C to Asynchronous Dataflow Circuits: An End-to-End Toolflow

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

Abstract

We present a complete toolflow that translates ANSI-C programs into asynchronous circuits. The toolflow is built around a compiler that converts C into a functional dataflow intermediate representation, exposing instruction-level, pipeline and memory parallelism. The compiler performs optimizations and converts the intermediate representation into pipelined asynchronous circuits, with no centralized controllers. In the resulting circuits, control is distributed, communication is achieved through local wires, and arbitration for datapath resources is unnecessary. Circuits automatically synthesized from Mediabench kernels exhibit excellent energy-delay.

1 Introduction

For five decades Moore's law has supplied chip designers with an exponentially increasing amount of computational resources. Computer architects have taken advantage of these additional resources to produce faster and more parallel machines. But this relentless advance is proving to be a double-edged sword: the chip complexity is also exponentially out-pacing the design productivity [1], and designers are unable to efficiently take advantage of the wealth of available resources. In this paper we describe a solution to the design scalability problem: A CAD tool flow which can automatically synthesize energy efficient asynchronous circuits directly from ANSI-C programs. Our compiler automatically creates Application-Specific Hardware (ASH) circuits directly from C programs, generating highly distributed computational structures which require no global wires and no centralized controllers. ASH uses the extra resources to reduce design time and manage complexity while providing excellent energy efficiency and energy-delay.

The important characteristics of our toolflow are: (1) fully automatic compilation of C programs to dataflow machines; (2) hardware synthesis of the dataflow machines as asynchronous circuits; (3) synthesis of dynamically-scheduled circuits that directly implement the source program, without resource sharing or the use of interpretive structures. ASH circuits can most naturally be used in combination with a regular processor: the processor handles tasks such as running the operating system and virtual memory, while ASH is best suited for implementing computations with a high amount of instruction-level parallelism (ILP). The ASH circuits can either be fabricated as an ASIC or can be converted into a configuration and loaded onto a reconfigurable fabric that is coupled with the processor.

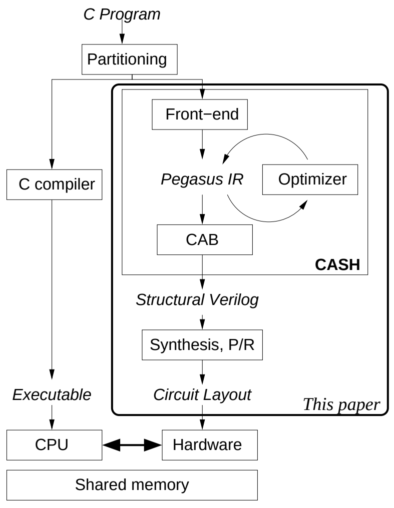

Figure 1: Compilation toolflow

Figure 1 shows how our compiler, CASH (Compiler for ASH), is used within a complete compilation/synthesis tool-flow.
(1) Hardware-software partitioning, which selects the C kernels to be mapped on ASH, is performed at procedure boundaries: entire procedures are mapped either to the processor or to ASH as an indivisible unit [11]. The partitioner can automatically synthesize the communication code for transferring control across the hardware-software interface.
(2) The software part of the program is mapped to the processor by a regular C compiler; the hardware part is processed by the CASH front-end and the CASH Asynchronous Back-end (CAB).

(2a) The CASH front-end translates the selected C kernels into a dataflow intermediate representation (IR) called Pegasus [9], and then performs a complex series of optimizations.

(2b) CAB produces asynchronous circuits from Pegasus; the output is a structural Verilog description of the circuits. Commercial tools then perform synthesis, and place-and-route.

1.1 Contributions

A distinguishing characteristic of our methodology is the distribution of control in a dynamically scheduled, purely dataflow circuit. Aggressive predication and speculation is used to increase parallelism and all control-flow constructs are transformed into datafbw. The resulting circuits contain no FSMs or global control circuits.

The tool-fbw compiles unmodified ANSI C programs into hardware, without requiring any hardware-related program annotations. This enables us (1) to leverage the enormous base of existing code, and (2) to directly compile the reference descriptions frequently used for describing hardware, which are customarily translated by hand into Verilog. The result is a fully automated CAD toolfbw that converts a C program into circuit layout.

We envision our toolfbw being useful in two scenarios: first, as a compiler for accelerating the performance of a hybrid microprocessor-reconfigurable hardware system, at a low energy cost. This avenue becomes even more important as technology continues to shrink feature sizes. Second, our tool can also be used to deliver a quick turnaround, low-power ASIC solution. In this regard, our research is similar to the "chip-in-a-day" project from Berkeley [17], but our approach is very different: that project starts from parallel statechart descriptions and employs sophisticated hard macros to synthesize synchronous designs with automatic clock gating. In contrast we start from C, use a library of standard cells, and build asynchronous circuits.

Our toolfbw also explores the use of established synchronous CAD tools for synthesizing asynchronous circuits. While using these tools did not produce optimal results for our asynchronous designs (see Section 6), the overall experience is very encouraging. In contrast, most of the proposed asynchronous synthesis fbws employ custom tools to synthesize their circuits.

1.2 Roadmap

In the remainder of this paper we focus our attention on the hardware compilation issues, shown in the highlighted box in Figure 1. The result of our hardware compilation is a micropipelined [35], asynchronous circuit that directly implements the functionality of the source program. The circuits use a 4-phase bundled-data protocol [32] for communication. The control logic is closely tied to the synthesized datapath, and the computational part uses only local communication. Each channel can only be written by a single data producer. Computation thus requires no arbitration or scheduling, and communication is a lightweight operation. Preliminary results from the synthesis of Mediabench kernels [25], which tend to produce million gate equivalent circuits, reveal two interesting facts — one, these circuits have significant energy-delay advantages over synchronous implementations employing comparable technologies; and two, such a design paradigm appears to be layout-friendly — in all the circuits we have synthesized so far, we have had a smooth place-and-route experience (using Cadence Silicon Ensemble) without any post-layout timing violations.

2 Related Work

Compiling high-level languages into circuits, whether for the creation of ASICs or for use in reconfigurable computing is a widely studied area. C has been the starting point for many of these efforts, but in most cases, the language is either restricted, extended with hardware-specific extensions, or both. We compile all of ANSI-C, and handle programs without regard to their intended target domain. SA-C [4], StreamsC [19], Rapid-C [15] all restrict the source to a "synthesizable" subset of the language, and may include extensions such as explicit bitwidth specification or timing constraints. These languages also leverage their domain-specific nature, e.g., streaming multimedia applications. HardwareC [29], Transmogrifier C [20], Handel-C [13], Esterel-C [24], SystemC [36] are all more general, but require the programmer to include notions of hardware, e.g., I/O ports, bitwidths, explicit parallelism, etc. Their goal is to create a version of C that looks more like traditional HDLs, but is still accessible to C programmers.

Using C as a source language is more common in the domain of reconfigurable computing [3, 12, 22, 30, 39]. Our work differs from this body of work in two ways. First, we compile all of ANSI C to hardware. Second, and more importantly, we generate asynchronous circuits, without any centralized control.

Synthesis fbws to asynchronous circuits generally start from a high-level language (usually based on CSP [23]) suitable for explicitly describing the parallelism of the application, e.g., Tangram [38], Balsa [18], OCCAM [28], CHP [26, 37]. Synthesis tends to follow a two step approach: the high-level description is translated in a syntax-directed fashion into an intermediate representation; then, each component of the IR is (template-based) technology mapped into gates. An optional optimization step [14] may be introduced to improve performance.

There are many differences between our approach and other fbws. In short, no other fbw compiles all of C into energy efficient pipelined asynchronous circuits. Our input language is a well-established, imperative, sequential programming language. In addition, the compiler also performs extensive analysis and optimization steps on the intermediate form.

3 The CASH Front-End

This section briefly describes the front-end of our compiler, which translates the input ANSI-C program into the compiler's IR, Pegasus. The C front-end is based on Suif 1 [40], and performs parsing and some optimizations. Afterward the front-end translates the low-Suif IR into the Pegasus datafbw IR. Next, CASH performs numerous optimizations on this representation, including scalar-, memory-, and Boolean optimizations.

The implementation of C into hardware is simplified by maintaining roughly the same memory layout of all program data structures as implemented in a classical CPU-based system. ASH currently uses a single monolithic memory for this purpose. There is nothing intrinsic in our model that mandates the use of a monolithic memory; on the contrary, using several independent memories (as suggested in [33, 3]) would be beneficial.

The remainder of this section presents a brief overview of the Pegasus IR, and then describes how some of the more complex C language constructs are represented in Pegasus. See [7, 8, 9, 10] for more details on Pegasus and CASH.

```
int squares()
{
  int i = 0,
      sum = 0;

  for (i=0;i<10;i++)
      sum += i*i;
  return sum;
}
```

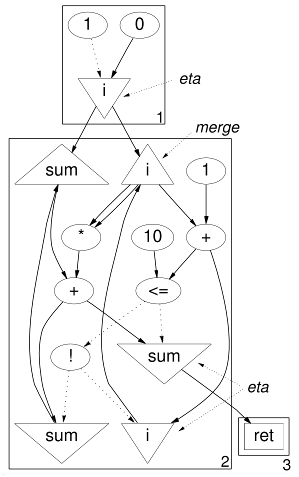

Figure 2: C program and its Pegasus representation. The dotted lines represent predicate values.

3.1 The Pegasus Intermediate Representation

In Pegasus, a program is represented as a directed graph in which nodes are operations and edges indicate value fbw. Pegasus is a form of Static Single Assignment (SSA) [16], an IR used for imperative programs in which each variable is assigned to only once. As such, it can be seen as a functional program [2]. Each ALU operation is represented as a unique node in the graph. When a value can be produced by several instructions (such as assignments to a variable in both the `then` and `else` branches of a conditional), Pegasus uses one of two special types of nodes to represent the "join" (corresponding to the ϕ nodes in SSA): decoded multiplexers (MUX) and MERGE nodes. While, MUX nodes are used to select the value from forward branches (like `if-else`), the MERGE nodes are used for merging the fbw at loop entry points. The latter can have several inputs, and when any one of the inputs is available, it is copied to the output.

A simple C program in Figure 2 illustrates Pegasus. The loop contains two MERGE nodes (shown as triangles pointing up), one for each of the loop-carried values, `i` and `sum`. Loop exit controls are handled by another Pegasus construct, the ETA node (shown as triangles pointing down). These nodes have two inputs—a value and a predicate—and one output. When the predicate evaluates to true, the input value is moved to the output; when the predicate evaluates to false, the input value and the predicate are simply consumed, generating no output. The example contains three ETA nodes: two of them send the values of `i` and `sum` back to the beginning of the loop, while the third sends the value of the `sum` to the return node when the loop is completed. Note that the predicate controlling the third ETA is the complement of the predicate of the first two.

Memory access is represented through explicit LOAD and STORE nodes. The compiler adds dependence edges, called *token* edges, to explicitly synchronize operations whose side-effects may not commute. These token edges are implemented as data-less channels. Operations with memory side-effects (LOAD, STORE, CALL, and RETURN) all have a token input. An operation with

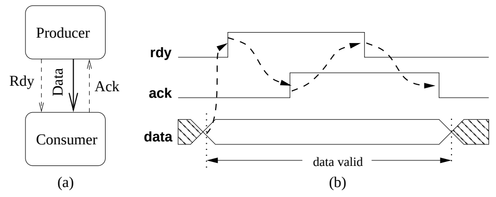

Figure 3: Bundled Data Protocol: (a) Signals (b) Complete Handshake

side-effects must collect tokens from all its potentially conflict-ing predecessors (e.g., a STORE following a set of LOADs).

3.2 Function Calls

Function calls can be implemented in several ways. The simplest method is inlining, which requires code duplication and is not applicable for recursive functions. Alternatively, each function is synthesized as a separate circuit. Calls to functions known at compile time involve routing control from the CALL node to the callee arguments, and from the callee's RETURN back to the call site. If the functions are not recursive this is easily achieved using a "call" asynchronous element [35]. Passing arguments can be done either over wires, on a stack, or using a hybrid solution (in the same way traditional compilers pass some arguments in registers and the rest on the stack).

Handling recursion simply requires the live variables of the caller to be saved on a stack (using STORE operations) before executing the call. After the recursive call completes, the live variables are restored from the top of the stack (using LOADs). One of the values saved is the identity of the caller's parent which is used by the caller to return when it completes.

We can partition the standard C library functions into two categories: functions implementable in C (e.g., memory allocation) and functions requiring system calls (e.g., file operations). Given the source code, CASH can synthesize the former as hardware circuits. The latter, which requires execution of system calls, cannot be described in pure C; these should be implemented on the processor by the hardware-software partitioner.

3.3 Compiler Status

The front-end and optimizer handle all of ANSI C except `alloca`, `varargs`, and `longjump`. While the former two constructs are relatively easy to integrate, handling `longjmp` is substantially more difficult. Strictly speaking, C does not have exceptions[1] and our compiler does not handle them.

While all of Pegasus is amenable to synthesis, we have not yet implemented fbating-point operations and procedure calls in CAB. The former is easily handled. The latter is harder to implement efficiently in full generality (i.e., in the presence of function pointers). However, our results indicate that ASH is most beneficial for implementing kernels, and thus a full-blown mechanism for handling large programs may be unwarranted.

[1]The meaning of a program which generates exceptions is undefi ned by the C standard.

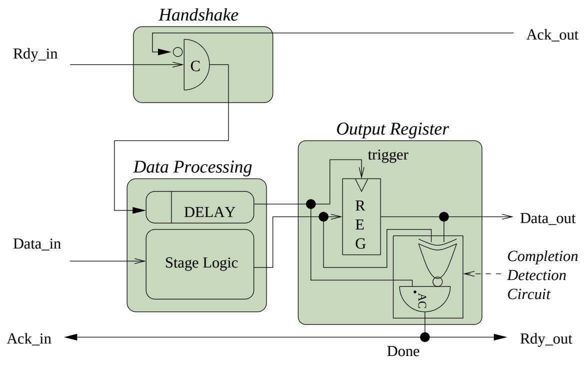

Figure 4: A Pipeline Stage for a single input, single output function

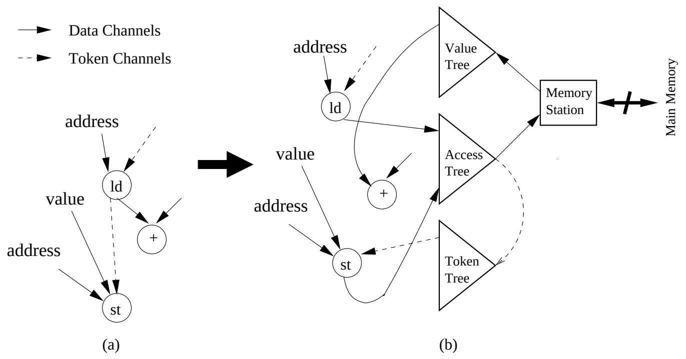

Figure 5: Memory access circuitry: (a) Sample snapshot of Pegasus graph with memory accesses (b) Circuit Implementation.

4 Circuit Architecture

CASH Asynchronous Back-end (CAB) synthesizes each node in the Pegasus graph as a unique pipeline stage. Data transfers between pipeline stages use the bundled data protocol [32] for signaling, as shown in Figure 3a: every data channel consists of a *data bus*, a *data ready* wire and an *acknowledgment* signal. A complete 4-phase handshake is shown in Figure 3b: a data item is sent from the producer to the consumer by placing it on the *data bus*, and then asserting the *Rdy* signal. This timing constraint (indicated by the left-most dashed line of the waveform) is called the *bundling constraint*. Once the consumer consumes the data item, it asserts the *Ack* signal. After that, the *Rdy* and *Ack* signals return to zero.

4.1 Basic Pipeline Stage

The architecture of a pipeline stage (Figure 4) is based on micropipelines [35]. Each stage performs three essential tasks — data processing (the stage's logic function), control-flow, and output data latching. The control-flow circuitry implements the 4-phase handshake using a Muller C-element. When the inputs arrive, the handshake asserts the *trigger* signal indicating that the stage is active. After a delay that matches the latency of the data processing logic, the *trigger* signal enables the latching of the output register. A completion detection circuit produces a *done* signal when the latching completes. This signal is then used as both the *data ready* (Rdy_{out}) for the consumers downstream, and as the *acknowledgment* (Ack_{in}) for the producers upstream.

Handling multiple inputs is straightforward: additional C-elements collect the *Rdy* signals of all inputs. The output can have a fanout greater than one; additional C-elements are used to collect the *Ack* signals from all of the consumers.

4.2 Memory Access

Every LOAD and STORE instruction in the IR is synthesized as a pipeline stage, and represents a point of memory access. Thus, there are potentially multiple points of access to a single shared resource, the memory. Figure 5 illustrates the four hardware structures currently used to implement memory accesses:

1. The Access Tree provides arbitration for accesses to main memory. The leaves of the tree are the different memory operations, and each internal node is a pipeline stage containing an asynchronous arbiter (mutex) guarding access to the parent node.

2. The Memory Station is the interface between the access tree and main memory. The memory station holds the LOAD operations that have been initiated but have not yet completed. Currently we allow only one outstanding LOAD. The memory station corresponds to the load-store queue (LSQ) in a classical superscalar processor [34]. Our current implementation approximates a 1-element LSQ.

3. The Value Tree mirrors the *access tree*. It routes LOAD results back to the circuit. Each internal pipeline stage of this tree is a demultiplexor. As a load access makes its way up the *access tree*, the path it takes is recorded and used by the *value tree* to return the result.

4. The Token Tree is similar to the *value tree*, but is used to forward the enabling token signal from the current memory access to the dependent operations once the current access has been issued. For example, the LOAD access in Figure 5(b) releases its token to the dependent STORE when it reaches the *memory station*.

CAB builds a program-specific memory access network using these structures. The protocol for issuing memory accesses mandates that an access can be issued only after all the memory operations that it is dependent on have issued. To enforce this, an access does not release its token until it reaches a common point for all access, the memory station, after which the token travels down the token tree.

4.3 Lenient (Early) Evaluation

One problem with the form of speculative execution employed by Pegasus is that the critical path of the entire construct is the longest of the critical paths, as shown in Figure 6.

In ASH we take advantage of additional control circuitry and use *leniency* to solve this problem. A lenient operation can compute its output using only a subset of its inputs. For example, a LOGICAL AND operation can determine that the output is false as soon as one of its inputs is false.

In general, there is a finite set of conditions under which an operation can be triggered early. If we define I as the set of all inputs and their data valid signals, then we can define the set of triggering conditions as: $C = C_{all} \bigcup_{I_k \subset I} \{C_i | C_i = F(I_k)\}$

```
if (x > 0)
   y = -x;
else
   y = b*x;
```

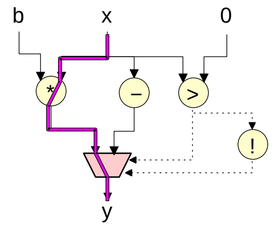

Figure 6: Sample program fragment and the corresponding Pegasus circuit with the static critical path highlighted.

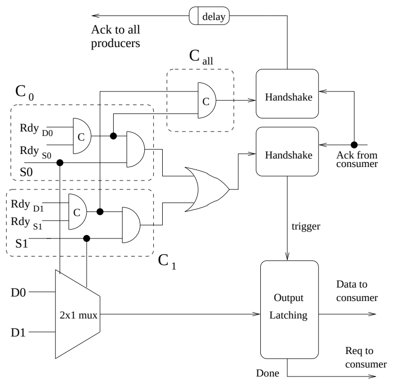

Figure 7: A lenient 2×1 mux: There are two sufficient conditions, C_0 and C_1, for the early triggering of the operation, and one necessary condition, C_{all}, for acknowledging all inputs

where $F(I_k)$ is a Boolean function which computes a lenient triggering condition, and C_{all} is the strict case when all inputs have arrived. The set on the right side of the union describes the lenient triggering conditions. If any one of the conditions in this set is true, then we say that a sufficient input condition has been met to trigger the node. In a strict operation, the set $C = C_{all}$.

Figure 7 shows a lenient implementation of a 2×1 MUX. The MUX has two data inputs, $D0$ and $D1$, and two selection inputs $S0$ and $S1$. Each of these inputs is accompanied by its Rdy signal. For this operation:[2] $C = \{C_{all}, C_0, C_1\}$, where $C_0 = (Rdy_{s0} \odot Rdy_{D0}) \wedge S0$ and $C_1 = (Rdy_{s1} \odot Rdy_{D1}) \wedge S1$.

Essentially, the MUX is triggered if either one of its data inputs, say D_i, and the corresponding selection input, S_i, have arrived, and S_i is true (condition C_i).

As a result of leniency the *dynamic critical path* is the same as in a non-speculative implementation. For example, if the multiplication in Figure 6 is not used, it does not affect the critical path. In this protocol a lenient operation sends an ack only after all its inputs have arrived. In the asynchronous circuits literature, leniency was proposed under the name “early evaluation” [31] and “or causality” [41]. A recently proposed enhancement [5] can send early acks, potentially speeding up computation at the cost of more machinery to cancel partially executed operations.

In addition to Boolean operations and multiplexers, all predicated operations are lenient in their predicate input. For example, if a LOAD operation receives a false predicate input, it can immediately emit an arbitrary output, since the actual output is irrelevant. The circuitry implementing leniency introduces some area overhead. However, since the number of operations that can be lenient is generally a small fraction of all nodes in a typical program, the performance gains outweigh the area overhead.

[2]In the equation, we use $\odot$ to denote a C-element operation.

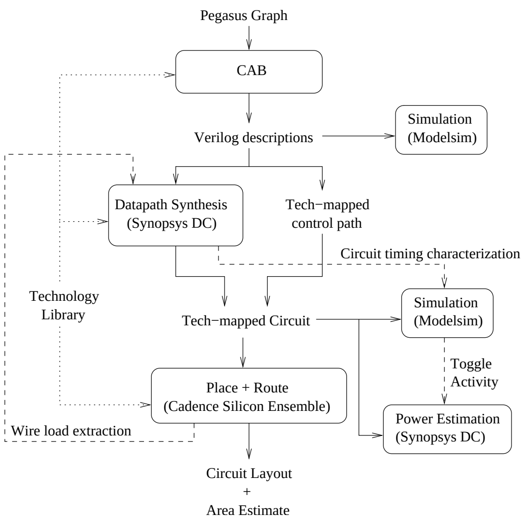

Figure 8: The Circuit Synthesis Flow

5 Circuit Synthesis Flow

In this section, we briefly describe the synthesis flow which transforms a Pegasus graph into a fully placed and routed circuit. Figure 8 illustrates this flow. The transformation takes place in three stages:

A) CAB: translates the Pegasus IR into Verilog. Most of the translation process is fairly straight-forward, as each node of the graph is translated into a pipeline stage (as described in Section 4). In addition, CAB also synthesizes the control-path and registers in each pipeline stage.

B) Synthesis: Synopsys Design Compiler (version 2002.05-SP2) is used only for the synthesis of the datapath ALUs. The result of this phase is gate-level Verilog description of the circuit, and its timing characterizations (for back-annotation in simulation).

C) Place-and-Route: Finally, this gate-level circuit is placed and routed using Cadence Silicon Ensemble (version 5.4). This produces a layout specification, and wire load information. The latter is back-annotated to the synthesis phase to generate new (and more accurate) circuit timing characterizations.

The circuit can be simulated after the completion of each of the above phases using a Verilog simulator (Modelsim SE version 5.8b, in our current tool flow). After processing by CAB, the circuit is not fully tech-mapped, but a mixed structural/behavioral simulation is useful for debugging. After synthesis, the critical path delay can be accurately estimated and used to obtain precise performance evaluations. After layout, the simulation uses the actual wire load information. Simulation produces a toggle activity file that is used to estimate dynamic power consumption by the Synopsys Design Compiler.

We have adapted commercial CAD tools customarily used for creating synchronous circuits to our asynchronous design fbw. This requires some awareness and caution. For example, we cannot allow Synopsys to synthesize (let alone infer) registers, and we cannot synthesize any combinational block with feedback loops (basic asynchronous blocks like C-elements contain feedback loops). We get around this problem by ensuring that only the datapath is synthesized by Synopsys, while CAB synthesizes the control path and registers.

We also use a standard cell library designed for use in synchronous circuits. In consequence, some commonly used gates in asynchronous circuits, such as C-elements, are not available in the library. Although we can create an implementation using existing library cells, our experiments show that such C-element implementations are 2-3 times slower and 2 times larger than a custom transistor-level design.

Our investigation indicates that the place/route phase is considerably simplified compared to a synchronous circuit design fbw. No clock-tree is required, routing does not need to be timing driven, and there are no clock-related timing violations. Timing closure is thus not necessary.

There are two possible sources of timing violations in our circuits, that can occur post-layout:

Matched Delay: each pipeline stage uses a delay element to match the data processing logic delay (see Figure 4). Actual logic delays after place-and-route may violate this assumption. However, we have engineered the circuits to reduce the probability of such a violation by ensuring that the data processing logic is a localized circuit with a fanout of exactly one (the output register is the only destination). This constraint allows for an early correct estimation of the matched delays, thus improving the whole synthesis run-time.

Bundling Constraint: the bundled data protocol works under the assumption that data *always* arrives at the destination before the corresponding Rdy signal. The completion detection circuit (see Figure 4) ensures that this constraint is obeyed at the output of a pipeline stage. However, layout could violate these constraints if the wires connecting pipeline stages are arbitrarily routed. The problem can be addressed by supplying hints to the layout tool — for example, by ensuring that the weight on the data wire is bigger than that on the Rdy wire (assuming that the tool optimizes for minimum wire-length).

In all of the circuits that we have synthesized (see the next section for a list), we have not encountered any timing violations. Overall, the experience with layout of our asynchronous circuits has been encouraging — not only have we had no timing violations, but often the post-layout circuit has better latency and power consumption than pre-layout estimates. It seems that the local communication and self-timed nature of our circuits simplify layout considerably.

6 Toolflow Evaluation

There are many metrics that can be used to evaluate a CAD tool fbw — design time, circuit performance, power, area. Our toolfbw optimizes design time. Despite this, it also produces circuits with excellent energy-delay, energy efficiency comparable to hand-optimized custom designs, and good performance. For a quantitative analysis, we compare our circuits against two

Benchmark	**Function**
adpcm_d	adpcm_decoder
adpcm_e	adpcm_coder
g721_d	fmult+quan
g721_e	fmult+quan
gsm_d	Short_term_synthesis_fi ltering
gsm_e	Short_term_analysis_fi ltering
jpeg_d	jpeg_idct_islow
jpeg_e	jpeg_idct_fslow
mpeg2_d	idctcol
mpeg2_e	dist1

Table 1: *Embedded benchmark kernels used for the low-level measurements. For `g721` the function `quan` was inlined into `fmult`.*

processor cores that execute the same C program: (1) a 4-wide out-of-order superscalar core, and (2) a single-issue in-order core. Both processors use aggressive clock gating. While it would be best if we could also compare our circuits to other asynchronous and synchronous implementations of the same programs, we do not have data for such implementations. Furthermore, we know of no other tools that will create such circuits from unannotated C-code; a head-to-head comparison is difficult.

Experimental Setup

We use the Mediabench suite [25] for the comparison. From each program we select the most important function, e.g., the one which takes the most time on the CPU (see Table 1). The data we present is for the entire circuit synthesized by CAB, including the memory access network (described in Section 4.2), but excluding the memory itself or I/O to the circuit. We report data only for the execution of the kernels itself, ignoring the rest of the program. We do not estimate the overhead of invoking and returning from the kernel.

We use a [180nm/2V] standard cell library from STMicroelectronics, tuned for high-performance. All numbers are post-layout estimates. The processors, operating at 2V, are simulated using Wattch [6]. The clock frequency is 600 MHz (the state-of-the-art for many commercial CPUs in an 180nm process). For all systems we assume a perfect L1 cache, and do not include the power consumption of the caches or memory in these results.

Measurements

Area. Figure 9 shows the area required for each of these kernels. For reference, in the same technology a minimal RISC core can be synthesized in 1.3mm^2, and a complete P4 processor die, including all caches, has 217mm^2. This shows that while the area of our kernels is sometimes substantial, it is certainly affordable, especially in deep sub-micron technologies.

Performance. Figure 10 compares the performance of our circuits against the two processor cores. The graph displays the ratio of the execution times of each kernel on the processors to the timing on ASH. Although ASH circuits perform better than the single-issue core, they often perform worse in comparison with the superscalar core.

We have uncovered two major sources of inefficiency in our circuits: **(1)** As described in Section 5, our C-element implementation from the available standard cells is highly suboptimal. Our experiments indicate that a faster C-element can speed-up

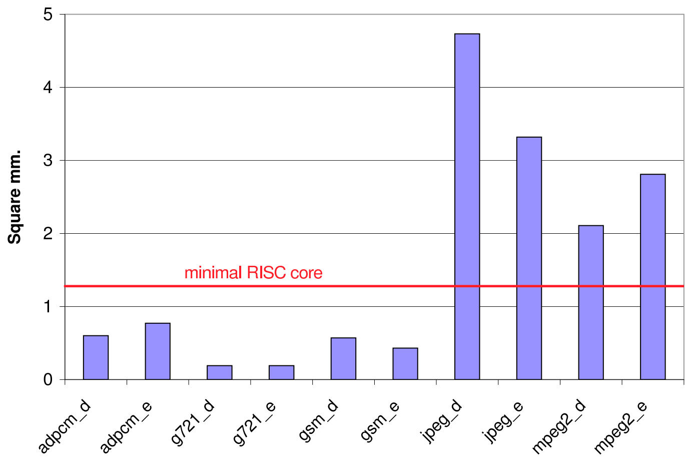

Figure 9: Silicon real-estate in mm^2 for each kernel

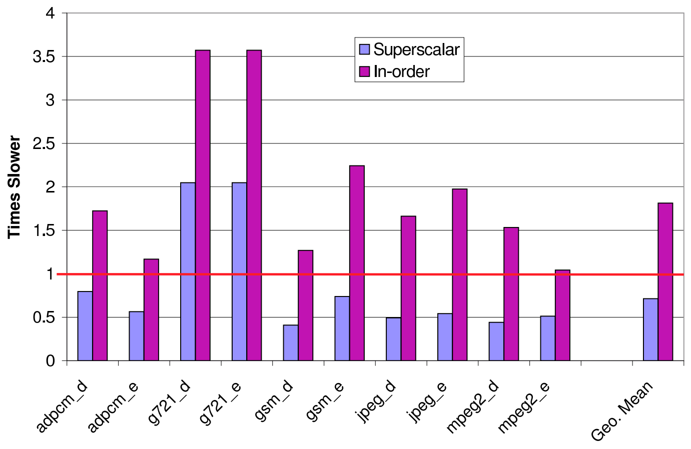

Figure 10: Performance comparison: how many times processor cores are slower than ASH. Values above 1 indicate that ASH is faster, while values less than 1 indicate a faster processor.

programs by 15% on an average. **(2)** The memory accesses protocol described in Section 4.2 is currently the main performance bottleneck. An operation cannot release a token to its dependents until its request has reached the memory station and the confirmation has traveled through the entire token tree. An improved construction would allow an operation to (i) inject requests in the network and (ii) release the token to the dependent operations immediately (e.g., the token should be sent to the dependents as the operation request enters the access tree). The network packet through the access tree must carry enough information to enable the memory station to execute operations in the original program order. A similar protocol is actually used by superscalar processors, which inject requests in order in the load-store queue, and can proceed to issue more memory operations before the previous ones have completed. A limit study we have performed by reducing the latency of a memory access tree stage to 10ps shows that performance can increase by as much as five times. Many of our programs are currently completely bound by the latency of the token path in the memory access network.

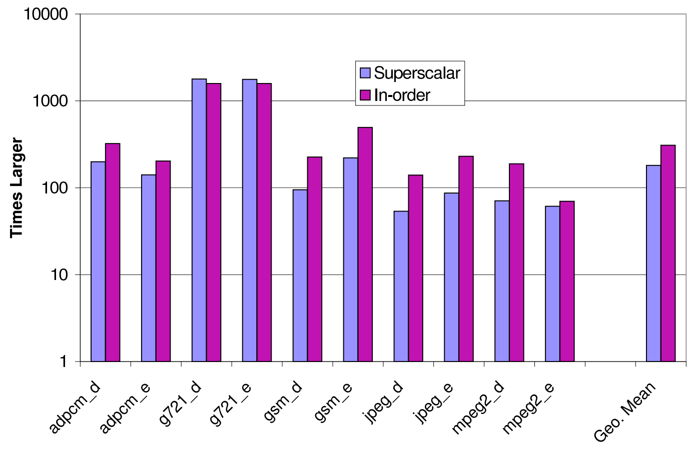

Figure 11: Energy-Delay product: Ratio of processors to ASH in logarithmic scale. Taller bars means that ASH circuits perform better.

Energy Efficiency. Our circuits are characterized by extremely low-power, and consume between 9mW and 24mW, with an average of 14mW. This compares with an average of 4.9W for the superscalar, and 1.3W for the in-order core. ASH consumes thus two to three orders of magnitude less power. In order to summarize power and performance in a single number we use the energy-delay metric, which was shown in [21] to be roughly supply-voltage and clock insensitive.

Figure 11 shows the ratio of energy-delay of the processors to ASH. ASH is one to three orders of magnitude better than a generic microprocessor. Intuitively, the only power drawn in ASH circuit is from computations that directly contribute to the result (with the exception of speculated operations). The distribution of control, and reduction of global wires have a significant impact on power consumption.

The energy efficiency of our circuits, expressed in useful arithmetic operations per nanoJoule varies between 10 and 100 ops/nJ. This makes our circuits 3 orders of magnitude better than superscalar processors, and one to two orders of magnitude better than DSPs and asynchronous processors [27, 42]. In fact, our circuits are comparable to the hand-optimized custom-hardware commercial implementations from [42]. [27] indicates that more than 70% of the power of the asynchronous Lutonium processor is spent in instruction fetch and decode, partly explaining the higher efficiency of ASH.

7 Conclusions

In this paper, we have presented a toolflow that automatically generates placed-and-routed asynchronous circuits from ANSI C programs. Our compiler extracts parallelism and constructs a dataflow graph from the input program. The dataflow graph is then translated into an asynchronous circuit with completely distributed control. Our results are extremely encouraging as the synthesized circuits are orders of magnitude more efficient (in terms of energy-delay) than a conditionally clocked superscalar processor, and is comparable to hand-designed dedicated hardware. The resulting circuits can be used in conjunction with a processor, or alone. Such an automated design flow helps to exploit the benefits of technology scaling while significantly reducing design time.

8 Acknowledgments

We want to thank the reviewers for their helpful and thorough comments. As usual we want to thank Dan Vogel for his great support. This research is funded in part by the National Science Foundation under Grants No. CCR-0224022 and No. CCR-0205523, by DARPA under contracts N000140110659 and 01PR07586-00.

References

[1] International technology roadmap for semiconductors (ITRS). http://public.itrs.net/Files/1999_SIA_Roadmap/Design.pdf, 1999.

[2] A. W. Appel. SSA is functional programming. ACM SIGPLAN Notices v. 33, no. 4, April 1998.

[3] J. Babb, M. Rinard, et al. Parallelizing applications into silicon. In *FCCM*, 1999.

[4] W. Böhm, J. Hammes, et al. Mapping a single assignment programming language to reconfigurable systems. *The J. of Supercomputing*, 21(2):117–130, February 2002.

[5] C. Brej and J. Garside. Early output logic using anti-tokens. In *International Workshop on Logic Synthesis*, pages 302–309, May 2003.

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

[7] M. Budiu. *Spatial Computation*. PhD thesis, Carnegie Mellon University, Computer Science Department, December 2003. Technical report CMU-CS-03-217.

[8] M. Budiu and S. C. Goldstein. Compiling application-specific hardware. In *Proceedings of the 12th International Conference on Field Programmable Logic and Applications*, September 2002.

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

[10] M. Budiu and S. C. Goldstein. Optimizing memory accesses for spatial computation. In *Proceedings of the 1st International ACM/IEEE Symposium on Code Generation and Optimization (CGO 03)*, March 23-26 2003.

[11] M. Budiu, M. Mishra, et al. Peer-to-peer hardware-software interfaces for reconfigurable fabrics. In *Proceedings of 2002 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2002.

[12] T. J. Callahan and J. Wawrzynek. Instruction level parallelism for reconfigurable computing. In Hartenstein and Keevallik, editors, *FPL'98*. Springer-Verlag, Sep 1998.

[13] Celoxica, http://www.celoxica.com. *Handel-C*.

[14] T. Chelcea and S. M. Nowick. Resynthesis and peephole transformations for the optimization of large-scale asynchronous systems. In *DAC*, pages 405–410. ACM Press, June 10–14 2002.

[15] D. C. Cronquist, P. Franklin, et al. Specifying and compiling applications for RaPiD. In K. L. Pocek and J. Arnold, editors, *FCCM*, pages 116–125. IEEE Computer Society Press, 1998.

[16] R. Cytron, J. Ferrante, et al. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[17] W. R. Davis, N. Zhang, et al. A design environment for high throughput, low power dedicated signal processing systems. *IEEE Journal of Solid-State Circuits*, 37(3):420–431, March 2002.

[18] D. Edwards and A. Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer J.*, 45(1):12–18, 2002.

[19] J. Frigo, M. Gokhale, et al. Evaluation of the Streams-C C-to-FPGA compiler: an applications perspective. In *FPGA*, pages 134–140. ACM Press, 2001.

[20] D. Galloway. The transmogrifier C hardware description language and compiler for FPGAs. In D. A. Buell and K. L. Pocek, editors, *FCCM*, pages 136–144, April 1995.

[21] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid State Circuits*, 31 (9):1277–1284, September 1996.

[22] R. Hartenstein. A decade of research on reconfigurable architectures - a visionary retrospective. In *DATE*, March 2001.

[23] Hoare. Communicating sequential processes. In *C. A. A. Hoare and C. B. Jones (Ed.), Essays in Computing Science, Prentice Hall*. 1989.

[24] L. Lavagno and E. M. Sentovich. ECL: A specification environment for system-level design. In *DAC*, pages 511–516. Association for Computing Machinery, June 1999.

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

[26] A. J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.

[27] A. J. Martin, M. Nyström, et al. The Lutonium: A sub-nanojoule asynchronous 8051 microcontroller. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, May 2003.

[28] D. May. OCCAM. *SIGPLAN Notices*, 18(4):69–79, May 1983.

[29] G. D. Micheli, D. C. Ku, et al. The Olympus synthesis system for digital design. *IEEE Design and Test of Computers*, pages 37–53, October 1990.

[30] R. Razdan and M. D. Smith. A High-Performance Microarchitecture with Hardware-Programmed Functional Units. In *MICRO*, pages 172–180, November 1994.

[31] R. B. Reese, M. A. Thornton, et al. Arithmetic logic circuits using self-timed bit level dataflow and early evaluation. In *ICCD*, page 18, September 23-26 2001.

[32] C. Seitz. *System Timing*. Introduction to VLSI Systems. Addison-Wesley, 1980.

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

[34] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 1995.

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

[36] Synopsys Inc., http://www.systemc.org. *SystemC Reference Manual*.

[37] J. Teifel and R. Manohar. Static tokens: Using dataflow to automate concurrent pipeline synthesis. In *10th Int'l Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 17–27, April 2004.

[38] K. v. Berkel. *Handshake Circuits: An Asynchronous Architecture for VLSI Programming*, volume 5 of *Intl. Series on Parallel Computation*. Cambridge University Press, 1993.

[39] K. Wakabayashi and T. Okamoto. C-based SoC design flow and EDA tools: An ASIC and system vendor perspective. *IEEE Trans. on CAD*, 19(12):1507–1522, December 2000.

[40] R. P. Wilson, R. S. French, et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In *ACM SIGPLAN Notices*, volume 29, pages 31–37, December 1994.

[41] A. Yakovlev, M. Kishinevsky, et al. *Formal Methods in Systems Design*, volume 9, chapter On the Models for Asynchronous Circuit Behaviour with OR Causality, pages 189–234. Kluwer, 1996.

[42] N. Zhang and B. Brodersen. The cost of flexibility in systems on a chip design for signal processing applications . http://bwrc.eecs.berkeley.edu/Classes/EE225C/Papers/arch_design.doc, Spring 2002.