Appeared in Proceedings of the 12th International Conference on Field Programmable Logic and Applications, FPL 2002

Compiling Application-Specific Hardware

Mihai Budiu and Seth Copen Goldstein

Carnegie Mellon University
{mihaib,seth}@cs.cmu.edu

Abstract. In this paper we describe ASH, an architectural framework for implementing Application-Specific Hardware. ASH is based on automatic hardware synthesis from high-level languages. The generated circuits use only localized computation structures; in consequence, we expect these circuits to be fast, to use little power and to scale well with program complexity.

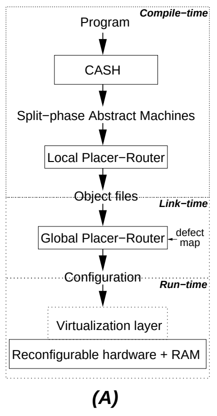
We present in detail CASH, a scalable compiler framework for ASH, which generates hardware from programs written in C. Our compiler exploits instruction level parallelism by using aggressive speculation and dynamic scheduling. Based on this compilation scheme, we evaluate the computational resources necessary for implementing complex integer-based programs, and we suggest architectural features that would support the ASH framework.

1 Introduction

For five decades the relentless pace of technology, expressed as Moore's law, has supplied computer architects with ample materials in their quest for high performance. The abundance of resources has translated into increased complexity. This complexity has already become unmanageable in several respects:

- Verification and testing costs escalate dramatically.
- Manufacturing costs grow dramatically with each new hardware generation.
- Defect density control gets more expensive as the feature size shrinks; in the near future we will be unable to manufacture large defect-free integrated circuits.
- The clock frequency has increased to a point where only a small fraction of the chip is reachable in a single cycle.
- The number of exceptions generated by the CAD tools requiring manual interventions grows quickly with design complexity.
- The dissipated power density (watts/mm^2) of state-of-the-art microprocessors reaches values that make air-cooling infeasible.
- Today's processors use extremely complicated hardware structures to enable the exploitation of the instruction-level parallelism (ILP) in large windows; however, the sustained performance is rather low.

Under the assumption that hardware density continues to improve at an exponential pace for the next decade, we propose in Section 2 an alternative approach to implement general-purpose computation, which consists of synthesizing — at compile time — application-specific hardware, on a reconfigurable-hardware substrate. We argue that such hardware can solve or alleviate all of the above problems. We call this model **ASH**, for Application-Specific Hardware. We propose a method for directly synthesizing custom, application-specific dataflow machines in hardware. ASH implementations



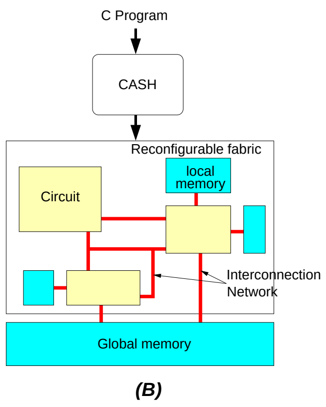


Fig. 1. *(A) The ASH tool-flow (B) Translation of programs into hardware.*

have low overhead, as they are precisely tailored to the program parallelism. ASH circuits can be used stand-alone to implement the whole application, or in tandem with a general-purpose processor. The main component of the ASH framework is **CASH**, a Compiler for ASH, presented in Section 3. CASH spans the realms of traditional compilation and hardware synthesis.

In Section 4 we evaluate the hardware resources needed to implement realistic programs within the ASH model of computation. Section 5 describes some implications of the ASH architecture on computer system design.

2 Application-Specific Hardware

In this section we give an overview of the ASH model of computation. The core of ASH is a reconfigurable fabric; compilation subsumes the role of traditional software compilation and hardware synthesis, translating high-level language programs into hardware configurations.

The left of Figure 1 summarizes our framework. Programs written in general-purpose high-level languages are the input to the CASH compiler.

From each procedure in the program, CASH constructs three different types of objects: computation structures, interconnection links and local memories (see Figure 1, right). In this paper we address only the construction of the hardware circuits.

Each procedure is independently optimized, synthesized, placed, and routed. The pre-placed and routed circuits for each procedure are then connected together in a global place and route phase. The resulting “executable” is a configuration for a reconfigurable hardware platform.

The procedures communicate asynchronously with each other. Each contains computation and possibly a small local memory. All the internal signals of the procedure have predictable latency, including the access to local memory. Procedures can however invoke remote operations, which have unpredictable latencies.

Whenever a procedure needs to execute an operation that has unpredictable latency it uses the interconnection network: remote memory accesses, and control-flow transfers are conceptually transformed into messages which can be routed dynamically on a network.

During program execution a procedure can be in one of three states: (1) Inactive: if is not being executed, does not have live state, and need not consume power; (2) Active: if is actively switching, being at the "top" of the stack; (3) Passive: if stores live values, but is blocked waiting for the completion of a callee (occasionally there may be some concurrent execution between a caller and a callee).

An example. Here we illustrate, using an example, the atomic operations and how they are assembled together to implement a simple C program. The program is an iterative implementation of the Fibonacci function, displayed in the left side of Figure 2; its ASH implementation is given in the right side.

CASH partitions each C procedure into a collection of *hyperblocks*, transforms each hyperblock into straight-line code using *speculation* and next translates each hyperblock into a dataflow circuit. The compilation process is discussed further in Section 3. Once a hyperblock starts execution, every one of its operations is executed exactly once.

In order to understand how ASH circuits operate, one should think of the data as produced by a source operator and consumed by a destination operator. Once the data is consumed, it is no longer available. In general, an operator is *strict*, i.e. cannot compute unless all its input data items are present. An operation may fanout the data value it produces to multiple consumers.

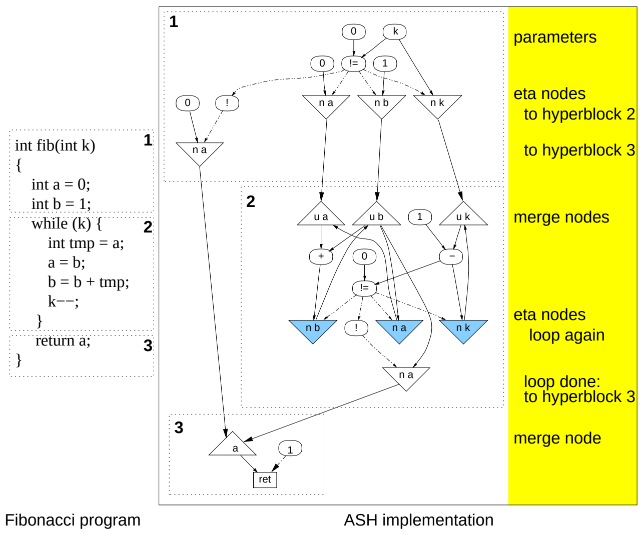
Dataflow circuits can be easily used to express straight-line code. In order to allow the implementation of control-flow constructs (branches, procedure calls), ASH augments the set of dataflow operations with two special constructs: *merge* and *eta* nodes[1]. These nodes are used between hyperblocks. Merge and eta nodes are sufficient for synthesizing circuits corresponding to arbitrary flow of control, including that of irreducible graphs. Merge is denoted by a triangle pointing upwards, while eta (η) is a triangle pointing downwards.

The *eta* operation has one data input, one *predicate* input and one data output. If the predicate is *true*, the input data is copied to the output, otherwise the input data is just consumed and no output is generated. Thus, an eta node is a gateway, which lets data flow to a different part of the circuit depending on the predicate. For instance, the eta nodes in hyperblock 1 will steer data to either hyperblock 2 or 3, depending on the test `k != 0`. Note that the etas going to hyperblock 2 are controlled by this predicate, while the eta going to hyperblock 3 is controlled by its complement.

Merge is the only non-strict operator. It has n inputs and one output; it copies one available data input to the output. A merge node accepts information from multiple sources, but only one of the sources should be active at some point. There are merge nodes in hyperblocks 2 and 3. The merge nodes in hyperblock 2 accept data either from hyperblock 1 or from the back-edges in hyperblock 2 itself. The back-edges denote the flow of data along the *while* loop. The merge node in hyperblock 3 can accept control either from hyperblock 1 or from hyperblock 2. The constant "1" feeding the "return" instruction is a predicate, showing that the return is (from a control-flow point of view) unconditionally executed, i.e., it fires as soon as its input data is available.

A formal definition of the semantics of all the basic ASH constructs can be found in [4]. This type of operational semantics, where data is explicitly produced and con-

[1] This terminology is historical, borrowed from the datafbw machine literature [18].



```
int fib(int k)                1
{
   int a = 0;
   int b = 1;
   while (k) {                2
      int tmp = a;
      a = b;
      b = b + tmp;
      k−−;
   }
   return a;                  3
}
```

Fig. 2. *C program for Fibonacci and its ASH implementation. The program is decomposed by CASH into 3 hyperblocks, depicted by dotted lines. Hyperblocks are described in Section 3.*

sumed, is also used in some types of asynchronous circuit descriptions and in dataflow machine architectures.

3 CASH

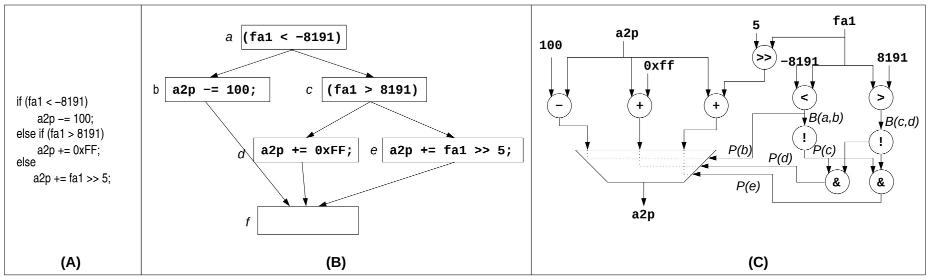
In this section we describe CASH, our current implementation of the ASH Compiler. Our compiler infrastructure is built around the SUIF 1.3 research compiler [20]. Currently, we do not use any of the parallelizing components of SUIF. Due to space restrictions this presentation is very abstract; for more details please refer to [4].

Hyperblocks. The main abstraction we use at the program level is the hyperblock. A hyperblock is part of a program control-flow graph (CFG) with a single entry point but possibly multiple exits. Hyperblocks have been introduced in the context of predicated execution [12] to uncover instruction-level parallelism (ILP) by removing the control dependences through the predication of the instructions within the hyperblock.

Using hyperblocks as a unit of compilation for reconfigurable hardware was earlier proposed by Callahan and Wawrzynek [6]. Their proposal was heavily influenced by resource constraints and selected only high-profile loop body fragments to map to hardware. Our method of hyperblock selection is unconstrained by resource limitations. We cover each procedure with disjoint hyperblocks, using a linear-time algorithm.

Work in predicated architectures developed a set of heuristics for hyperblock selection [12] (considering hyperblock fragmentation, code duplication, loop peeling and other optimizations). Currently we are using a simple heuristic, building hyperblocks of maximal size.

Some hyperblocks will be loop bodies; we follow the same approach as Callahan and consider the back-edges as part of the hyperblock. If a loop has multiple back-edges, all of them are considered part of the same hyperblock. We next synthesize each



```
if (fa1 < −8191)
    a2p −= 100;
else if (fa1 > 8191)
    a2p += 0xFF;
else
    a2p += fa1 >> 5;
```

Fig. 3. *(A) Code fragment (B) Control-flow graph (C) Speculative implementation.*

hyperblock separately; later merge and eta nodes are used to connect hyperblocks to each other.

Hyperblock synthesis. Through the use of predication, each hyperblock is transformed into straight-line code. Next the code is brought into static-single assignment form (SSA) [9] through variable renaming. The computation of the combinatorial portion of each hyperblock (i.e. excluding the back-edges) is next implemented speculatively in the style of predicated static-single assignment [8] and predicated speculative execution [1], as described in [6].

To illustrate the implementation, we use the example in Figure 3, a code snippet from the `g721` Mediabench [10] program.

Path Predicates. Each basic block in a hyperblock has an associated *path predicate*, as described in [8]; the path predicate associated to block B is true if and only if block B is executed during the current loop iteration. The predicates corresponding to blocks are recursively defined: $P(entry) = True$ and $P(s) = \vee_{p \in Pred(s)}(P(p) \wedge B(p,s))$, where $B(p,s)$ is true if block p branches to s. This is read as: "Block s is executed if and only if one of its predecessors p is executed and p jumps to s."

We next use *instruction promotion* [12] which removes predicates from some instructions or replaces them with weaker predicates, enabling their speculative execution. If the hyperblock code is in static-single assignment form, any instruction with no side-effects can be safely and completely promoted to be executed unconditionally.

Since predicate computations do not need to be guarded, they can be implemented like any regular computation. The predicates are then used for three tasks: (1) to guard the execution of instructions with side-effects (memory writes, memory reads that can trigger exceptions, function calls and returns), (2) to control looping, and (3) on exit from the current hyperblock to indicate which successor hyperblock is executed.

Each edge in the CFG contributes one term to the predicate computation, so the implementation of all predicates is linear in the hyperblock size.

The speculative program implementation just described is essentially a gated static single assignment representation (GSA) [14] of the predicated program. The ϕ operators used by the SSA form become in our representation multiplexors, selecting among the many definitions of a value that reach a join point in the CFG. The multiplexor selectors are path predicates. Unlike other proposed SSA representations for predicated code [8], we explicitly build the circuitry to compute the ϕ functions, which become multiplexors in hardware (see Figure 3C).

Multiplexor placement and optimization. As already noted, the placement of multiplexors corresponds to the placement of ϕ functions in SSA form. However, multiplexor placement is simpler than ϕ placement, because all the back edges in a hyperblock go to the entry point and the rest of the hyperblock is a directed acyclic graph.

We next run a multiplexor simplification pass, which repeatedly applies one of the following rules: (1) constant selector predicates values can be removed; (2) multiple identical data inputs of a mux are merged into a single input and the predicate is set to the logical "or" of the corresponding predicates (3) a mux with a single data input is removed and the input is connected directly to the output (4) two chained mux are transformed into a single mux (whose predicates are the logical "and" of the two muxes).

Merge and eta insertion. The circuits generated for the hyperblocks are "stitched" together using merge and eta operations as follows: (1) for each live variable at the entry of a hyperblock we create a merge node; (2) for each live variable at an exit of a hyperblock (i.e., on a hyperblock exit edge) we create an eta node; the eta is controlled by the edge predicate. The eta's output is connected to the input of the corresponding merge node of the successor hyperblock.

Scheduling and synchronization. Crucial for performance is the efficient scheduling of the dataflow operations. Here we depart from Callahan's proposal, by implementing dynamic scheduling using a completely distributed synchronization scheme. The difference between these two methods is analogous to the difference between VLIW and superscalar processors. Static scheduling requires little hardware support (in the form of a very simple sequencer, which is implemented as a circular shift register in Garp), while dynamic scheduling requires a more complicated handshaking protocol.

The producer of data must signal that data is valid, while the consumer(s) must signal that they have extracted the data, i.e., that the channel can be reused. This protocol is essentially the Two-Phase Bundled Data convention used in asynchronous hardware implementations. ASH is perfectly suitable for an asynchronous implementation.

We expect that mixed implementations that combine static and dynamic scheduling are feasible: portions of the computation between unpredictable latency operations can use simple sequencers that are started by a "data valid" signal. "Data valid" signals are also used to preserve the original program order between instructions which have side effects. Notice that these signals ensure that operations are issued in the original program order; they do not specify the order in which they will complete.

Lenient evaluation. One problem of predicated-execution architectures is that execution time on speculated control-flow paths may be unbalanced [2]. For instance, assume that subtraction takes much longer than addition; then the leftmost path in Figure 3C is the critical path. We propose to solve this problem by using *lenient*, *fully decoded multiplexors*.

Fully-decoded multiplexors have as many selector bits as there are inputs. Each selector bit selects one of the inputs, as shown by the dotted lines in Figure 3C. These multiplexors do not need complicated decoding logic. A lenient multiplexor can generate its output as soon as one selector predicate is *true* and the corresponding selected data item is valid. We use lenient evaluation both for boolean operations and multiplexors.

			Units				Bit-operations			
Program	LOC	Circuits	tokens	memory	call/ret	fp	predicates	mux	fbw	arithmetic
adpcm_e	183	6	12	10	8	0	53	1,408	1,749	4,450
adpcm_d	183	6	7	9	8	0	36	832	1,653	2,114
g721_e	924	39	115	148	86	0	421	2,816	8,673	25,488
g721_d	922	46	113	153	108	0	493	3,200	8,464	30,741
gsm_e	4156	247	761	1,080	463	0	1,951	15,104	52,707	291,065
gsm_d	4155	243	751	1,069	459	0	1,966	14,976	51,865	290,552
epic_e	1632	254	323	379	231	53	1,389	8,928	252,613	124,164
epic_d	1538	127	310	474	112	6	139	864	175,386	69,670
mpeg2_e	5123	553	2,490	2,482	876	197	6,714	37,152	205,309	430,367
mpeg2_d	6854	457	1,297	1,714	837	11	3,820	18,816	102,395	254,513
jpeg_e	15623	1,705	3,605	7,429	1,676	153	11,624	42,016	702,402	701,323
jpeg_d	15039	1,651	3,386	6,993	1,605	153	11,099	38,848	695,943	713,640
pegwit_e	5013	320	1,008	1,711	604	0	2,412	8,832	69,231	194,744
pegwit_d	5013	320	1,008	1,711	604	0	2,412	8,832	69,231	194,744
mesa	47063	3,299	10,491	19,742	4,104	4,900	125,938	175,008	2,024,765	2,032,393
129.compress	1431	84	216	241	82	3	214	1,760	19,007	24,794
124.m88ksim	12910	750	4,225	4,951	2,080	47	25,650	50,208	178,035	373,232
099.go	25665	2,229	11,324	9,516	2,966	0	35,889	139,360	961,999	817,972
130.li	4888	653	2,000	2,251	1,672	13	4,480	30,560	87,436	101,028
132.ijpeg	17563	1,658	3,609	7,632	1,854	157	10,182	29,216	649,388	741,338
134.perl	23365	1,960	26,835	14,143	4,924	47	140,598	191,040	1,045,737	858,553
147.vortex	49224	1,165	17,389	20,054	7,665	4	53,324	147,744	693,659	966,205

Table 1. *Static resource consumption for each benchmark. Some resources are expressed in units, while other are expressed in bit-operations.* **LOC:** *lines of source code,* **hypers:** *number of hyperblocks generated;* **tokens:** *token-merging operators;* **memory:** *loads and stores;* **call/ret:** *calls and returns;* **fp:** *fbating-point operations;* **predicates:** *boolean operations computing predicates;* **mux:** *multiplexors;* **flow:** *control-fbw operators;* **arithmetic:** *integer arithmetic. We do not include the cost of handshaking control circuitry.*

4 Resources Required for ASH Implementations

In this section we present a preliminary evaluation of the required resources for the complete implementation of programs in hardware. We analyze a set of programs from the Mediabench [10] and SpecInt95 [17] benchmark suites.

Resources. Table 1 displays the resources required for the complete implementation of these programs in hardware. We do not include in these numbers the standard library or the operating system kernel. All the values are static counts. The lines of code were counted with the sloccount program [19], which skips whitespace and comments. We have used a conservative approximation of the program call-graph to eliminate some of the procedures which are never called.

For some of the operations it is fairly easy to estimate the required hardware resources; we listed these under the heading “bits”, and the values indicate the approximate number of bit-operations required to implement them. For remote operations (memory access, call/return), the implementation size can vary substantially, depending for instance on the nature of the interconnection network. For these we report just operation counts.

Comments: The raw computation resources required (the total of the “bits” columns) is below 2.2 million for all benchmarks except `mesa`, which is below 5 million (without any floating-point resources). Even by today’s standards, these are reasonably small and many of them can be implemented completely in hardware now—the rest will soon fit within a single chip.

This data doesn't include the savings that can be achieved by implementing computations of custom sizes. Research has shown [5] that simple static methods can eliminate 20% of the bit computations in these benchmarks.

Notice that the resources taken by the predicate computations are minor compared to the actual computation; this suggests that coarse-grained reconfigurable fabrics are more suitable for ASH systems than today's fine-grained FPGAs.

5 Benefits of the ASH Model

The ASH model has better scalability properties than traditional CPU architectures. For instance:

- The verification and testing of a homogeneous reconfigurable fabric is much simpler than general purpose microprocessors. ASH translates the applications directly into hardware, so there is no interpretation layer (i.e., the CPU) which can contain bugs. By using the translation validation [15], (used by a certifying compiler to emit a formal proof that the executable is equivalent with the input program) we completely eliminate one complex layer needing verification and testing.
- Only one hyperblock is actively switching at any time, requiring little power.
- ASH implementations use only local signals, which scale well with clock frequency. All inter-procedural communication can be made using a switched, pipelined interconnection network, so there is no need for global electrical signals.
- Dynamic methods of extracting ILP from programs (as implemented in today's out-of-order processors) are hindered by limited issue windows. Our compiler analyzes large program fragments and can uncover substantially more parallelism than processor issue windows.

The main disadvantage of the ASH paradigm is the requirement for substantial hardware resources. However, this can be alleviated through use of virtualization, or by hardware-software partitioning between a CPU and an ASH fabric. According to data presented in Section 4, the evolution of component density according to Moore's law will soon provide sufficient resources for all but the most complex programs.

6 Related Work

This work has two different lineages: research on intermediate program representation and compilation for reconfigurable hardware architectures.

Many researchers have addressed the problem of compiling high-level languages for reconfigurable architectures: e.g., [16, 6, 3, 13, 11]. Our compilation scheme is most closely related to the scheme proposed by Callahan in the Garp compiler [6, 7]. While we exploit many of the ideas in his proposal, we differ in the following respects:

- Our approach reflects our different assumptions about the amount of available computational resources. In his proposal each hyperblock becomes a separate configuration. In our implementation we translate entire procedures into hardware: each procedure is decomposed into a collection of disjoint hyperblocks, and inter-hyperblock communication is synthesized. We also handle procedure calls and returns in hardware, albeit with some restrictions (currently we do not handle recursion).

- In Callahan's work, the synthesized hyperblock implementation is statically scheduled, using a fixed sequencer. We propose the use of a dynamically scheduled execution, which, despite requiring potentially more hardware resources to implement, has the capability to gracefully absorb unpredictable latency operations. Our dynamic scheduling scheme naturally generalizes his software-pipelining scheme.

The output of our compiler is a series of circuits. These bear a striking resemblance to some forms of intermediate representations of the program in other optimizing compilers. Our circuits are most closely related to predicated static-single assignment [8]. Our circuits are also closely related to dataflow machines [18], but are meant to be implemented directly in hardware and not interpreted on dataflow machines using token-passing.

7 Conclusions

In this paper we have presented a proposal for a new model of computation, called Application-Specific Hardware (ASH), which implements programs completely in hardware, on top of a reconfigurable hardware platform. Our preliminary evaluations suggest that soon there will be enough hardware resources to accommodate complete realistic programs, and that the sustained performance of this model will be comparable to processor-based computations.

We have discussed the compilation technology which can scalably translate large programs written in high-level languages into hardware implementations. Our compilation strategy transforms hyperblocks into circuits which execute many operations speculatively, and thus expose a substantial amount of instruction-level parallelism. The execution of the hardware is dynamically scheduled by using only local synchronization structures, tolerating unpredictable latency events.

We have also outlined those features of the ASH model of computation that promise to make this model *scalable*. ASH implementations can easily and naturally take advantage of the exponentially increasing amount of hardware resources, avoiding many of the problems that the increased complexity brings to standard CMOS-based microprocessor design and manufacturing.

8 Acknowledgements

This research is funded in part by the National Science Foundation under Grant No. CCR-9876248 and by Darpa under contract #2156-CMU-ONR-0659. We are grateful to Tim Callahan for his comments on a preliminary version of this paper.

References

1. David I. August, Daniel A. Connors, Scott A. Mahlke, John W. Sias, Kevin M. Crozier, Ben-Chung Cheng, Patrick R. Eaton, Qudus B. Olaniran, and Wen mei W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 227–237, June 1998.

2. David I. August, Wen mei W. Hwu, and Scott A. Mahlke. A framework for balancing control fbw and predication. In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.
3. Jonathan Babb, Martin Rinard, Csaba Andras Moritz, Walter Lee, Matthew Frank Rajeev Barua, and Saman Amarasinghe. Parallelizing applications into silicon. In *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999.
4. Mihai Budiu and Seth Copen Goldstein. Pegasus: An efficient intermediate representation. Technical Report CMU-CS-02-107, Carnegie Mellon University, May 2002.
5. Mihai Budiu, Majd Sakr, Kip Walker, and Seth Copen Goldstein. BitValue inference: Detecting and exploiting narrow bitwidth computations. In *Proceedings of the 2000 Europar Conference*, volume 1900 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.
6. Timothy J. Callahan and John Wawrzynek. Instruction level parallelism for reconfigurable computing. In Hartenstein and Keevallik, editors, *FPL'98, Field-Programmable Logic and Applications, 8th International Workshop, Tallinin, Estonia*, volume 1482 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1998.
7. Timothy J. Callahan and John Wawrzynek. Adapting software pipelining for reconfigurable computing. In *Proceedings International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2000*, 2000.
8. Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Jeanne Ferrante. Path analysis and renaming for predicated instruction scheduling. *International Journal of Parallel Programming, special issue*, 28(6), 2000.
9. R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.
10. Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Micro-30, 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, 1997.
11. Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure, and Jon Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In *DAC 2000*, 2000.
12. Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, Dec 1992.
13. Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart memories: A modular reconfigurable architecture. In *Proceeding of the International Conference on Computer Architecture 2000*, June 2000.
14. Karl J. Ottenstein, Robert A. Ballance, and Arthur B. Maccabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the Conference on Programming Language Design and Implementation PLDI 1990*, pages 257–271, 1990.
15. Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In Springer Verlag, editor, *Proceedings of TACAS'98*, volume 1384 of *LNCS*, pages 151–166, 1998.
16. Rahul Razdan. *PRISC: Programmable reduced instruction set computers*. PhD thesis, Harvard University, May 1994.
17. Standard Performance Evaluation Corp. *SPEC CPU95 Benchmark Suite*, 1995.
18. Arthur H. Veen. Datafbw machine architecture. *ACM Computing Surveys*, 18 (4):365–396, 1986.
19. David A. Wheeler. More than a gigabuck: Estimating GNU/Linux's size. http://www.dwheeler.com/sloc, November 2001.
20. Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In *ACM SIGPLAN Notices*, volume 29, pages 31–37, December 1994.