Managing Dynamic Reconfiguration Overhead in Systems-on-a-Chip Design Using Reconfigurable Datapaths and Optimized Interconnection Networks

Zhining Huang, Sharad Malik
Electrical Engineering Department
Princeton University

Abstract

This research examines the role of dynamically reconfigurable logic in systems-on-a-chip (SOC) design. Specifically we study the overhead of storing and downloading the configuration code bits for different parts of an application in a dynamically reconfigurable coprocessor environment. For SOC designs the different configuration bit-streams will likely need to be stored on chip, thus it becomes crucial to reduce the storage overhead. In addition, reducing the reconfiguration time overhead is crucial in realizing performance benefits. This study provides insight into the granularity of the reconfigurable logic that is appropriate for the SOC context. Our initial study is in the domain of multimedia and communication systems. We first present profiling results for these using the MESCAL compiler infrastructure. These results are used to derive an architecture template that consists of dynamically reconfigurable datapaths using coarse grain logic blocks and a reconfigurable interconnection network. We justify this template based on the constraints of SOC design. We then describe a design flow where we start from an application, derive the kernel loops via profiling and then map the application using the dynamically reconfigurable datapath and the simplest interconnection network. As part of this flow we have developed a mapping algorithm that minimizes the size of the interconnection network, and thus the overhead of reconfiguration, which is key for systems-on-a-chip. We provide some initial results that validate our approach.

1. Introduction

Recent research on reconfigurable computing has shown that a tightly coupled reconfigurable co-processor with a general purpose CPU can achieve significant speedup on a general class of applications [2]. However, the hardware resource constraints associated with the reconfigurable logic are a major barrier in applying this to large applications.

One way to overcome the hardware resource constraint limitation is to use dynamic reconfiguration which can reconfigure the logic at run time. This partitions the application temporally and time multiplexes the programmable logic to meet the hardware resource constraints[1,9,11]. However, dynamic reconfiguration comes with its own problems. This is the reconfiguration overhead, both in time and space. Since the reconfiguration is done at run time, the reconfiguration time is part of the run time overhead. Also, multiple reconfiguration bitstreams need to be stored for the different contexts being multiplexed onto the programmable logic. This problem is exacerbated for SOC implementations where the entire application needs to be stored in on-chip memory. In this paper we propose a solution to this overhead problem by providing a design style, methodology and algorithm that minimizes the configuration size overhead. This proposal uses the idea of distributed caches [1] to reduce the configuration time overhead.

2. Architectural Model

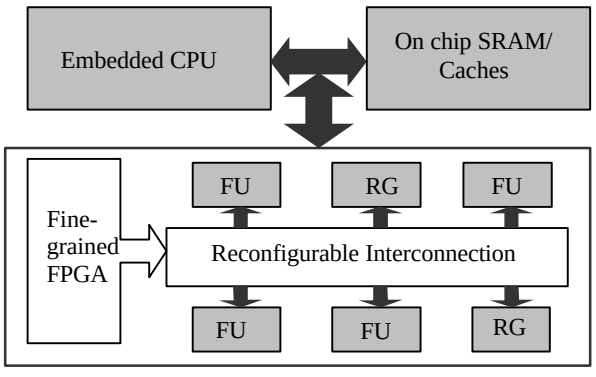

Figure 1. Architecture template

The architecture template for our methodology is shown in Figure 1. The key element of this is to use coarse grain blocks such as functional units, register files and memories, in a reconfigurable datapath constructed using a programmable interconnection network. This is similar to templates used in other reconfigurable computing projects such as Pleiades [3,4]. The motivation for using the coarser grained logic is to reduce the configuration bit overhead, since now only the interconnection network needs to be programmed.

For a given application, we partition the software into two parts, one is to be executed as software on the general-purpose processor, and the rest mapped to the reconfigurable coprocessor. The general-purpose processor consists of an embedded CPU and on-chip L1 cache. The coprocessor consists of fixed logic blocks (Function Units, FU), Registers (RG) and fine grained programmable logic (FPGA). The datapath of the kernel loops is mapped onto the logic blocks and configurable interconnections. The fine-grained FPGA is used to implement control logic to generate the control signals for the datapath. We would like to exploit the same logic macros (FUs, RGs) across different parts (kernel loops) of the application by just reconfiguring the interconnection and control logic on fine grained FPGA.

3. Design Methodology Flow

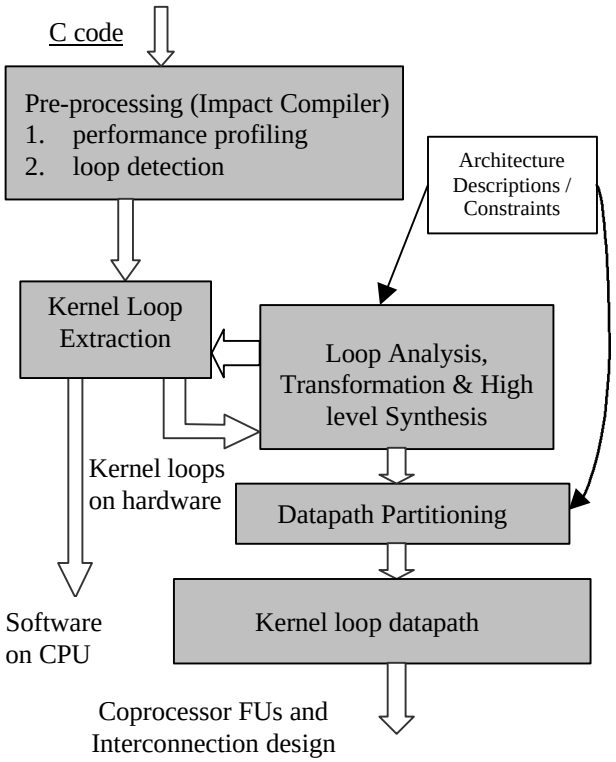

Figure 2. Design methodology flow

The open elements of the design in the architectural template described above are the logic blocks and design of the interconnection network. This is designed specific to an application family. The methodology flow for this is shown in Figure 2. The input to the design is C source code for an application or a set of applications from a domain.

First we use the *IMPACT* [10,7] compiler , which is part of the MESCAL compiler infrastructure, as the front end to do some pre-processing, including performance profiling and loop detection. The preprocessing is done on scheduled and register allocated intermediate code (IMPACT lcode). The performance profiling information used includes loop invocation counts, loop iteration counts, and loop execution time in clock cycles.

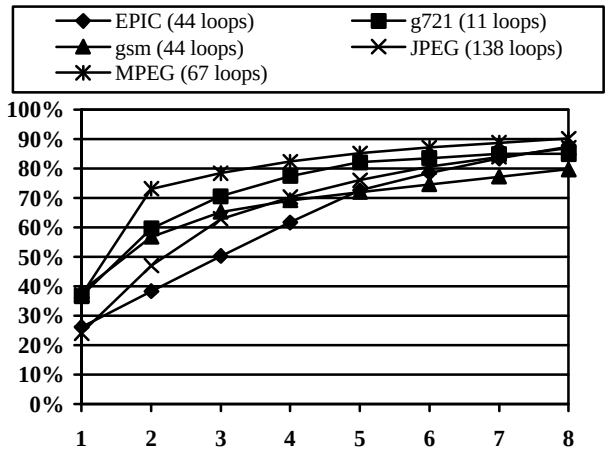

Figure 3. Percentage execution time vs. number of top kernel loops

The performance profiling is used for kernel loop extraction. The goal is to select kernel loops which have the highest execution time. Profiling results on Media Bench [6] in Figure 3 show that few kernel loops usually occupy most of the execution time of an application. Across these benchmark applications we see that no more than eight kernel loops together account for more than 80% of the total execution time. This fact provides the rationale for mapping only the top few (<9) kernel loops onto the reconfigurable coprocessor. However, not all the top execution count loops are selected as kernel loops to be mapped to coprocessor. Those loops which are too large in terms of total computation and will overflow the reconfigurable logic hardware resource constraint during the transformation and synthesis step are not used. Other top loops are chosen to make up for those not included.

The loop transformation and high level synthesis step generates the hardware version of the chosen kernel loops. Currently we do not use any loop unrolling or other inter-iteration parallelism transformations. We primarily

use high-level synthesis to exploit intra-iteration parallelism. This is used to get the best hardware with least number of execution cycles within acceptable hardware cost. Currently we use the NEC Cyber tool to do the loop transformations and high level synthesis [5]. The input is C like source code. The output is VHDL code of best hardware. The hardware constraint in this step corresponds to the maximum number of function units and registers we can have in the coprocessor.

Datapath partitioning partitions the hardware of each kernel loop into the datapath and the control circuit. The datapath contains all the function units and signals (wires, registers) which have the bit-width of 32 bits or 64 bits. The control signal usually is 1 bit wide. The control circuits are configured using a fine-grained FPGA. The datapath part is mapped to the logic blocks and configurable interconnections.

4. Datapath Mapping

The design of the configurable datapath is done using the hardware generated for each kernel loop using Cyber. The mapping of the hardware to the coprocessor is done using a graph formulation that minimizes the size of the interconnection network.

The first step in this process is to generate a datapath topology graph G, for each kernel loop hardware, as shown in Figure 4. Vertices in the directed graph G correspond to the hardware blocks, i.e. the function units and registers in datapath. Each vertex is labeled with its block type. The edges correspond to the interconnections between the function units and registers.

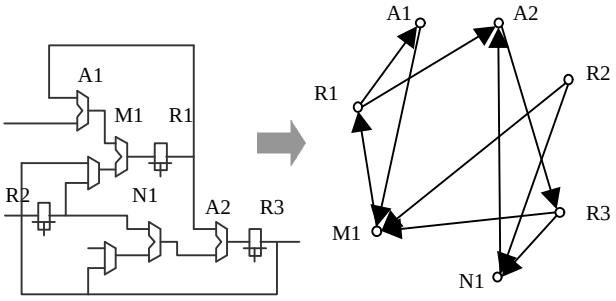

Figure 4. A directed graph generate from datapath circuit

For each kernel loop i for which we have generated a hardware circuit, we generate a topology graph G_i. We then generate the coprocessor design as well as the mapping of the loops one by one. Let G1 and G2 be two graphs generated. The design of the final coprocessor and interconnection network is done by constructing a graph G as follows:

- Create a graph G (with labeled vertices) such that, G1⊆G, G2⊆G, and the cost of G is least;
- The cost of G is the total number of edges in G. Since edges represent interconnections and we need to minimize the number of interconnections in the final interconnection network.

The above formulation results in a datapath corresponding to G, which has the datapath for G1 and G2 embedded in it. Also, the mapping of vertices of G1 and G2 to vertices of G is done so as to minimize the number of interconnections in G.

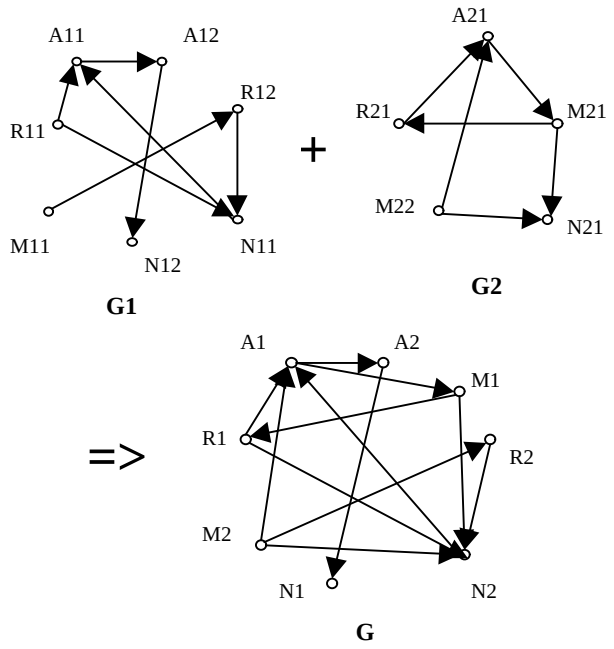

Figure 5. Graph mapping

From the above we can see that the number of logic macros chosen for a type is the maximum number among all kernel loops for that type. That means the number of each type vertices in graph G is decided by the maximum number of that type vertices among all the graphs G_i. This allows for the maximum parallelism for each kernel loop in the final coprocessor, while still permitting sharing to logic across the kernel loops. Clearly adding any more vertices (blocks of any type) cannot increase parallelism for any kernel loop. We further show that this cannot even reduce the number of edges. This is proved as follows:

Assume n is the maximum number of A-type (adder) vertices in all graphs G_i and we have n+1 A-type vertices in G. For each graph G_i, if vertex n+1 is used, at least 1 A-type vertex in A-type vertices 1 to n has not been used. If we re-map the A-type vertex n+1 to that free vertex, no more edges are added and we can possibly reduce the number of edges further. Thus, adding more vertices cannot reduce the number of edges.

The problem of finding the least cost G and the mapping of vertices of G1 and G2 to vertices of G is solved using maximum bipartite mapping.

A short review of maximum bipartite matching is provided here and illustrated in Figure 6 (a) [8].

A graph G=(V,E), is said to be bipartite if the vertex set can be partitioned into V=L∪R, where L and R are disjoint and all edges go between a vertex in L and a vertex in R.

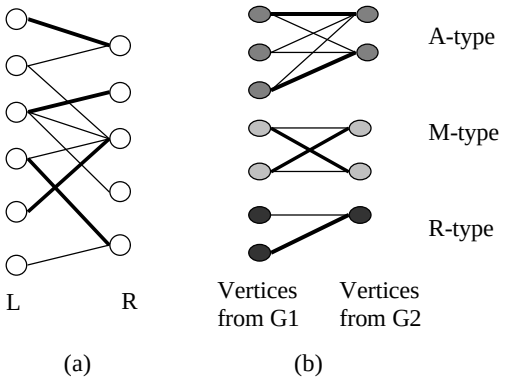

Figure 6. Bipartite matching

A matching is a subset of edges M⊆E such that for all vertices v∈V, at most one edges of M is incident on v. A maximum matching is the matching with maximum total edge weight of the subset.

In our datapath mapping problem here we construct a graph G12 as follows. (This is illustrated in Figure 6 (b)). Vertices of subset L are from G1, vertices of subset R are from G2. Edges only go between vertices from L and vertices from R. Edges here mean mapping. An edge connects vertex v1 from L and v2 from R, if V1 and V2 can be mapped onto the same vertex in G. Since the function units and registers can only be mapped to the same type, edges here only go between the same type of vertices from G1 and G2.

The weight of edge (v1, v2) refers to the number of edges that go into v1 in graph G1 and edges go into v2 in G2 which come from the same type of vertices. If v1 and v2 were mapped to the same vertex in G, then these edges could be shared in G. The purpose of the edge definition here is to search the maximum similarity between the two graphs G1 and G2 in terms of interconnections between logic blocks.

Maximum weight bipartite matching is used as a heuristic here to reduce the number of connections in G. If (v1, v2) is in the matching, then v1 in G1 and v2 in G2 are mapped to the same vertex in G.

For more than two graphs, we iteratively combine them into a single graph, adding one new graph at a time.

5. Designing the Interconnection Network

There are two options for designing the interconnections for a given coprocessor. One is to have full crossbar between the function units we have chosen enabling any logic block to be connected to any other. The other is to select the interconnections that will actually be used in the kernel loop hardware reconfigurations. A full crossbar, while providing flexibility for the future, has the problem of occupying too large an area on the chip and more configuration bits and control signals for the reconfigurable coprocessor. Assume that the datapath width is 32bits and we have 40 function units and registers, we will have 40*40=1600 32-bit width wires running on the chip, which is excessive. The second option can be implemented using the datapath mapping algorithm outlined in the previous section. That technique already shows us how to minimize the datapath connections (edges in graph G) for the chosen kernel loops for the given set of applications. Given graph G, the interconnections can be made at run time such that G implements either G1 or G2.

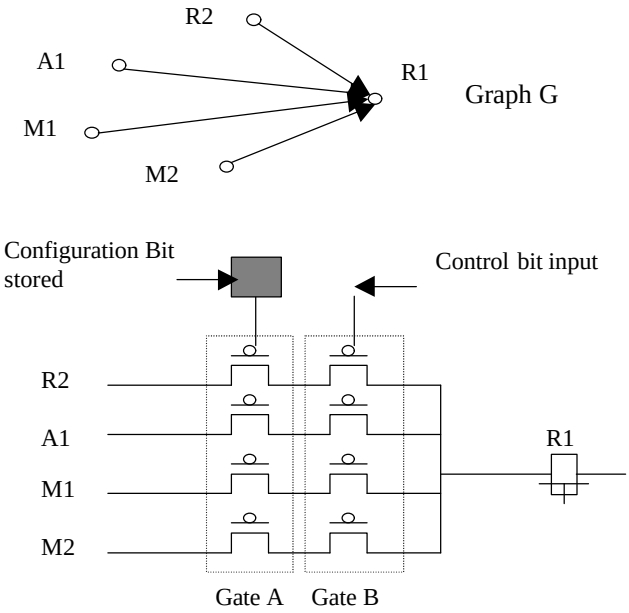

Figure 7. Interconnection network

Figure 7 shows the interconnection network between function units and registers. If an edge exists in the graph G, a hardware connection needs to connect the 2 vertices, i.e, the function units or registers. Figure 7 shows part of graph G and interconnection network as example.

Two n-type transmission gates (or complementary gates) are used for each interconnection. One gate is for configuration and another one is for mux control. So one configuration bit is needed for each edge in graph G, as well as a one-bit control signal. While the coprocessor is

configured for kernel loop i, if the edge in graph G also exists in graph Gi the configuration gate on the interconnection according to this edge will be configured on. Otherwise it is off. The mux control signals are from the control circuit which is configured on fine grained FPGA. The on or off of the mux control gate will be changed during the kernel loop at run time while the configuration gates remain fixed and will be only changed while the coprocessor is configured for another kernel loop.

As discussed above, the major reconfiguration overhead is the time for downloading the reconfiguration bits from memory to the coprocessor configuration gates. One way to solve this problem is to use distributed cache as suggested in some recent research. We now describe its use in our context.

6. Distributed Cache for Reconfiguration

We can use distributed cache to pre-store the configuration bits for use in dynamic re-configuration. Figure 8 showed the architecture of the distributed cache used for dynamic reconfiguration. For example, in the NEC version of the distributed cache [1], up to 8 configuration contexts can be pre-stored in the cache, with a single cycle context-switch. This minimizes the reconfiguration time to its bare minimum. Thus, if we choose to map no more that 8 kernel loops to the co-processor, we need no more than 8 contexts for dynamic reconfiguration. In Section 1 we showed how for the MediaBench examples, 80% of the execution time was spent in no more than 8 kernels. This enables us to load the 8 contexts prior to the execution of the application, with no subsequent overhead during run time.

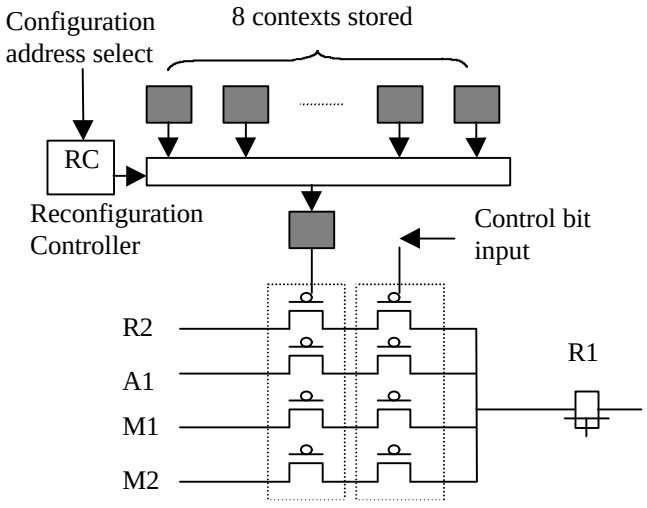

Figure 8. Distributed cache for reconfiguration

7. ADPCM

Our initial study on this project involves examination of multimedia and communication systems. ADPCM is one of the benchmarks we began with. ADPCM stands for Adaptive Differential Pulse Code Modulation. It is a family of speech compression and decompression algorithms.

After preprocessing using the IMPACT compiler as front end and loop extraction, two kernel loops are extracted from ADPCM source file, one for the ADPCM coder and another one for the ADPCM decoder. The 2 kernel loops occupy 99% of the total running time of ADPCM as determined from the profiling result. Two graphs G1 and G2 are generated from the hardware version of the kernel loops in VHDL files from the CYBER behavioral synthesis tool. The mapping result of graphs G1 and G2 are listed in Table 1.

Table 1. ADPCM datapath mapping result

	Coder (G1)	Decoder (G2)	Mapping result (G)
Vertices	45	32	46
Edges	133	79	165

From Table 1 we see that almost all of the vertices of G1 can be reused in G2, only one additional vertex is needed to map G2. Similarly, more than half of the connections in G1 can be shared between G1 and G2. Since there are 165 edges in graph G, we need 165 interconnections between those 46 logic macros, thus 165 configuration bits are need for interconnection, compared with 46*46=2116 bits and 2116 interconnections for a full crossbar interconnection between the logic macros.

The speed up gained from hardware execution can be significant. In the ADPCM coder application, the kernel loop of the coder is executed 1000*147 times as per the profiling result. The software execution time of each iteration is 66 cycles under the assumption of ideal memory access (no cache miss). The hardware execution time can be as fast as 20 cycles per iteration. Assuming the use of the distributed cache for context switch, the reconfiguration time and entry initialization take only a few cycles, far less than the run time for each configuration, i.e, 1000*66=66,000 cycles. Since the kernel loops occupies 99% of the run time of ADPCM, the total running time can be reduced from 10M cycles to 3M cycles in the best case. From the example of ADPCM, it is obvious that maximum gain from hardware execution can be achieved once the reconfiguration overhead has been reduced significantly. Note that in the above we are only utilizing intra-iteration concurrency in the kernel loops. Inter-iteration concurrency can lead to further speedup and is the subject of our current extensions.

8. Conclusions

This paper describes an efficient approach for dynamic reconfiguration to reduce the reconfiguration overhead in both size and time for dynamic reconfiguration. This is especially important in systems on a chip where the reconfiguration bits are likely to be stored on chip. We propose the use of a coarse-grained reconfigurable coprocessor to reduce the size of the configuration bit stream.

We then show how the programmable interconnection network can be minimized using a mapping algorithm that maps the hardware of individual kernel loops onto a single coprocessor.

We then go on to show how a distributed cache with a small number of contexts can be used to switch between the different kernel loops with very low overhead. We use profiling data to show that the small number of contexts is sufficient in practice.

Finally, we illustrate our methodology using ADPCM as a case study. The mapping results for ADPCM validate the efficacy of our approach.

9. References

[1] K. Furuta, T. Fujii, M. Motomura, K. Wakabayashi, M. Yamashina "Spatial-Temporal Mapping of Real Applications on a Dynamically Reconfigurable Logic Engine (DRLE) LSI", CICC 2000.

[2] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, J. Stockwoo "Hardware-Software Co-Design of Embedded Reconfigurable Architectures" in Proceedings, 37th Design Automation Conference (DAC 2000), June 2000.

[3] M. Wan, H. Zhang, V. George, M. Benes, A. Abnous, V. Prabhu, J. Rabaey "Design Methodology of a Low-Energy Reconfigurable Single-Chip DSP System" Journal of VLSI Signal Processing, 2000.

[4] Hui Zhang, Marlene Wan, Varghese George, Jan Rabaey "Interconnect Architecture Exploration for Low Energy Reconfigurable Single-Chip DSPs" Proceedings of the WVLSI , Orlando, FL, USA, April 1999.

[5] K. Wakabayashi, “Cyber: High Level Synthesis System from Software into ASIC,” in High Level VLSI Synthesis, edited by R. Camposano and W. Wolf, Kluwer Academic Publisher, pp.127-151, 1991.

[6] C. Lee, M. Potkonjak, W. Mangione-Smith, “MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems”, Micro 30, 1997.

[7] IMPACT research group, University of Illinois, at Urbana-Champaign, http://www.crhc.uiuc.edu/IMPACT/.

[8] T. Cormen, C. Leiserson, R. Rivest, “Introduction to Algorithms”, The MIT Press, 1990, pp. 600-603.

[9] D. C. Cronquist, P. Franklin, S. G. Berg, C. Ebeling "Specifying and Compiling Applications for RaPiD" FCCM 1998.

[10] P. Chang, S. Mahlke, W. W. Hwu “Using Profile Information to Assist Classic Compiler Code Optimizations “Software Practice and Experience, Dec. 1991, Vol. 21, No. 12, pp. 1301-1321.

[11] Rahul Razdan, Karl Brace, and Michael D. Smith "PRISC Software Acceleration Techniques" Proc. 1994 IEEE Intl. Conf. on Computer Design, pp. 145-149, October 1994.