Hardware Compilation of Application-Specific Memory-Access Interconnect

Girish Venkataramani, *Student Member, IEEE*, Tobias Bjerregaard, *Member, IEEE*, Tiberiu Chelcea, *Member, IEEE*, and Seth C. Goldstein, *Member, IEEE*

***Abstract*—A major obstacle to successful high-level synthesis (HLS) of large-scale application-specified integrated circuit systems is the presence of memory accesses to a shared-memory subsystem. The latency to access memory is often not statically predictable, which creates problems for scheduling operations dependent on memory reads. More fundamental is that dependences between accesses may not be statically provable (e.g., if the specification language permits pointers), which introduces memory-consistency problems. Addressing these issues with static scheduling results in overly conservative circuits, and thus, most state-of-the-art HLS tools limit memory systems to those that have predictable latencies and limit programmers to specifications that forbid arbitrary memory-reference patterns. A new HLS framework for the synthesis and optimization of memory accesses (SOMA) is presented. SOMA enables specifications to include arbitrary memory references (e.g., pointers) and allows the memory system to incorporate features that might cause the latency of a memory access to vary dynamically. This results in raising the level of abstraction in the input specification, enabling faster design times. SOMA synthesizes a memory access network (MAN) architecture that facilitates dynamic scheduling and ordering of memory accesses. The paper describes a basic MAN construction technique that illustrates how dynamic ordering helps in efficiently maintaining memory consistency and how dynamic scheduling helps alleviate the variable-latency problem. Then, it is shown how static analysis of the access patterns can be used to optimize the MAN. One optimization changes the MAN interconnect topology to increase concurrence. A second optimization reduces the synchronization overhead necessary to maintain memory consistency. Postlayout experiments demonstrate that SOMA's application-specific MAN construction significantly improves power and performance for a range of benchmarks.**

***Index Terms*—Communication synthesis, dataflow synthesis, high-level synthesis (HLS), interface design.**

I. Introduction

THE SCALING of microchip fabrication technologies together with the drive for shortened design cycles have pushed the productivity requirements on chip designers. Between 1997 and 2002, the market demand reduced the typical design cycle by 50%. As a result of larger chip sizes, shrinking geometries, and the availability of more wiring layers, the design complexity increased by 50 times in the same period [1]. As the increase in design productivity continues to lag behind increasing application-specific integrated circuit (ASIC) complexity [2], high-level synthesis (HLS) tools can play a central role in delivering high-performance low-power large-scale ASICs from abstract complex behavioral specifications [3].

With one major caveat, today's HLS tools can often synthesize efficient high-performance circuits. The caveat is that specifications must make all memory dependences statically explicit. Large-scale applications with dynamic memory dependences continue to present an obstacle to HLS for two main reasons: 1) In a hierarchical memory system, accesses may have variable latency, e.g., a cache miss. Hence, these cannot be statically scheduled. 2) More importantly, many natural specifications include memory-reference patterns whose dependences cannot be statically determined. For example, even state-of-the-art sophisticated (and often unscalable) analysis techniques can only disambiguate 60% of all memory dependences in C programs [4]. We know of no HLS solutions that can synthesize memory references with such properties. Although some HLS tools allow some type of memory abstraction, they all impose heavy restrictions on the use of these abstractions. For example, many HLS tools impose a fixed access latency for memory accesses [3], [5], [6]. This is usually the worst case (and often unacceptable) latency for a memory access, and, in general, all of the tools restrict the input specification to statically explicit memory dependences. Pointer aliasing, for example, is disallowed in System-C [7]. These restrictions severely limit the class of specifications to which HLS can be applied. Embedded system-on-a-chip (SoC) design is an example of an application space that can benefit from the relaxation of these restrictions. The software specification of many streaming applications in the Mediabench suite [39], for example, are currently written in ANSI-C with pointer aliasing. If such applications can be synthesized by an HLS flow without modifications to the source, then the design cycle time can be greatly reduced.

There are two main contributions in this paper. First, we describe a synthesis framework (SOMA) that makes HLS more general by efficiently synthesizing specifications and systems that include dynamic memory dependences. This framework, called synthesis and optimization of memory accesses (SOMA), can be embedded within any HLS flow (e.g., System-C), thus expanding the capabilities of the tool.

Second, this paper introduces a highly scalable pipelined distributed arbitration network architecture [memory access

Manuscript received July 15, 2005; revised October 10, 2005. This work was supported in part by the National Science Foundation under Grants CCR-0224022 and CCR-0205523, by DARPA under Contracts N000140110659 and 01PR07586-00, by the Semiconductor Research Corporation, and by an equipment grant from Intel Corporation. This paper was recommended by Guest Editor R. I. Bahar.

G. Venkataramani, T. Chelcea, and S. C. Goldstein are with Carnegie Mellon University, Pittsburgh, PA 15213 USA (e-mail: girish@cs.cmu.edu).

T. Bjerregaard is with the Department of Informatics and Mathematical Modelling, Technical University of Denmark, Lyngby, 2800 Denmark.

Digital Object Identifier 10.1109/TCAD.2006.870411

network (MAN)] that provides for communication between the circuit and the shared-memory resource. This architecture, called MAN, enables higher concurrency among memory accesses, thus raising the memory-level parallelism (MLP) of the application at runtime. For a given input specification (Section III), SOMA synthesizes the MAN, which also implements a dynamic synchronization mechanism that resolves ordering dependences (Section IV) at runtime.

In addition, we present two optimization techniques that improve the basic MAN architecture. Concurrency-based (CB) MAN construction (Section V), performs a static analysis of the patterns of memory accesses in the source application to increase concurrency and reduce congestion in the MAN. The goal of early-token-return (ETR)-based MAN construction (Section VI) is to reduce the dynamic synchronization overhead needed to enforce memory-ordering dependences. This overhead can be large due to conservative design decisions in the basic architecture. We identify the benefits of each technique and discuss how they can be combined together in a memory synthesis flow.

SOMA has been integrated in the HLS toolflow, the compiler for application-specific hardware (CASH), which synthesizes unrestricted ANSI-C programs into pipelined clockless circuits [8]. We use postlayout simulation of CASH-generated circuits to highlight the usefulness of our concurrency analysis, and the robustness of the heuristics applied, in improving the MAN performance (Section VII). The techniques described in this paper can easily be adapted to HLS flows that use other design styles, e.g., globally asynchronous locally synchronous (GALS), clocked synchronous, etc.

II. Related Work

A. Specification Support

There are four broad categories of work that relate to supporting memory references in HLS.

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 [9]–[13]. Séméria *et al.* [14] 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 *et al.* [15] uses tree height reduction to consider memory-access latencies and redundancies in forming a schedule. A recent study by Stitt *et al.* [16] shows 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], [6], [17]–[19]. Most of these efforts start with a control dataflow-graph 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 [5], [6]. There are also some efforts that simultaneously consider both memory-access scheduling and memory allocation [20], [21].
4) **Tool support:** A number of C-like toolflows [22]–[27] 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 [7].

The first two categories contain research that is orthogonal to the focus of our paper. Methods from these areas can be used in conjunction with our techniques. The common feature among all the studies in the last two categories 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 worst case schedules, and hence, none of the above techniques address this problem.

Our study differs from all of the above and, to the best of our knowledge, is the first HLS system that can support input specifications that can include arbitrary memory-reference patters, i.e., a specification can include memory references that cannot be disambiguated until runtime. Our tool uses an explicit memory-dependence representation, which becomes a runtime synchronization construct. Hence, all memory accesses are dynamically scheduled once their dependences have been dynamically resolved.

B. Architecture Support

Although some tools support memory accesses whose ordering dependences are statically determinable [3], [13], [19], [28]–[31], the memory-interface design is quite different from our solution. Since all memory accesses are always statically scheduled in these approaches, contention between memory accesses is statically reduced or eliminated. Further, there is usually a single centralized memory controller that provides access to memory. In [19] and [30], memory accesses are bus based; in [30], memory operations are statically scheduled and will never contend for the bus, while in [19], a global controller selects memory accesses from several FIFOs in a round-robin fashion. Luthra *et al.* [13] allow concurrent accesses to multiple memory banks, but they place important restrictions on the input language to be able to partition the memory into separate banks. Huang *et al.* [29] rely on a platform-based architecture that obviates the need for elaborate memory-access interfaces that must support concurrence. Finally, an interesting architecture is proposed in [3]: Memory accesses flow through a flexible FIFO, which communicates with a centralized memory controller. The FIFO polls the controller to see whether a memory access is completed or not. However, this solution is based on the particular networking application implemented in their paper, and it is not clear how it can be generalized to arbitrary number of memory accesses and unknown memory latencies.

In contrast, we take a distributed approach, and our solution is intended to enable higher concurrence. Our system allows for a large number of dynamically scheduled potentially concurrent accesses to be initiated. The major novelty and challenge of our solution is to provide scalable arbitration between these

```
int A_arr[100], B_arr[100];
int foo (int* ptr, int idx, int offset)
{
    int result =    A_arr[idx]    //lod 1
                  + B_arr[idx]    //lod 2
    if (idx)
     result += *(ptr+offset);     //lod 3
    else
     *ptr = result;               //str 1
    return result;
}
```

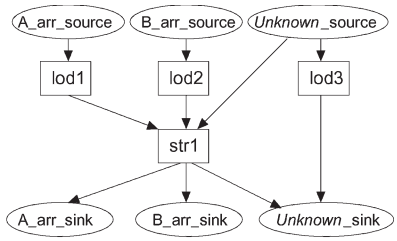


Fig. 1. Code fragment and associated memory may-dependence flow graph.

contending accesses, thereby efficiently sustaining high levels of memory concurrence.

III. Dependence Representation

The input specification to SOMA is a flow graph in which nodes represent unique memory accesses in the source program and edges represent synchronization tokens. A token edge between two accesses indicates that both may be assigned to the same memory location at runtime. In other words, a token edge, or a may-dependence exists between any two references that cannot be statically proven to be independent. A may dependence in this compiler representation translates to an ordering dependence at runtime. The producer of the token must perform its memory access before the recipient of the token. Thus, the token graph explicitly represents a partial ordering of memory accesses through these tokens. Alias analysis is used to eliminate false dependences, and assigns memory accesses to unique location sets [32]. In the worst case, when nothing can be statically disambiguated, there will exist a single location set representing the entire memory block.

The transfer of a token along a token edge is also modeled as a runtime construct. Thus, at runtime, a memory access is initiated only after it receives tokens along all of its input edges. After accessing the memory, tokens are released to all its successors. Thus, memory accesses are, in essence, dynamically scheduled, which is a necessary requirement in any efficient synthesis framework supporting memory-access dependences that can only be dynamically disambiguated.

Consider the code and its token representation shown in Fig. 1. 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, reference different location sets (`A_arr` and `B_arr`, respectively), and therefore, there is no edge between them. Nothing is known about `ptr`, and hence, it is associated with $Unknown$. To preserve memory consistency, we must assume that ptr can point to either of the other two location sets. Since lod3, lod1, and lod2 are all memory reads, there is no need to add edges between them. Similarly, no synchronization is needed between lod3 and str1 because they occur in different branches of the if−else statement. However, we do need to synchronize lod1 and lod2 with str1. At runtime, each edge forms an ordering dependence that the synthesized MAN must enforce. Elimination of redundant dependences results in lesser synchronization at runtime; this has been addressed in [33], and is orthogonal to the focus of this paper. The concept of using tokens to explicitly represent memory-ordering dependences was first proposed in the context of compilers [34] for static program analysis. To our knowledge, we are the first to synthesize these into runtime constructs.

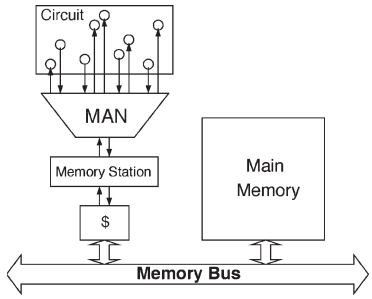


Fig. 2. System architecture.

IV. MAN Architecture

This section describes the synthesis of the MAN for a given token-graph specification. We assume that all accesses are to a shared monolithic single-ported memory; thus, one of the goals of the MAN is to sustain concurrence in the face of contention for this resource. The framework proposed in this paper can, however, be extended to cover systems with multiported memories, or multiple distributed memories.

A. Basic Architecture

The system architecture is shown in Fig. 2. The MAN is an interconnection network that provides communication between the main shared-memory resource and a number of access points, distributed throughout the circuit. As indicated in the figure, one or more caches can be part of the memory hierarchy. The MAN must:

1) support multiple potentially concurrent accesses to the memory system;
2) route data read from memory (load-value responses) to the appropriate destinations;
3) enforce memory-ordering dependences.

The MAN consists of three trees associated with the three MAN requirements described above: The access tree routes requests to the memory, the value tree routes values read from the memory to their destinations, and the token tree enforces ordering dependences. The challenge of constructing the access tree is that there are multiple memory-access initiation points in the circuit, and all of them can potentially initiate an access concurrently. Thus, the access tree must arbitrate among

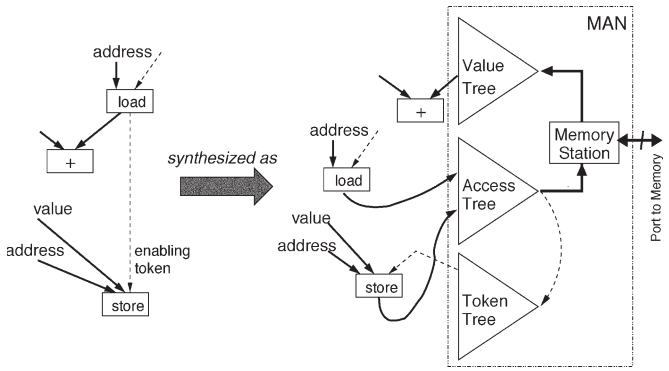


Fig. 3. Example of MAN synthesis.

a potentially large number of concurrent requests, offering sequential access to a memory station. The efficient construction of the access tree is the subject of Sections V and VI.

The interface to memory is implemented by the memory station. In addition, it also performs some bookkeeping to remember the route to be taken by load values that return from memory. Upon arrival at the memory station, a load value is sent along this route through the value tree.

Memory ordering can be guaranteed if the partial ordering of memory accesses, as defined by the token edges in the input graph, is maintained at runtime. In the final circuit, these token edges are synthesized as dataless handshake channels—a completion of the handshake on one of these channels corresponds to the transfer of the token along the associated graph edge.

Fig. 3 illustrates the basic architecture of the proposed MAN, and how a simple example flow graph with dependent memory operations is synthesized. The data read from memory by the load is used by an adder. There is a memory-ordering dependence between the load and the store. In this example, the circuit must guarantee that the load access is performed before the store. By releasing the load's output token from a common point on the two accesses' routes to the memory, we can enforce this guarantee. The root of the access tree is guaranteed to be such a common point. Since access delivery from this point to the memory is guaranteed to be in order, and since the store cannot initiate the access until it receives this token, memory ordering is enforced. The released token is routed down the token tree to the store.

An important observation in this token-synchronization architecture is to notice that we do not wait until the load is actually performed before releasing its token. Instead, we enable the dependent store as soon as we can guarantee that the consumer of the token cannot overtake the producer on its route to the memory. The chosen common point, the access-tree root, is thus a much closer synchronization point than the main memory itself.

The encoding of the access-tree datapath contains: 1) access-specific information including memory address, store value, and some control bits; 2) the path to be taken by the token through the token tree; and 3) if the access is a load, then the path to be taken by the load value through the value tree. The token-tree and value-tree datapaths both encode the route taken through the respective trees. In addition, the value-tree datapath also contains a field for the value that was read from memory.

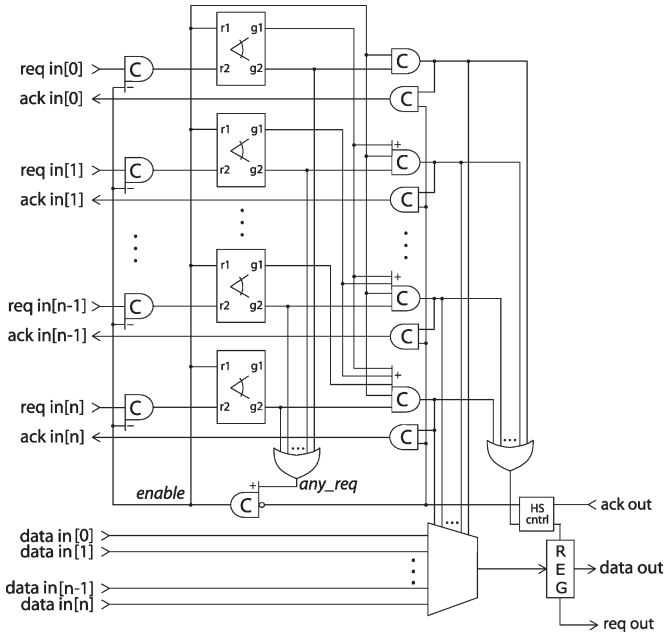


Fig. 4. Handshake multiplexer implements access tree node.

B. Building Blocks

We will now describe how these trees are implemented. 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, this communication can be implemented synchronously, as a GALS architecture, or, as we have done in this paper, as a completely self-timed or clockless implementation. Clockless circuits are characterized by the absence of a global synchronization signal such as a clock. They are data driven, and all flow control is handled by local handshaking mechanisms. We make use of the four-phase bundled-data push protocol that uses two control signals, request (req) and acknowledge (ack), to implement the handshake protocol in a communication channel [35]. The request signal being driven high by the producer side of the channel indicates that data are valid. The consumer responds by driving the acknowledge signal high, indicating that the data has been accepted. What follows is a return-to-zero phase, during which the producer resets the request first, then the consumer resets the acknowledge.

Each tree is constructed out of pipelined handshake blocks. A node in the access tree is an arbitrating handshake multiplexer, as shown in Fig. 4. 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 avoid metastability when arbitrating between these. The key control signal for this arbitration 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 [36], which function as asynchronous lock registers. The function of a mutual exclusion element is to allow only one of its inputs to propagate to the output, and it contains a metastability filter. When enable is high, arriving input requests cannot propagate through the lock registers. Enable going high starts the arbitration phase. In this phase, the inputs, which are

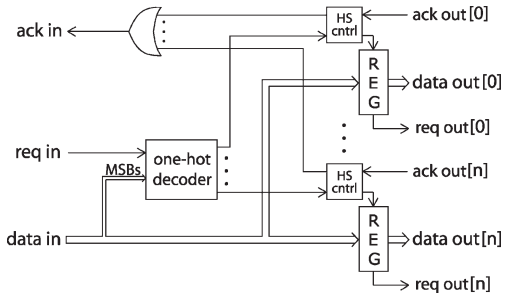


Fig. 5. Handshake demultiplexer used in value/token trees.

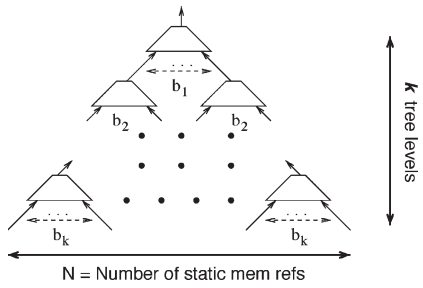


Fig. 6. Generic definition of balanced access tree. It has k levels, and each tree node at level i has b_i inputs. Each of N static memory references has dedicated leaf-level input port.

locked and stable to the right of the lock registers, are arbitrated according to a static priority. This design (based on [37]) is optimized for throughput. It was first used in [38].

The nodes in the token and value trees are implemented as handshake demultiplexer elements, as illustrated in Fig. 5. The demultiplexor selection signal is encoded in the datum itself, and provides the routing information for demultiplexing. The difference between the value and token-tree nodes is that the former have a wider datapath for the load values read from memory.

Given the input specification, SOMA uses these building blocks to automatically synthesize an application-specific MAN. Each static memory reference in the specification is synthesized into a memory-access initiation point. Each of these points connect to a dedicated input port at the leaves of the access tree. The value and token trees have as many output ports as there are load-value and token destinations, respectively. The result of the synthesis is a standard cell-based gate-level implementation of these tree nodes with customized datapath widths.

V. CB MAN Construction

For a single-ported monolithic memory, the construction of the value and token trees is simple since no congestion can occur. The 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. We present an access-tree-construction technique in this section, which balances latency through the tree with the throughput demands of the application, as determined by a static program analysis.

A. Modeling Access Tree

In this section, we derive a performance cost model for the access tree, which is then used to determine the optimal tree topology for a given application. The two important application-specific metrics used are: 1) the total number of memory access points,[1] which is referred to as N throughout this analysis and 2) the MLP of the application. The cost model is constructed based on the timing characteristics of the underlying nodes that make up the tree.

1) Node Model: An access tree node can be characterized by two parameters: forward latency f and cycle time C. 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, assuming that there is no congestion on the output channel. An analysis of the circuit implementing a j-input tree node (as shown in Fig. 4) yields the worst case costs of the forward latency (f_j) and cycle time (C_j) for a given bit width as

$$f_j = \lambda + \beta \log_2 j \tag{1}$$

$$C_j = \tau + \gamma \log_2 j. \tag{2}$$

The j-dependent term arises from the need to drive and merge internal signals that scale with the number of inputs. The constant terms λ and τ themselves scale logarithmically with the bit width of the data path. This is due to the need to drive internal nodes that scale linearly with the bit width. We have evaluated the constants of (1) and (2) in terms of logic levels to be

$$\lambda = 15 + \log_4 B, \qquad \beta = 4.5 \tag{3}$$

$$\tau = 22 + \log_4 B, \qquad \gamma = 4.5 \tag{4}$$

where B is the bit width of the data path. We use step-up drivers with a step-up value of approximately 4, hence the log base of 4. Note that these are analytical approximations based on the circuit architecture. In an actual implementation, the parameters have discontinuities; e.g., when going from eight to nine 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 the analysis presented in this section.

2) Tree Model: Using (1) and (2), we derive a cost model, in terms of latency and bandwidth, for an entire access tree. We limit the discussion in this section to balanced access trees. Section VI addresses an asymmetric access-tree construction. A balanced access tree is illustrated in Fig. 6. It has k levels, and each tree node at level i has b_i inputs, or children; $i = 1$ represents the root level. Each of the N memory access points in the application has a dedicated input port to some leaf of this tree. Thus, $N = \prod_{i=1}^{k} b_i$. Using (1), we can now model forward latency, i.e., the time to travel from a leaf to the root when the tree is uncongested. This is given by

$$\begin{aligned} F(k) &= f_{b_1} + \cdots + f_{b_k} \\ &= (\lambda + \beta \log_2 b_1) + \cdots + (\lambda + \beta \log_2 b_k) \\ &= k\lambda + \beta \log_2 N. \end{aligned} \tag{5}$$

[1] A static memory reference in the application is synthesized into a memory access point.

For the trivial case where a single access makes its way through an uncongested tree, the access latency through the tree is $F(k)$. 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 $t_{\text{next}} \geq C_{b_1}$ (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 $F(k)$. 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 $= (F(k) + t_{\text{next}})$, if $C_{b_1} < t_{\text{next}} < F(k)$, and by Cost $= (2 \times F(k))$, if $t_{\text{next}} \geq F(k)$ and $t_{\text{next}} > C_{b_1}$. We call such accesses mutually nonconcurrent, and Cost for these accesses can be minimized by minimizing $F(k)$, the forward latency through the tree.

If $t_{\text{next}} \leq C_{b_1}$, then 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, and their cost overhead is now given by Cost $= (F(k) + C_{b_1})$. A group of accesses are said to be mutually concurrent if $t_{\text{next}} < C_{b_1}$ 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

$$\text{Cost} = F(k) + (w-1) \times C_{b_1}. \tag{6}$$

Based on (5) and (6), we notice that the two main factors that affect Cost are as follows.

1) Average MLP: The number of memory accesses that are predicted to be concurrently initiated most commonly during program execution. This affects w in (6).
2) Total number of accesses N: As the number of access points N increases, either $F(k)$ or C_{b_1} must increase.

As can be seen from (5), a $k = 1$ (or a one-level) tree minimizes $F(k)$. However, from (2) and (6), we notice that this maximizes C_{b_1} to C_N. On the other hand, a multilevel $(k > 1)$ tree increases $F(k)$ but reduces C_{b_1}. Our tree-construction algorithm minimizes Cost by managing this tradeoff between the depth of the tree and the size of the root node.

For a given N, the access points attached to the leaves, there are a variety of topologies ranging from the shallowest tree $(k = 1)$ to the deepest tree $(k = \log_2 N)$. To understand the tradeoff between $F(k)$ and C_{b_1}, we ran experiments using the two extreme tree topologies: a one-level tree [to minimize $F(k)$], and a binary tree (to minimize C_{b_1}) with $\log_2 N$ levels. We synthesized the most frequently executed kernels from the Mediabench [39] suite, as shown in Table I. The last two columns represent N, the number of memory access points in the kernel, and MLP_a, the statically estimated average MLP of the kernel, respectively. MLP_a is estimated by the compiler as described later in Section V-C.

Fig. 7 shows the relative performance speedup of the $k = 1$ topology (one-level) over the $k = \log_2 N$ topology (binary).

TABLE I
LIST OF BENCHMARK KERNELS SYNTHESIZED AND AVERAGE ESTIMATED MLP

Benchmark	Kernel	Static Refs, N	Avg MLP, MLP_a
pgp_d	mp_smul	7	1.2
pgp_e	mp_smul	7	1.2
gsm_e	Short_term_ analysis_filtering	5	1.3
gsm_d	Short_term_ synthesis_filtering	6	2.1
adpcm_e	adpcm_coder	10	2.8
adpcm_d	adpcm_decoder	9	3.3
jpeg_e	jpeg_fdct_islow	32	8
mpeg2_e	dist1	43	8.8
jpeg_d	jpeg_idct_islow	68	14
mpeg2_d	idctcol	35	17

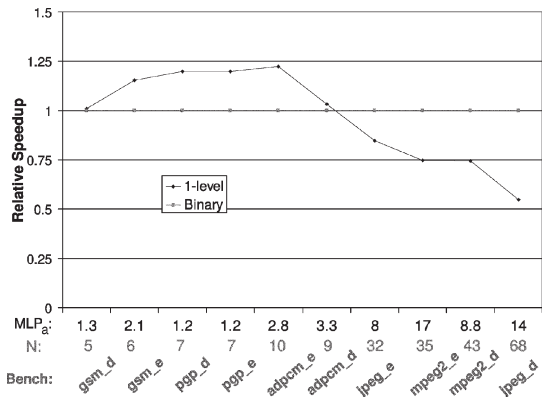


Fig. 7. Relative performance of $k = 1$ (one-level) versus $k = \log_2 N$ (binary) trees.

Notice that when MLP_a is small, the one-level topology is better. However, as MLP_a increases, the binary tree tends to be the superior choice.

The optimization presented in this section constructs a tree with the smallest depth that still supports the MLP demands of the application. Given the potentially large search space for a given N, our first phase shows how we can restrict the size of this space, while still ensuring that the optimal topology lies within this restricted space. In the second phase, we search this restricted space to find the best topology match for the statically estimated MLP of the application.

B. Topology-Space Pruning

The goal of this analysis is to show that for a given N, we can restrict the search space to trees that are at most d levels deep, where $d \ll \log_2 N$. Further, we show that the tree with the lowest possible cost, according to (6), exists within this space.

In this analysis, we assume that all N accesses are expected to be mutually concurrent; i.e., $w = N$ in (6). The reason is as follows: to minimize Cost, the depth k of the tree must be increased only when the expected concurrence w is high, and the $(w-1) \times C_{b_1}$ term of (6) dominates. With a lower w, $F(k)$ dominates, hence (6) is minimized with a smaller tree depth. The pruned space accounts for all N accesses being concurrent. With any less concurrence, shallower trees may be more cost effective. Since these shallower trees are still members of the

pruned space, the optimal tree topology for a given N is contained within this space, irrespective of the application's MLP.

Our approach to this analysis is to find the largest N that an optimal k-deep tree can support. As a result, we will define a relation R such that $R(k') = N'$ implies that for up to N' accesses, the lowest cost topology is at most k' levels deep. To find $R(k')$, we first determine the lowest cost of a tree with $(k'+1)$ levels. Then, we compare this to the cost function of a k'-deep tree to determine the maximum N', beyond which we can no longer prove that there exists some k'-deep tree that could improve upon the cost of the $(k'+1)$-deep tree. Thus, $R(k')$ is defined to be N'.

1) Bounding Relation for $k = 1$: We first find $R(k=1)$. From (5), we know that $F(k)$ is constant for a given k. Hence, the lowest cost two-level tree must simply have the smallest root size, and therefore, the smallest C_{b_1}. Thus, the lowest cost is $\text{Cost}_{k=2} = F(2) + (N-1) \times C_2$. If we compare this to the cost of a one-level tree, we find that a one-level tree is better than a two-level tree as long as the following inequality holds:

$$\begin{aligned} \text{Cost}_{k=2} &> \text{Cost}_{k=1} \\ F(2) + (N-1) \times C_2 &> F(1) + (N-1) \times C_N \\ F(2) - F(1) &> (N-1) \times (C_N - C_2) \\ \lambda &> (N-1) \times (C_N - C_2) \\ \lambda &> (N-1) \times \log_2\left(\frac{N}{2}\right). \end{aligned} \quad (7)$$

Notice that the left-hand side (LHS) of (7) is constant, but the right-hand side (RHS) is monotonically increasing in N. Thus, $R(1)$ is equal to the largest value of N, for which (7) is satisfied.

2) Bounding Relation $k \geq 2$: Now, consider what happens when we use the same approach to find $R(2)$. The lowest cost three-level tree must also have a two-input root, hence, $\text{Cost}_{k=3} = F(3) + (N-1) \times C_2$. However, since a two-level tree can also employ a two-input root, its cost remains $\text{Cost}_{k=2} = F(2) + (N-1) \times C_2$. We know from (5) that for a given N, $F(2) < F(3)$. Thus, under the assumption in (6) that the cycle time of the root is the only bottleneck to throughput, we may conclude that $\text{Cost}_{k=2}$ is always smaller than $\text{Cost}_{k=3}$.

The assumption of (6) is that the rate at which memory accesses are generated by the tree is constrained only by the cycle time of the root C_{b_1}. However, if the cycle time of the level below the root is larger than the time it takes for the root to service inputs from all its children, then the bottleneck is not in the root, and (6) is no longer valid. Of course, this condition may exist at any level of the tree. Thus, for a node at level i, a bottleneck condition occurs at level $i+1$ if

$$b_i \times C_{b_i} < C_{b_{i+1}}. \quad (8)$$

When (8) holds, the actual Cost may be larger than that computed by (6), and thus we cannot use (6) to compare costs. Our approach to finding $R(k)$ for $k \geq 2$ is to find an upper bound for N, call it $N_{\max_k}$, such that (8) is not satisfied at any level i in the tree, and (6) still remains valid. We can then compare this tree with one that has $(k+1)$ levels, using (6).

This approach to finding $R(k)$ is conservative; if (8) is satisfied for some k, we conservatively settle for a space bounded by a larger $(> k)$ depth. However, the optimal tree may in fact be a k-deep tree. Though we run the risk of finding a larger space, this makes the analysis easier.

We now show how we can define the topology of this (maximal sized) lowest cost k-deep tree. Since, for the same k, the lowest cost is obtained with the smallest C_{b_1}, this tree always has a root size of 2. The maximum number of concurrent accesses that can be sustained by such a k-deep tree is $N_{\max_k}$, the highest N for which (6) is still valid. $N_{\max_k}$ can be calculated as follows

$$b_i = \begin{cases} 2, & \text{if } i = 1 \\ \text{bmax}_i \left| \left(C_{b_{i-1}} \times b_{i-1}\right) = C_{\text{bmax}_i}, \right. & \text{if } i > 1 \end{cases}$$

$$N_{\max_k} = \prod_{i=1}^{k} b_i.$$

Notice that the construction of this tree is such that increasing N to $(N_{\max_k} + 1)$ causes (8) to be satisfied for some level i in the tree. Then, we can no longer compute Cost using (6), while we can do so when $N = N_{\max_k}$. If we compare the lowest cost for this maximal k-deep tree and for a $(k+1)$-deep tree that also supports $N_{\max_k}$, we find that the k-deep tree has a lower cost if

$$F(k) + (N_{\max_k} - 1) \times C_2 < F(k+1) + (N_{\max_k} - 1) \times C_2.$$

Since $F(k) < F(k+1)$, the above inequality is always satisfied. Thus, we have observed that for up to $N_{\max_k}$ concurrent accesses, we can use (6) to prove that a k-deep tree will always have a lower cost than any tree with more than k levels. We can summarize the search for $R(k \geq 2)$ as follows.

1) Given that (8) is not satisfied anywhere in the tree, the lowest possible cost for any tree of depth k is always $\text{Cost}_k = F(k) + C_2 \times (N-1)$. This is because $F(k)$ is constant for a given N and k, and by using a two-input root, we minimize (6).
2) For a given N, the forward latency of the tree $F(k)$ grows monotonically with k. Thus, when comparing two trees with k and $(k+1)$ levels, the k-deep tree always has a lower Cost as long as it has a two-input root, and (8) is not satisfied at any level, $i \leq k$.
3) Finally, we find the largest $N = N_{\max_k}$ for which the above condition is met. As described earlier, the value of $N_{\max_k}$ represents this upper bound on N. Thus, $R(k) = N_{\max_k}$ for any $k \geq 2$, because for any $N \leq N_{\max_k}$, we can prove that there exists a k-deep tree achieving a lower cost than that of any tree with $(k+1)$ levels.

Observe the importance of (8). Note that if (8) is satisfied for some k-deep topology supporting N accesses, it only implies that the actual cost of the topology is greater than that predicted by (6). Hence, as per our approach, we would conservatively settle on a larger depth $(> k)$ topology, for which (6) is valid. However, a k-deep topology with a three-input root may achieve a lower cost than that determined for the $(k+1)$-deep tree. However, the k-deep topology will still lie within the topology space bounded by $(k+1)$ levels, and thus, we would

still satisfy our goal for this analysis. This is sufficient for the purposes of our heuristic presented in Section V-C, since the whole space is searched to find the best topology match for the given application.

3) Discussion: Applying (2), with a bit width[2] of $B = 70$ bits and $n = 2$, in (8), we find the cycle_ratio $= C_{187}/C_2 = 2$. This implies that to keep a two-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 and other circuit effects. For realistic estimates, we conducted some postlayout experiments to find the actual C_i and f_i times for nodes with up to $i = 68$ inputs, and extrapolated these data for larger i. We found that the largest node that can keep a two-input parent node busy is a 32-input node. This implies that a realistic two-deep tree can sustain up to 64 memory accesses.

Using these simulation results, we also found $\{R(1), R(2), R(3)\}$ to be {8, 64, 512}. Thus, for an application with up to 512 concurrent accesses, a three-level tree will suffice to deliver optimal throughput. Even for large values of N, the search space could be pruned substantially. Notice that the biggest benchmark in our experiments (Table I) only has 68 memory accesses, and a two-level tree has turned out to suffice for all.

Further, remember that these results are under worst case conditions when all N accesses are mutually concurrent, i.e., $w = N$, which is rarely the case in a real application. If the actual concurrence (or MLP_a) is much smaller, then we can shrink the depth of the tree, thus optimizing forward latency, since a fast cycle time at the root is not as essential any more. Although the depth of the tree grows sublinearly with N, the size of the circuit for the tree grows linearly with N. However, as we show in Section VII, the size of the entire MAN is a small fraction of the overall circuit size.

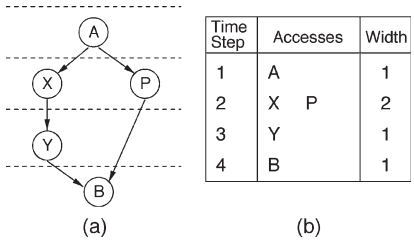
C. Tree-Topology Selection

This section describes an algorithm that explores the pruned space to find the best tree topology for an application with N memory access points, and a given MLP. The first step in finding the best topology match is to estimate the memory concurrence in the application. For this purpose, we classify the N_app accesses into c groups of mutually concurrent memory accesses. The total cost overhead for the application is then the sum of the cost overhead of each of these groups

$$\text{Cost}_\text{app} = \sum_{i=1}^{c} \text{Cost}_i. \tag{9}$$

Cost_i is computed using (6). The value of w used in Cost_i is the size $|c_i|$ of the ith group of mutually concurrent accesses. Cost_app is the objective function we use in evaluating the different topologies against the application's access profile.

1) Mutual Concurrence Classification: Precise concurrence information is, in principle, obtainable from a detailed examination of an accurate execution schedule. Such a schedule is not always available at synthesis time, e.g., in our case, the circuits are dynamically scheduled. However, the token graph provides an excellent approximation since any two accesses connected by a token edge are guaranteed to be nonconcurrent.



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

Fig. 8. (a) ASAP schedule creates (b) C-table. All accesses within a row are deemed mutually concurrent, while accesses from different rows are mutually nonconcurrent.

Given a token graph, the concurrence estimator creates an as-soon-as-possible (ASAP) schedule, as shown in Fig. 8(a) (as-late-as-possible (ALAP) scheduling is not feasible for pure dataflow machines without centralized controllers). Using this schedule, we form a concurrence table (C-table) in which all accesses scheduled in the same time step form a row of the C-table [Fig. 8(b)]. Since they are scheduled concurrently, row i in the table forms the ith group of mutually concurrent accesses. This approach to finding concurrence relations is accurate in most cases, e.g., except for (X, P) and (Y, P), all other concurrence relations in the table are accurate under all runtime conditions. The estimated MLP_a of an application as shown in Table I is the average width over all rows in the C-table.

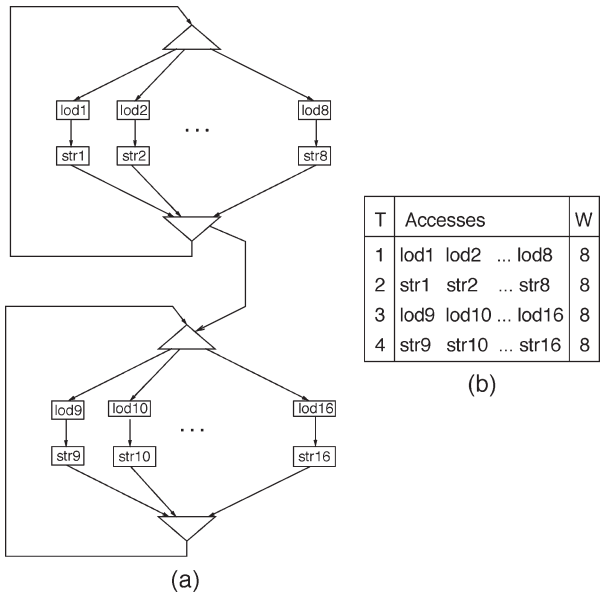
2) Topology Evaluation: For an application with N_app, we can determine the upper bound tree depth k_app within which the optimal tree lies. Given this bound, we enumerate all possible topologies that are of depth k_app or lesser. For each topology in the space, we compute the cost overhead for a given row (with w accesses) in the C-table, according to (6). The total cost overhead is computed according to (9), by adding up the costs of each row. Finally, we pick the tree topology with the lowest cost. The complexity of this search corresponds to the size of the pruned topology space. In practice, k_app is small (as described in Section V-B3), and our algorithm is scalable.

3) Access-to-Leaf Assignment: In addition to the condition stated in (8), another requirement in the cost model is that accesses must be assigned to leaves such that, at runtime, concurrent accesses meet at level i, if and only if all the nodes at levels $j < i$ are busy. In other words, if two concurrent accesses are injected through their respective leaf ports into an otherwise idle tree, then the accesses must meet only at the root.

This requirement is enforced heuristically by a balanced leaf assignment of the accesses within a given C-table row. All accesses within each row are distributed evenly among all leaves. As we show in Section V-D, this strategy is effective.

4) Criticality: A further optimization is to account for accesses that are on the dynamic critical path of execution. We use static analysis to predict a quantifiable criticality factor for each memory access; for example, accesses within loops are always deemed more critical than those outside loops. Now, each C-table row's criticality is defined as the sum of the criticalities

[2]A memory access packet has 32 bits each for the address and value and some additional control bits.



T	Accesses	W
1	lod1 lod2 ... lod8	8
2	str1 str2 ... str8	8
3	lod9 lod10 ... lod16	8
4	str9 str10 ... str16	8

Fig. 9. Token flow graph for jpeg_e.

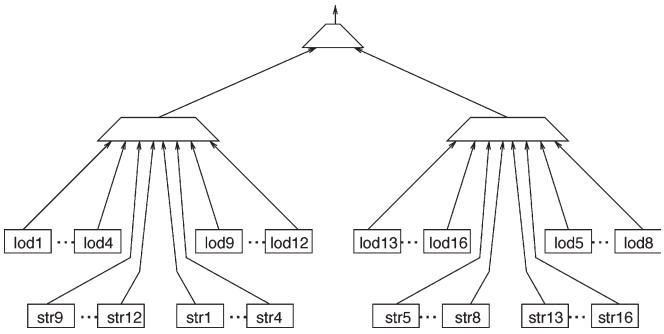


Fig. 10. Final access tree for jpeg_e.

of its access members. The total cost overhead (9) is now computed as a weighted total, where the weights are the row criticalities. This has the effect of biasing the tree construction in favor of more critical accesses.

5) Example (jpeg_e): We now present an example that demonstrates the heuristic presented in this section. Fig. 9(a) shows the token flow graph for the jpeg_e kernel. It consists of two sequential loops with 16 accesses in each loop. In each loop, there is a burst of eight concurrent load accesses that are then followed by eight concurrent store accesses. An edge between any two accesses signifies an ordering dependence.

The ASAP schedule for this kernel is as shown in Fig. 9(b). Using the model in Section V-B, we determined that the optimal tree for this kernel has a depth of at most two levels. After evaluating each topology within this space, the lowest cost topology was found to be a $[k = 2, b_1 = 2, b_2 = 16]$ tree, as shown in Fig. 10.

Finally, to ensure proper access-to-leaf assignment, accesses from each row of the C-table are distributed evenly among all leaves. Hence, as shown Fig. 10, half of the accesses from each row are assigned to each of the two children of the root. As shown in the next section, this topology turns out to be the best for the concurrence profile of jpeg_e.

D. Experimental Evaluation

We will now evaluate the quality of our tree-construction heuristic, and the assumptions that the heuristic makes. We have incorporated the heuristic into CASH [8], an HLS flow that automatically synthesizes clockless standard-cell circuits in a [180-nm/2-V] technology, from unannotated unrestricted ANSI-C programs. All results reported in this section and the rest of the paper are extracted from postlayout simulations. Our benchmarks are the most frequently executed kernels from the Mediabench [39] suite, as listed in Table I.

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 $\{2, 4, 8, 16, 32\}$; and a one-level tree was also included. This creates a variety of topologies whose depths range from $k = 1$ to $k = \log_2 N$. We compared the performance of these trees against the one constructed using our heuristic. For each kernel, we simulated between 15 and 30 reference experiments depending on the size of the search space for that kernel.

Tree Congestion: First, let us evaluate the congestion in the tree during periods of high memory parallelism. When a burst of concurrent memory accesses is 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.

1) Root congestion: This is when congestion occurs only at the root level. This 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 concurrence 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: Congestion occurs at level m when there exists some node at a higher level $k < m$ that is uncongested. This is bad because we have serialized the concurrent accesses far too early; this wastes the tree resources and can degrade performance.

We computed the dynamic congestion in the trees by examining the postlayout 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 when the first access enters the tree and the last one exits the tree. The throughput for each burst is then w/t. The average throughput aggregated across all bursts is shown in Fig. 11 in terms of mega-accesses per second. The shaded region of the graph shows the space occupied by the reference experiments, and the

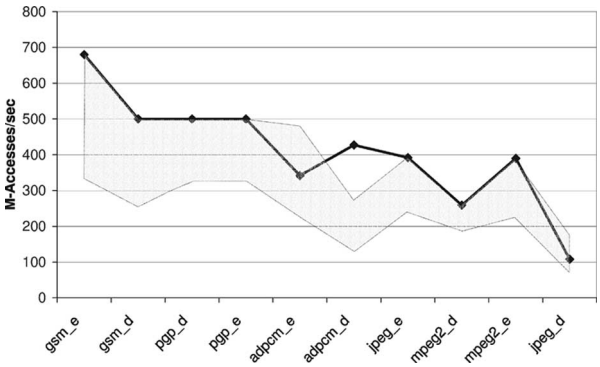


Fig. 11. Throughput: shaded region marks throughput space achieved by random topologies, while trendline shows throughput of heuristic.

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. Note that the reference topology has type-3 congestion, and the overall system performance of the heuristically constructed MAN is superior to that of the reference topology.
2) adpcm_e is the only kernel for which the heuristic does not deliver the best performance. Our analysis of the C-table for this kernel reveals an MLP of 2.8, and the heuristic constructs a two-level tree with a four-input root. However, the observed runtime MLP is much lower (< 2) for this kernel. This is a consequence of using only the token graph to determine the MLP. Although four accesses were observed to be mutually concurrent in this graph, an inspection of the entire dataflow graph with all data dependences reveals that there exist additional data dependences between the accesses. Hence, analyzing the complete dependence context is important to make better concurrence predictions, and we are currently incorporating this into our tool.

Summary: In addition to better throughput, our heuristic also results in better overall timing, lower power, and better energy delay of the entire application: the heuristic achieves improvements of about 19% in performance, and about 20% in energy delay, over the average reference topology. Hence, the heuristic provides a methodical approach to the access-tree-construction problem, which is also efficient in terms of power and performance.

VI. Early-Token-Return-Based MAN Construction

In this section, we present a very different approach to optimizing the MAN. The optimization goal here is to reduce the dynamic synchronization overhead that occurs between dependent memory accesses due to token transfer. Assume that we have two memory operations that are linked together by a token, as shown in Fig. 12(a). We must guarantee that the load is performed before the store. As described in Section IV, we enforce this by releasing the load access's token from the root of the access tree, which is then routed down to the store through the token tree, as shown in Fig. 12(b). The time spent waiting for the token to arrive at the store from the time the load is initiated (i.e., enters the access tree) is the cost of enforcing memory ordering, and we refer to this as the dynamic synchronization overhead.

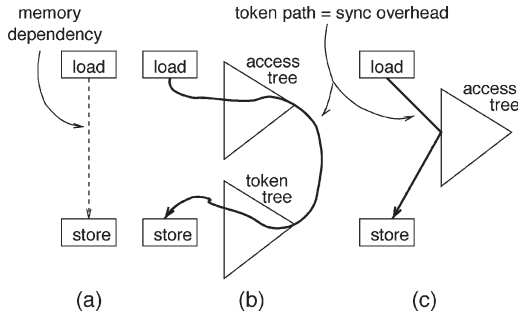


Fig. 12. (a) Input spec, (b) baseline MAN, and (c) improved MAN if we hook up both accesses to same leaf node in access tree.

A source of inefficiency in the basic MAN architecture is that memory ordering is enforced conservatively. An access releases its token only from the root of the access tree, thereby synchronizing with every other access in the application. However, it needs to synchronize only with its token dependents. In Fig. 12, for example, the store only has a single predecessor dependent, the load. Our architecture guarantees that the requests sent from a given tree node always arrive in order at the shared memory. If both these accesses are connected to the same leaf node in the access tree, then that leaf node becomes the merge point on their paths to memory. Therefore, the load can release the token to the store once it has reached the leaf node in the tree, as illustrated in Fig. 12(c). As is evident from the figure, we have drastically reduced the synchronization overhead[3] compared to Fig. 12(b). In this section, we present the ETR optimization, which identifies such opportunities and builds a customized MAN to take advantage of them.

A. Single-Entry Single-Exit (SESE) Regions

An important concept that enables this optimization is the SESE region [40]. 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 fan-out/fan-in of edges and 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 connect to the same subtree within the access tree, then the subtree root is the earliest common point on their paths to the memory. All accesses within the SESE region (except for the exit node) can release their tokens from this common point. The exit node must synchronize with its dependents outside the SESE region.

Dominator and postdominator trees of the input graph can be used to find SESE regions [40]. For our analysis, we define

[3]We also refer to this as token path or token round trip time (RTT).

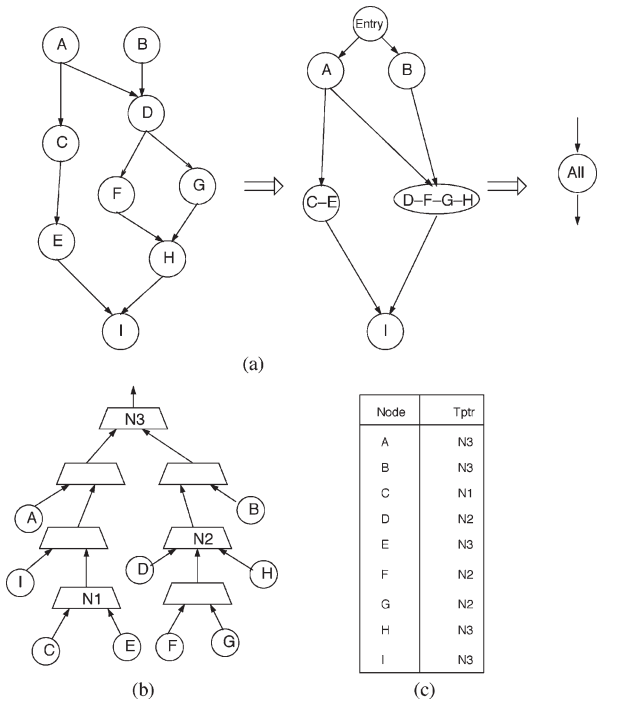
```
Reduce(G) {
    Add pseudo entry/exit nodes, if necessary
    Initalize Tptrs of all nodes in G to 0.

    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
        }
    }
}
```

Fig. 13. Pseudocode of Reduce algorithm.



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

Fig. 14. (a) Example of applying Reduce on graph. (b) Resulting memory-access tree. (c) Table on right shows Tptr pointers for each access specifying where in tree this access can release its token.

two special types of SESEs: 1) a linear SESE (LSESE) region is the maximal subgraph in which each node has exactly one predecessor and one successor and 2) an innermost SESE (ISESE) region is one that contains no other SESEs (including LSESEs) within it.

B. Reduce: Access-Tree-Construction Algorithm

Using SESEs, we have devised an algorithm, called Reduce, that generates an application-aware MAN for a given input graph G. Reduce is described in the pseudocode presented in Fig. 13. It requires G to have unique entry and exit nodes; if a graph has multiple source (sink) nodes, then a pseudonode entry (exit) can be created as a predecessor (successor) of all source (sink) nodes. In addition, each graph node is associated with a tree pointer (Tptr) indicating which tree node in the access tree releases the output token of that node. Initially, all Tptr pointers are NULL.

Next, we find all ISESE and LSESE regions in the graph and create a local subtree for the graph nodes in each of the identified regions. Creating a subtree for an LSESE region is easy since we know that they are all mutually exclusive. Hence, we can simply create an unarbitrated access tree node and connect all graph nodes in the LSESE to the inputs of the tree node. This forms our local subtree for the region. For an ISESE, we first construct a balanced arbitrated tree t using all the internal nodes of the region. Since the entry node, the root of t, and the exit node are all mutually exclusive, we connect these to an unarbitrated three-input access tree node, which forms the region's subtree root.

Once we form a subtree s for a given SESE, 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 X and alias $\text{Tptr}(\text{exit}(s)) = \text{Tptr}(X)$. This SESE reduction is repeated on the newly reduced graph. When the SESE region containing X is reduced, $\text{Tptr}(X)$ [and hence, $\text{exit}(s)$] will be updated.

The progression of Reduce is illustrated using an example token graph in Fig. 14(a). The initial token graph is progressively reduced, until a single node is left in the graph. Since the original graph itself is one large SESE region, Reduce is guaranteed to converge. The final access tree generated after reduce is shown in Fig. 14(b). The table in Fig. 14(c) displays the values of Tptr pointers corresponding to all accesses after Reduce completes. These values specify the node in the tree that will release the access's token. The complexity of a single iteration of Reduce is equivalent to the complexity of finding SESEs. The number of iterations depends on the input graph structure, but typically converges in less than five iterations.

C. Loop Optimizations

Fig. 15(a) shows the token graph for 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 the last iteration's access tokens.

Loop accesses present a unique opportunity because we know that between any two given loop iterations, the memory dependences are local to the loop region. However, a loop can never form an 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

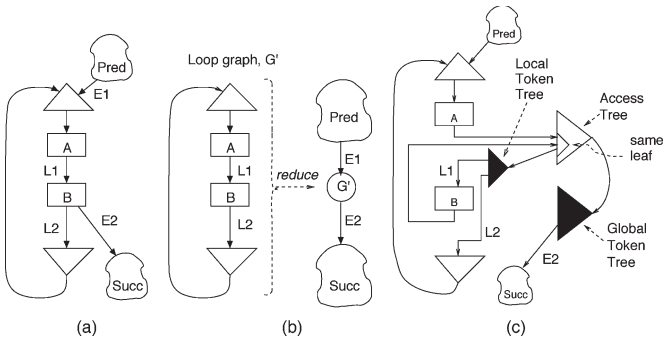


Fig. 15. Optimizing Reduce in loops. (a) Typical loop token graph. (b) Reducing loop graphs. (c) Global and local tokens for loop accesses; dark-colored triangles are token trees.

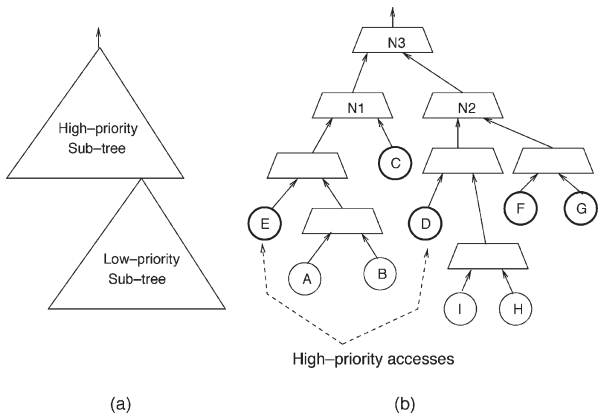


Fig. 16. (a) For two groups of high- and low-priority nodes, tree construction can be biased. (b) Result of applying this optimization on access tree generated in Fig. 14 to optimize high-priority accesses (bold-circled nodes).

into a new graph 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 an SESE region and optimize the resulting subtree independently of its external context.

Next, the fully reduced loop node is inserted into the top-level graph by reintroducing the deleted exit edges, as shown in Fig. 15(b). 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 access tree—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.

This is illustrated in Fig. 15(c), showing the simplified circuit for the synthesized loop. The dark-colored triangles are token trees through which all tokens are routed. A and B form an LSESE, and we would connect them to the same leaf of the access tree. A local token tree is now connected to this leaf that routes tokens from all accesses (A and B, in this case) to their respective destinations. Access B has a fan-out on its token, edges L2 and E2, the former staying within the loop, while the latter goes to Succ. Thus, L2 is deemed to be a local token that can be emitted from the local token tree, but E2 is emitted by the global token tree, since in the last iteration, B will synchronize with its successors in Succ. In this manner, loop accesses are accelerated through localized synchronization.

D. Priority-Based Construction

A problem with Reduce is that accesses in SESE regions reduced earlier on end up at the bottom of the access tree (for example, C–E and D–F–G–H in Fig. 14). 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 access tree. If the access is critical, then the 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. Fig. 16(a) illustrates this 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 Fig. 14(b), if we assign high priority to accesses C, E, D, F, and G, then the resulting access tree is shown in Fig. 16(b). Most importantly, this heuristic preserves the Tptr of the smaller subtrees involved in the construction. Hence, nodes C and E, for example, still point to $N1$ as the point of their token release, but have a shorter path through the AccessTree.

E. Experimental Evaluation

We incorporated ETR into CASH and synthesized the kernels from Table I using ETR. Fig. 17 shows the savings in synchronization overhead due to the deployment of ETR. Token (or value) RTT is defined as the time difference between when an access is inserted into the access tree and when its token (or load value) emerges from the token tree (or value tree). Using simulation traces, we computed the weighted average of the dynamic token and value RTTs across all accesses. The graph shows the ratio of these averages for the basic MAN versus an ETR MAN. A value greater than one implies that RTT is shorter after ETR.

By creating asymmetric trees, the ETR optimization reduces the token RTT (and hence, synchronization overhead) for some accesses, while increasing it for others. The use of priorities guides the optimization so that we reduce this overhead for the most common (i.e., high priority) case, thereby reducing the overall token RTT. Fig. 17 suggests that this strategy works quite well as the token RTT is reduced by a factor of about $4\times$, on average. The most illustrative cases are the adpcm kernels. These contain loops (and hence, SESE regions) with a single access within them; thus, their loop structure is similar to that in Fig. 15(a), but without the access A. This means that the subtree for each such loop consists of a single node, and the synchronization token for the access never needs to enter the

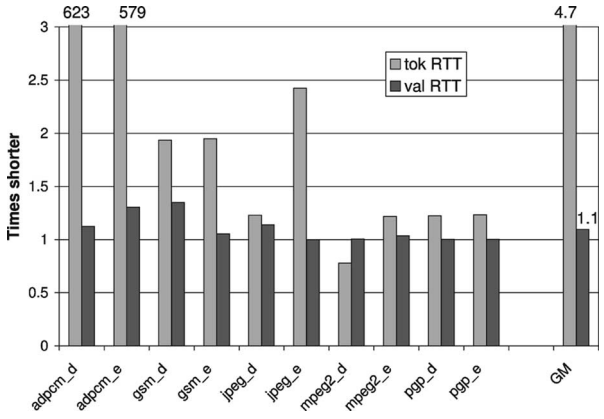


Fig. 17. Reduction in dynamic synchronization overhead (tok RTT) and load-value RTT.

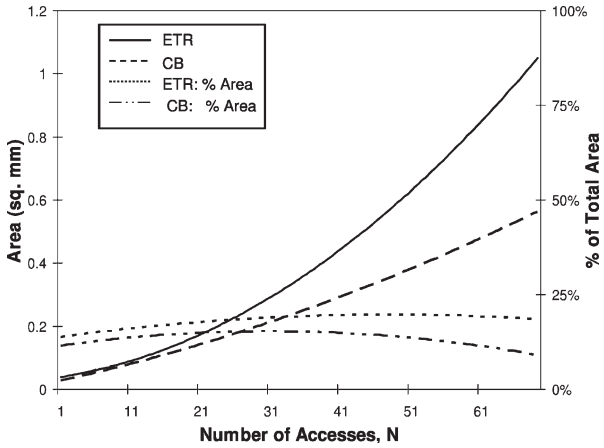


Fig. 18. Analysis of MAN area complexity. On primary y axis is growth in MAN area as N increases. On secondary y axis is fraction of MAN area to total circuit area in corresponding benchmark.

MAN. Hence, the token RTT is a single hop. For the same reason, the synchronization overhead increases in the mpeg2_d kernel. This kernel contains no loops. Hence, the strategy to improve the common case fails, and hence, the overall token RTT degrades.

VII. Results and Discussion

In this section, we evaluate the quality of the MAN. We have implemented the MAN construction and the optimizations within the CASH [8] toolflow, and all results presented in this section are for circuits synthesized from kernels in Table I. We start by discussing the MAN's cost overhead, followed by a performance evaluation of the MAN. Finally, we compare and evaluate the two optimizations presented in this paper and discuss the impact and opportunities of applying them in conjunction.

A. Cost Overhead

The MAN area consists of the area occupied by the access tree, the token tree, and the value tree. An analysis of the MAN area is depicted in Fig. 18. It shows two types of trends for the CB and ETR-based constructions. The trendlines labeled ETR and CB depict the growth in the absolute MAN area (square millimeters in a 180-nm process) as N, the number of accesses connected to the MAN increases. The ETR: % Area and the CB: % Area report the fraction of the total circuit area occupied by the MAN.

The make-up of the MAN area can be understood by studying Fig. 3. There are four hardware structures in the MAN—the access tree, the token tree, the value tree, and the memory station. The memory station is analogous to a load store queue (LSQ) in microprocessors. It provides the interface to the main memory while tracking outstanding load accesses. The design of the memory station is independent of the number of access N in the application, and thus has a constant size irrespective of the application. As N grows, the sizes of the access and token trees grow proportionately. The size of the value tree grows only if the number of `load` accesses in the application grows.

As N increases, the growth trends in the absolute MAN area (in Fig. 18) is almost linear for CB, and superlinear but subquadratic for ETR. The increased area cost for higher N comes from larger trees. The linear scalability of the MAN can be explained by its modularity and the absence of a centralized controller. Irrespective of the size of N, each tree in the MAN is built out of small composable modular building blocks. These building blocks are the tree-node implementations, as depicted in Figs. 4 and 5. As N increases, the depth of the MAN trees scale logarithmically, while the number of building blocks (and thus, the overall MAN area) scales linearly. Although the area complexity of the MAN scales linearly, the logic design complexity is constant and is equivalent to the design complexity of the individual building blocks. This is because of the distributed design of the MAN and the absence of any centralized controllers. Implementing these blocks with clockless (or self-timed) circuits eliminates the need for timing closure and retiming during the synthesis flow; this allows for: 1) arbitrary instantiation and interconnection of the various building blocks and 2) placement of these blocks as close to their producers/consumers as possible.

The difference in the absolute area between the ETR and CB MAN circuits can be explained by differences in the shape of the access tree. The area cost of each node in the access tree can be classified into: 1) the muxing cost; 2) the pipeline register; and 3) the handshake controller. From postlayout circuits, we have seen that the handshake circuitry accounts for only about 1% of the total area. Irrespective of the depth, if the access tree supports N accesses, it must contain some form of a cascaded $N \times 1$ multiplexer that is distributed throughout the tree. The biggest additional cost of a deeper tree is the additional pipeline registers. As these registers are usually about 70 bits wide, they are a significant contribution to the total area. For example, the register accounts for about 35% of the total area of a two-input access tree node. By reducing $F(k)$ in (6), CB effectively minimizes the latency through the tree by reducing its depth. On the other hand, by introducing early points of token synchronization, ETR tends to introduce additional tree levels. Further, with decoupled token regions, ETR also tends to have a higher token-tree area overhead.

The contribution of the MAN to the total area (the ETR: % Area and CB: % Area trendlines in Fig. 18) is application dependent. The trend in the graph shows that, for a variety of different applications with different N, the MAN contribution is a tolerable overhead, about 15% of the total area, with slight differences between ETR MAN and the CB MAN area.

B. Performance Comparison

To evaluate the quality of the MAN approach, it would be best to compare it against existing HLS approaches for handling dynamic memory dependences. However, as discussed in Section II, we know of no other HLS tool that supports dynamic memory disambiguation and that can tolerate unpredictable latency in the memory system. Given this difference, a fair comparison of our study with others is difficult. Further, the C benchmarks we use can be directly compiled in our system, while other flows impose severe restrictions on the C language; thus, these benchmarks must be considerably rewritten to make them synthesizable.

To provide the reader with a feeling for the overall effectiveness of the resulting circuits, we compare the performance of the automatically synthesized circuit from a given C program to that of a superscalar processor running the same program. The basis for this comparison is that: 1) both approaches use the same C source and 2) both approaches use a single-ported shared-memory resource, and the same external interface to the memory. Despite the fact that the superscalar is general purpose, the comparison is illustrative of two very different approaches to accessing the memory. The superscalar core is representative of all processor- and platform-based approaches to supporting memory accesses. It has a single point of access initiation, which is the LSQ, and all instructions are fetched and dispatched to the LSQ in program order. Accesses to this single initiation point are scheduled beforehand, and no contention can occur.

On the other hand, we implement the resulting ASIC as a spatial computing architecture [41]. The circuit is dynamically scheduled, and we allow for multiple concurrent accesses to be initiated simultaneously, and for contention to occur when accessing the memory interface. Thus, resolving contention while sustaining high levels of memory concurrence is the primary challenge in designing the MAN. The cost of performing a memory access in the MAN-based ASIC is thus much more expensive since contention must also be resolved.

For this comparison, we used MASE [42] to obtain simulation results for the superscalar core. The ASICs synthesized by CASH use a [180-nm/2-V] CMOS standard-cell technology. The core is assumed to run at 600 MHz, which is a reasonable clock frequency for an aggressive out-of-order core in the 180-nm process technology. Fig. 19 compares the execution of the given C kernels on the processor core versus execution on the synthesized ASICs. The graph shows the MAN's performance speedup over the processor core (a value greater than one implies speedup over the core). The RD bar corresponds to the average performance of the baseline MAN aggregated over all reference experiments (see Section V-D); the CB and ETR bars correspond to the performance of the CB and the early-token-based MAN construction.

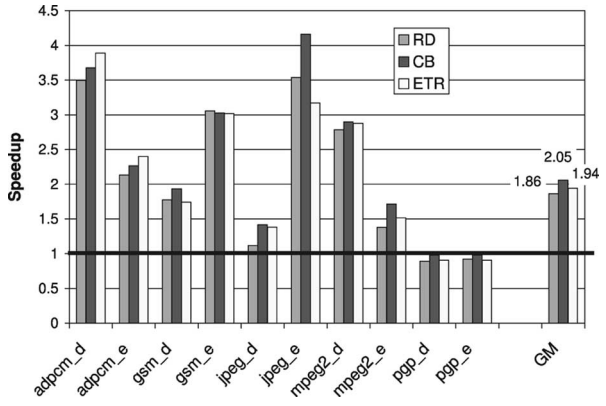


Fig. 19. Performance comparison of different MAN constructions against superscalar core (baseline).

The MAN-based ASIC constructions are superior in terms of execution on the aggressive superscalar core in almost all cases. The bottleneck due to contention arbitration can severely impact performance, as earlier limit studies have shown [41]. Despite this, our results show that an ASIC with an optimized MAN still delivers superior overall performance (on average, $2\times$ better than the core).

C. Optimization Evaluation

Comparison: Fig. 19 also shows that the CB MAN construction is generally superior to ETR. The main reason for this is that ETR only optimizes the token RTT paths in the circuit; it ignores the impact on the value RTT altogether. Further, the ETR optimization is highly sensitive to the priorities assigned to accesses, as described in Section VI-D. While a detailed compiler analysis can often statically predict criticalities, dynamic dependences lead to inaccuracies. The combination of these two factors results in some load accesses being demoted in the asymmetric tree. If priority information is inaccurate, then the performance deteriorates. On the other hand, CB optimizes the access-tree depth for all accesses. By reducing the depth of the access tree, it implicitly reduces the token and value RTT for all accesses. Further, it is much more resilient to inaccuracies in priority and criticality information than ETR. Hence, it can readily be applied without requiring complicated compiler analysis.

Returning tokens early implies an overhead in controlling the token return, as well as hurting other (demoted) accesses. It is thus necessary to justify this overhead, by saving a number of hops through the tree. This is also evidenced in our results, which show that ETR outperforms CB only in the cases of the adpcm kernels. As described in Section VI-E, ETR performs well in these cases because every loop in these kernels contains just a single access. Hence, as long as the loop executes, dynamic token RTT never enters the MAN, while in CB, the token still has to travel through the MAN. Hence, ETR optimizes the most common case, and the overhead is justified.

Combination: Since the goals of these two optimizations are vastly different, it is difficult to apply both in conjunction.

While CB relies exclusively on the construction of a balanced access tree, ETR always generates an asymmetric access tree. Despite this, we attempted to apply the two in conjunction: We built the access tree for these kernels using CB, and looked for LSESE chains (see Section VI-A) that are connected to the same leaf. For each LSESE chain, all accesses except the exit of the LSESE can return their tokens early to their destinations without having to route the token through the MAN. Fig. 12 is an example that depicts this scenario. However, our experiments show little or no performance benefit over a purely CB construction due to two reasons: 1) the dynamic critical path of the application rarely pass through the token paths that are optimized by ETR and 2) this still adds circuit overhead at the access-tree root, which must now decide when it should forward a token to the token tree. If the optimized token paths are not critical, then this overhead will increase the latency for all unoptimized accesses, resulting in an overall performance degradation.

When the source application is large, however, the access tree will have more levels. Most commonly, large applications consist of regions of accesses where most synchronization occurs within a region, whereas inter-region synchronization is less common. For example, our experiments have shown MAN construction for a single kernel. When synthesizing a multikernel application, each kernel will form such a region. Thus, the MAN within each kernel can be constructed using CB, while synchronization between kernels, which will be much fewer and less common, can be optimized using ETR. Hence, we foresee CB as a fine-grained optimization and ETR as a coarser grained optimization.

A final observation is that in a large application, the MAN needs to be spread out across a large chip area, and the wires between the tree nodes begin to dominate the performance. Binary trees naturally pipeline these wires. This same trend is true also on a smaller scale when looking at future technologies where wire loads are predicted to dominate over gate delays [43]. Since the number of hops through binary trees are larger, ETR has better potential, even for a relatively small number of access points.

VIII. Conclusion

We have presented SOMA, a framework for synthesizing and optimizing unconstrained memory accesses in HLS. Starting from a graph representation that specifies may dependences, we synthesize a distributed MAN, which implements communication to and from the memory. The architecture is scalable and provides a dynamic synchronization mechanism that maintains consistency in the context of memory-ordering dependences that are known only at runtime.

We also present two optimizations of the baseline MAN construction. The first uses a CB analysis of the application to select the MAN topology that is tailored to the application's memory concurrence profile. The second optimization explores other MAN topologies that reduce dynamic synchronization overhead. This is done by identifying local regions of memory dependences and tailoring the MAN topology to take advantage of these regions.

While the MAN described here is synthesized as a clockless circuit, there is 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 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.

References

[1] OCP-IP Association, *The Importance of Sockets in SoC Design*. White paper downloadable. [Online]. Avalable: www.ocpip.org

[2] *International Technology Roadmap for Semiconductors (ITRS)*. (2003). [Online]. Available: http://public.itrs.net/Files/2003ITRS/SysDrivers200.pdf

[3] Arvind, R. Nikhil, D. Rosenband, and N. Dave, "High-level synthesis: An essential ingredient for designing complex ASICS," in *Proc. IEEE/ACM ICCAD*, San Jose, CA, 2004, pp. 775–782. [Online]. Available: http://csg.csail.mit.edu/pubs/memos/ Memo-473/memo473.pdf

[4] M. Hind and A. Pioli, "Evaluating the effectiveness of pointer alias analyses," *Sci. Comput. Program.*, vol. 39, no. 1, pp. 31–55, Jan. 2001. [Online]. Available: http://www.elsevier.nl/gej-ng/10/39/21/43/20/22/abstract.html; http://www.elsevier.nl/gej-ng/10/39/21/43/20/22/article.pdf

[5] W. Wang, T. K. Tan, J. Luo, Y. Fei, L. Shang, K. S. Vallerio, L. Zhong, A. Raghunathan, and N. K. Jha, "A comprehensive high-level synthesis system for control-flow intensive behaviors," in *Proc. 13th ACM GLSVLSI*, Washington, DC, 2003, pp. 11–14.

[6] G. Mittal, D. C. Zaretsky, X. Tang, and P. Banerjee, "Automatic translation of software binaries onto FPGAs," in *Proc. 41st Annu. DAC*, San Diego, CA, 2004, pp. 389–394.

[7] S.Y. Liao, "Towards a new standard for system level design," in *Proc. 8th CODES*, San Diego, CA, 2000, pp. 2–6.

[8] G. Venkataramani, M. Budiu, T. Chelcea, and S. C. Goldstein, "C to asynchronous dataflow circuits: An end-to-end toolflow," in *Proc. IWLS*, Temecula, CA, Jun. 2004, pp. 501–508, (full paper). [Online]. Available: http://www.cs.cmu.edu/~mihaib/research/iwls04.pdf

[9] I. M. Verbauwhede, C. J. Scheers, and J. M. Rabaey, "Memory estimation for high level synthesis," in *Proc. 31st Annu. DAC*, San Diego, CA, 1994, pp. 143–148.

[10] S. Y. Ohm, F. J. Kurdahi, N. Dutt, and M. Xu, "A comprehensive estimation technique for high-level synthesis," in *Proc. 8th ISSS*, Cannes, France, 1995, pp. 122–127.

[11] Y. Zhao and S. Malik, "Exact memory size estimation for array computations without loop unrolling," in *Proc. 36th ACM/IEEE DAC*, New Orleans, LA, 1999, pp. 811–816.

[12] P. G. Kjeldsberg, F. Catthoor, and E. J. Aas, "Storage requirement estimation for optimized design of data intensive applications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 9, no. 2, pp. 133–158, Apr. 2004.

[13] M. Luthra, S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Interface synthesis using memory mapping for an FPGA platform," in *Proc. Int. Conf. Computer Design*, San Jose, CA, 2003, pp. 140–145.

[14] L. Séméria, K. Sato, and G. De Micheli, "Synthesis of hardware models in C with pointers and complex data structures," *IEEE Trans. VLSI Syst.*, vol. 9, no. 6, pp. 743–756, Dec. 2001. [Online]. Available: http://azur.stanford.edu/~lucs/paper/TVLSI00/semeria tvlsi.pdf

[15] D. J. Kolson, A. Nicolau, and N. Dutt, "Integrating program transformations in the memory-based synthesis of image and video algorithms," in *Proc. IEEE/ACM ICCAD*, San Jose, CA, 1994, pp. 27–30.

[16] G. Stitt, Z. Guo, W. Najjar, and F. Vahid, "Techniques for synthesizing binaries to an advanced register/memory structure," in *Proc. ACM/SIGDA 13th Int. Symp. FPGA*, Monterey, CA, 2005, pp. 118–124.

[17] G. Corre, E. Senn, P. Bomel, N. Julien, and E. Martin, "Memory accesses management during high level synthesis," in *Proc. 2nd IEEE/ACM/IFIP CODES+ISSS*, Stockholm, Sweden, 2004, pp. 42–47.

[18] P. Gupta and A. C. Parker, "Smash: A program for scheduling memory intensive application-specific hardware," in *Proc. 7th ISSS*, Niagara, ON, Canada, 1994, pp. 54–59.

[19] J. Park and P. C. Diniz, "Synthesis of pipelined memory access controllers for streamed data applications on FPGA-based computing engines," in *Proc. 14th ISSS*, Montreal, QC, Canada, 2001, pp. 221–226.

[20] C.-G. Lyuh and T. Kim, "Memory access scheduling and binding considering energy minimization in multi-bank memory systems," in *Proc. 41st Annu. DAC*, San Diego, CA, 2004, pp. 81–86.

[21] J. Seo, T. Kim, and P. R. Panda, "An integrated algorithm for memory allocation and assignment in high-level synthesis," in *Proc. 39th DAC*, New Orleans, LA, 2002, pp. 608–611.
[22] C Level Design, *C2HDL*. [Online]. Available: http://www.cleveldesign.com/
[23] CoWare, *N2C*. [Online]. Available: http://www.coware.com/
[24] Frontier Design, *A—rt Builder*. [Online]. Available: http://www.frontierd.com/
[25] K. Wakabayashi, "C-based synthesis experiences with a behavior synthesizer, cyber," in *Proc. Conf. DATE*, Munich, Germany, 1999, pp. 390–393.
[26] T. Kambe, A. Yamada, K. Nishida, K. Okada, M. Ohnishi, A. Kay, P. Boca, V. Zammit, and T. Nomura, "A c-based synthesis system, bach, and its application, (invited talk)," in *Proc. ASP-DAC*, Yokohama, Japan, 2001, pp. 151–155.
[27] A. Ghosh, J. Kunkel, and S. Liao, "Hardware synthesis from C/C++," in *Proc. Conf. DATE*, Munich, Germany, 1999, pp. 387–389.
[28] M. Luthra, S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Interface synthesis using memory mapping for an FPGA platform," Univ. California, Irvine, Tech. Rep. 03-20, Jun. 2003.
[29] C. Huang, S. Ravi, A. Raghunathan, and N. K. Jha, "High-level synthesis of distributed logic-memory architectures," in *Proc. IEEE/ACM ICCAD*, San Jose, CA, 2002, pp. 564–571.
[30] T. J. Callahan and J. Wawrzynek, "Adapting software pipelining for reconfigurable computing," in *Proc. Int. Conf. CASES*, Grenoble, France, 2000, pp. 57–64.
[31] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: A high-level synthesis framework for applying parallelizing compiler transformations," in *Proc. VLSI*, New Delhi, India, Jan. 2003, pp. 461–466. [Online]. Available: http://www.cecs.uci.edu/~spark/ pubs/Spark-System-Vlsi03.pdf
[32] R. P. Wilson and M. Lam, "Efficient context-sensitive pointer analysis for C programs," in *Proc. SIGPLAN Conf. Program Language Design and Implementation*, La Jolla, CA, 1995, pp. 1–12. [Online]. Available: http://suif.stanford.edu/papers/wilson95/paper.html
[33] M. Budiu and S. C. Goldstein, "Optimizing memory accesses for spatial computation," in *Proc. Int. ACM/IEEE Symp. CGO*, San Francisco, CA, Mar. 23–26, 2003, pp. 216–227. [Online]. Available: http: //www-2.cs.cmu.edu/~mihaib/research/cgo03.pdf
[34] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill, "Dependence flow graphs: An algebraic approach to program dependencies," in *Proc. POPL*, Orlando, FL, 1991, vol. 18, pp. 67–78. [Online]. Available: http://iss.cs.cornell.edu/Publications/Papers/popl.91
[35] A. M. G. Peeters, "Single-rail handshake circuits," Ph.D. dissertation, Dept. Comput. Sci., Eindhoven Univ. Technol., Eindhoven, The Netherlands, Jun. 1996.
[36] I. Sutherland, "Micropipelines: Turing award lecture," *Commun. ACM*, vol. 32, no. 6, pp. 720–738, Jun. 1989. [Online]. Available: http:// www.acm.org/pubs/citations/journals/cacm/1989-32-6/p720-sutherland
[37] A. Bystrov, D. J. Kinniment, and A. Yakovlev, "Priority arbiters," in *Proc. 6th Int. Symp. ASYNC*, Eilat, Israel, 2000, pp. 128–137.
[38] T. Bjerregaard and J. Sparsø, "A scheduling discipline for latency and bandwidth guarantees in asynchronous network-on-chip," in *Proc. 11th IEEE Int. Symp. Advanced Research Asynchronous Circuits Systems*, New York, 2005, pp. 34–43.
[39] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. 30th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Research Triangle Park, NC, 1997, pp. 330–335. [Online]. Available: http://www.icsl.ucla.edu/~billms/Publications/mediabench.ps
[40] R. Johnson, D. Pearson, and K. Pingali, "The program structure tree: Computing control regions in linear time," in *Proc. SIGPLAN Conf. PLDI*, Orlando, FL, Jun. 1994, pp. 171–185.
[41] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein, "Spatial computation," in *Proc. Int. Conf. ASPLOS*, Boston, MA, pp. 14–26, Oct. 2004. [Online]. Available: http: //www.cs.cmu.edu/~mihaib/research/asplos04.pdf
[42] E. Larson, S. Chatterjee, and T. Austin, "MASE: A novel architecture for detailed microarchitectural modeling," in *Proc. IEEE ISPASS*, Tucson, AZ, Nov. 4–6, 2001, pp. 1–9.
[43] D. Sylvester and K. Keutzer, "A global wiring paradigm for deep submicron design," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 19, no. 2, pp. 242–252, Feb. 2000.

Girish Venkataramani (S'00) received the undergraduate degree in information systems from Birla Institute of Technology and Science (B.I.T.S.), Pilani, India, in 1998, and the M.S. degree in computer science from University of California, Riverside, in 2001. He is currently working toward the Ph.D. degree in electrical and computer engineering at Carnegie Mellon University, Pittsburgh, PA.

Prior to pursuing graduate studies, he worked as a Software Engineer at Aztec Software, India, Philips Innovation Systems, India, and Philips Research, The Netherlands. Since starting graduate school, he has interned at Microsoft Research and at the Embedded Systems Lab., National University of Singapore. His research interests include the direct implementation in hardware of programs written in very high-level languages like ANSI-C. He is specifically examining an asynchronous implementation of these circuits and is interested in building a framework of support tools that can automatically optimize the performance and power characteristics of these circuits.

Tobias Bjerregaard (M'00) received the M.S. degree in electrical engineering with specialization in microelectronics and the Ph.D. degree from the Technical University of Denmark (DTU), Kgs. Lyngby, Denmark, in 2000 and 2006, respectively. The topic of his Ph.D. thesis was clockless circuit design for on-chip networks.

From 2000 to 2002, he was part of the fabless startup IC design company IP Semiconductors, doing back-end, custom/standard cell logic integration, and floor planning for a network processor. He has served as a Student Member of the Ph.D. Program Committee at his university and as a Ph.D. Representative on the board of his department. He is currently a Post-Doctoral Researcher at DTU, with plans to start a company based on his thesis research. His research interests are global communication and timing in system-on-chip designs.

Tiberiu Chelcea (S'99–M'04) received the B.S. and M.S. degrees from Politehnica University of Bucharest, Romania, in 1995 and 1996, respectively, and the Ph.D. degree in computer science from Columbia University, New York, in 2004.

Currently, he is a Postdoctoral Fellow in the Computer Science Department at Carnegie Mellon University, Pittsburgh, PA. His research interests include asynchronous circuits, design techniques for mixed-timing digital systems, and synthesis methods from high-level languages down to asynchronous systems.

Seth C. Goldstein (S'92–M'96) received the undergraduate degree from the Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ, and the Ph.D. degree in computer science at the University of California (UC), Berkeley, in 1997.

Currently, he is an Associate Professor in the School of Computer Science at Carnegie Mellon University, Pittsburgh, PA. Before attending UC Berkeley, he was CEO and Founder of Complete Computer Corporation, which developed and marketed object-oriented programming tools. His research interests include using nanotechnology in computer science. He is involved in two main efforts: the first, Phoenix, involves compilers and architectures for novel computing systems with a focus on reconfigurable computing, and his second effort, Claytronics, involves realizing programmable matter.