heuristic topology in terms of overall performance. On average, the heuristic performs better than 90% of all reference experiments.

Tree Congestion. First, we evaluate the congestion in the tree during periods of high memory parallelism. When a burst of concurrent memory accesses are initiated, congestion is fundamental and cannot be avoided. However, we can differentiate between three types of congestion depending on where in the tree they occur:

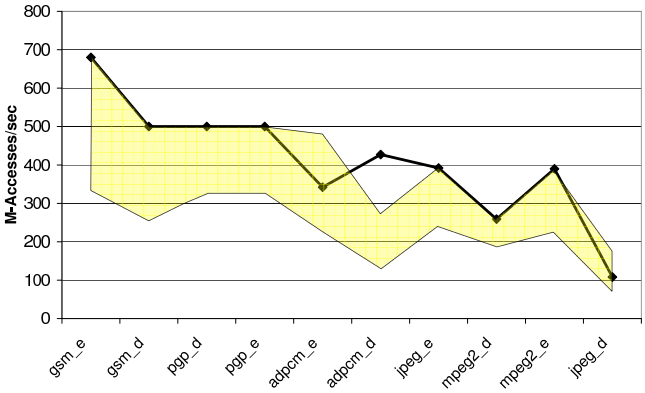


Figure 5: *Throughput: the shaded region marks the throughput space achieved by the random topologies, while the trendline shows the throughput of the heuristic.*

1. Root congestion: is ideal since it means that the concurrent accesses have moved freely up the tree, and are only serialized at the point of exit from the tree; hence, the tree allows for maximal concurrency when the accesses meet each other at the root.
2. Higher-level congestion: Congestion occurs at a given tree node at level, m, only when all tree nodes on the path from level m to the root (i.e., nodes at levels $\{1, \ldots, m\}$) also experience congestion. This is also desirable since it implies that arbitration is being pipelined, and allows for maximal parallelism until the point of serialization.
3. Lower-level congestion only: is bad because we have serialized the concurrent accesses far too early; this wastes the tree resources and can degrade performance. In fact, this is antithetical to the first assumption in our analysis in Section 5.2.

We computed the dynamic congestion in the trees by examining the post-layout simulation traces. Our observation is that there is no type 3 congestion in any of the trees constructed by our heuristic. This confirms our claim that we can easily meet the first assumption in our analysis. In contrast, type 3 congestion in the reference trees account for about 15% of all congestion.

Throughput. From the traces, we identify bursts of w accesses, and note the time interval, t, between the time the first access enters the tree and the last one exits the tree. Throughput for each burst is then $\frac{w}{t}$. The average throughput across all bursts is shown in Figure 5 in terms of Mega-accesses per second. The shaded region of the graph shows the space occupied by the reference experiments, and the trendline shows the performance of the heuristic tree. Notice that the heuristic constructs the best trees most of the time, and its throughput is better than the average reference topology by about 25%. There are two interesting results here:
1. One of the reference topologies for the `jpeg_d` kernel, has better throughput than the heuristic. However, the overall system performance is better for the heuristic because the reference topology experiences a lot of bad congestion.
2. `adpcm_e` is the only kernel for which the heuristic makes some bad decisions. Our analysis of the C-Table for this kernel reveals that a two-level tree with four input root nodes is the best topology, since it expects four accesses to be mutually concurrent most of the time. This is a consequence of using a simple scheduling heuristic that only examines the memory-dependencies in the token graph. Although four accesses can be mutually concurrent in this graph, inspection of the entire dataflow graph that encodes all data dependencies reveals that there exist data dependencies between the accesses; sure enough, the dynamic concurrency is at most two. Hence, analyzing the complete dependency context is important to make better concurrency predictions, and we are currently incorporating this into our heuristic.

Summary. The heuristic presented in this paper improves performance of the baseline MAN architecture. To provide the reader with some context, we compared the performance of the automatically synthesized kernel to that of a superscalar processor running the same C program. The superscalar core was simulated in MASE [14] using a 600 MHz clock at 2V. The geometric mean of performance speedup (over the processor) of the presented benchmarks aggregated across all randomly generated reference experiments is about 1.86x, while kernels synthesized with the heuristic tree construction are about 2.13x faster than the core. In terms of energy-delay, the heuristic construction is about two orders of magnitude better than the superscalar, and about 20% better than the aggregate of the reference MAN topologies. Although the comparison with the superscalar core illustrates the performance of the MAN against a standard implementation, we would have ideally liked to compare the MAN against other existing solutions for synthesizing unconstrained memory accesses. However, to our knowledge, no other HLS tool tackles this problem.

7. CONCLUSIONS

We have presented, SOMA, a framework for synthesizing and optimizing unconstrained memory accesses in high-level synthesis. 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 inherently provides a synchronization mechanism that maintains memory consistency in the context of memory-ordering dependencies that are known only at runtime.

The design space for MAN topologies is large, and for optimality it must match the memory parallelism profile of the application. We have presented a concurrency-based analysis of the MAN topology's performance. The analysis shows that the design space of MAN topologies for a given application can be bounded. This allows our heuristic tree construction algorithm to explore this smaller space by predicting the dynamic memory parallelism of the application, and then selecting the best topology for the application.

While the MAN described here is synthesized as a clockless circuit, there's nothing intrinsic in SOMA that prevents us from implementing it synchronously. In fact, a GALS-like solution may be attractive in some cases. Thus, the SOMA framework can be directly embedded within HLS tools that synthesize circuits from abstractions like C, e.g., System-C, thereby expanding the tool's capability to support arbitrary memory access references.

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.

9. REFERENCES

[1] T. Bjerregaard and J. Sparsø. A scheduling discipline for latency and bandwidth guarantees in asynchronous network-on-chip. In *Async*. IEEE, 2005.

[2] 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.

[3] A. Bystrov, D. J. Kinniment, et al. Priority arbiters. In *Async*, pages 128–137. IEEE Comput. Soc., 2000.

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

[5] G. Corre, E. Senn, et al. Memory accesses management during high level synthesis. In *CODES+ISSS*, pages 42–47. ACM, 2004.

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

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

[8] A. Ghosh, J. Kunkel, et al. Hardware synthesis from c/c++. In *DATE*, page 82. ACM, 1999.

[9] P. Gupta and A. C. Parker. Smash: a program for scheduling memory-intensive application-specific hardware. In *ISSS*, pages 54–59. IEEE Computer Society, 1994.

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

[11] T. Kambe, A. Yamada, et al. A c-based synthesis system, bach, and its application (invited talk). In *ASP-DAC*, pages 151–155. ACM, 2001.

[12] 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.

[13] D. J. Kolson, A. Nicolau, et al. Integrating program transformations in the memory-based synthesis of image and video algorithms. In *ICCAD*, pages 27–30. IEEE Computer Society, 1994.

[14] E. Larson, S. Chatterjee, et al. MASE: A novel architecture for detailed microarchitectural modeling. In *ISPASS*, November 4–6 2001.

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

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

[17] C.-G. Lyuh and T. Kim. Memory access scheduling and binding considering energy minimization in multi-bank memory systems. In *DAC*, pages 81–86. ACM, 2004.

[18] G. Mittal, D. C. Zaretsky, et al. Automatic translation of software binaries onto fpgas. In *DAC*, pages 389–394. ACM, 2004.

[19] S. Y. Ohm, F. J. Kurdahi, et al. A comprehensive estimation technique for high-level synthesis. In *ISSS*, pages 122–127. ACM, 1995.

[20] J. Park and P. C. Diniz. Synthesis of pipelined memory access controllers for streamed data applications on fpga-based computing engines. In *ISSS*, pages 221–226. ACM, 2001.

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

[22] J. Seo, T. Kim, et al. An integrated algorithm for memory allocation and assignment in high-level synthesis. In *DAC*, pages 608–611. ACM, 2002.

[23] G. Stitt, Z. Guo, et al. Techniques for synthesizing binaries to an advanced register/memory structure. In *FPGA*, pages 118–124. ACM, 2005.

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

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

[26] I. M. Verbauwhede, C. J. Scheers, et al. Memory estimation for high level synthesis. In *DAC*, pages 143–148. ACM, 1994.

[27] K. Wakabayashi. C-based synthesis experiences with a behavior synthesizer, cyber. In *DATE*, page 83. ACM, 1999.

[28] W. Wang, T. K. Tan, et al. A comprehensive high-level synthesis system for control-flow intensive behaviors. In *GLSVLSI*, pages 11–14. ACM, 2003.

[29] R. P. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI*.

[30] Y. Zhao and S. Malik. Exact memory size estimation for array computations without loop unrolling. In *DAC*, pages 811–816. ACM, 1999.