General-Purpose Computation without General-Purpose Processors
A Thesis Proposal

Mihai Budiu

July 26, 2001

[Bud01]

Abstract

This document proposes a new architecture for general-purpose computing. This architecture uses a reconfigurable-hardware substrate and compiler technology to generate Application-Specific Hardware (ASH). The novelty of this architecture consists in the absence of resource reuse: each different static program instruction can have its own dedicated hardware implementation. ASH enables the synthesis of circuits with only local computation structures, which promise to be fast, inexpensive and use very little power. This also presents a scalable compiler framework for ASH, which generates hardware from programs written in high-level languages. Finally, this proposal describes an evolutionary path that can bridge the gap between the contemporary computer architectures and ASH.

1 Introduction

For five decades the relentless pace of technology, expressed by Moore's law, has supplied computer architects with ample materials in the quest for higher performance architectures. Each new hardware generation brings with it changes in the balance of the various architectural components, triggering new waves of research activity. (For example, processor performance improves at a higher rate than memory performance; when processors became faster than memory, cache technology became important.)

I will call the technological balance reversals *crossover effects*, from the crossing of the graphs of two functions with different rates of growth. Such crossovers cause the birth of new technologies. Sometimes crossover effects cascade: for instance, when VLSI technology could package on a single die enough transistors to implement a complete processor, suddenly the distances between components shrank. As a consequence we witnessed a second crossover: signals that used to cross chip boundaries could now be propagated in a single clock cycle, enabling a whole new set of architectural features.

By anticipating the technological evolution and by predicting crossover points, we can make educated guesses about the bottlenecks and tradeoffs faced by future architectures and we can devise timely solutions. This proposal addresses (in Section 3) the shortcomings looming in front of computer architecture in the next five to ten years and suggests a radical method to overcome some of them. I argue that the general-purpose processor has evolved to run any program and, thus, carries a lot of overhead baggage, mostly useless for any specific program.

In Section 4 I propose an alternative approach to implement general-purpose computation, which consists of synthesizing — at compile time — application-specific hardware, on a reconfigurable-hardware substrate. I argue that such hardware can be more efficient than a general-purpose CPU. I call this model **ASH**, from Application-Specific Hardware. The core of this computation model is the dataflow model. I propose to synthesize directly in hardware custom, application-specific dataflow machines. In the past, the dataflow paradigm proved unsuccessful because of high overheads steming from the interpretation of the dataflow on a (more or less) general-purpose architecture; however, the ASH machines have a very low overhead, as they implement the whole application in reconfigurable hardware, and avoid time multiplexing of hardware resources. The term *spatial computation* has been used for ASH-like paradigms.

ASH architectures hold the promise for low power consumption, high performance and very low cost. These architectures naturally provide the potential to explore a wide space of options for increased reliability and fault tolerance. In Section 5 I present a preliminary study which provides arguments for these claims.

In Section 6 I describe the **CASH** (Compiler for ASH) technology, a compiler spanning both the realm of traditional compilation and parts of hardware synthesis. Section 7 presents an evolutionary path that builds a bridge between two worlds and offers a continuum of architectures between classical general-purpose processors and ASH architectures. I conclude this document in Section 8 with an outline of the future course of this research.

2 Thesis Statement

Generating application-specific hardware and executing it on a reconfigurable-hardware substrate is a viable solution for enabling computer architecture to smoothly ride the technology curve for the next few decades. Scalable compiler technology for translating high-level language programs into hardware is feasible.

3 Shortcomings of Contemporary Architectures

In this section I motivate the need for the ASH architecture.

Moore's law has been equated with a guaranteed stream of good news, bringing higher speeds and more hardware resources in each new hardware generation. However, if we extrapolate the technology graphs using the historical growth rate, we rapidly approaching some important crossover points, which spell trouble for conventional architectures. Here I discuss some of these crossover points, what their (mostly negative) implications are, and how ASH promises to overcome some of these problems.

3.1 Billion Gate Devices

Crossover: Complexity becomes unmanageable.

According to Moore's law, the feature size of CMOS integrated circuits decreases exponentially with time: every 3 years designers have four times as many transistors to play with in the same area. In addition to the miniaturization of the basic components, the historic trend has been towards a steady increase in silicon-die size. Current microprocessors already are at 30 million transistors and, in a few years, we will reach the threshold of 1 billion. Other radical technologies, such as nanotechnology, promise to push this number even higher, to 10^{10} gates per square centimeter [CWB+99, GB01]. This wealth of resources has enabled designers to create more and more complex circuits, harnessing extra circuitry to exploit the parallelism in programs and to overcome the latency bottlenecks. The downside of such huge sizes is the enormous complexity of designing, testing, verifying and debugging such circuits. Most processors today are shipped with bugs [Col].

Solution: ASH addresses these shortcomings by offering a simple and clean path from programs to hardware implementations. ASH decomposes the problem into many smaller problems, having manageable size, which are compiled and optimized separately.

3.2 Low Reliability and Yield

Crossover: Device sizes decrease to a point where frequent defects are unavoidable.

When feature sizes become on the order of a few atoms, minor imperfections in the manufacturing process, random thermal fluctuations, or even cosmic rays can invalidate a circuit. Although some of the modern circuits do have error detection and correction capabilities (for example mainframe computers [SG99] and memories/buses with ECC), these solutions are still not universally applied to all parts of a processor design.

There are two types of defects we need to be concerned with: manufacturing-time, permanent defects and one-time events, caused, for instance, by high-energy particles.

Solution: ASH architectures offer a very simple solution to the problem of permanent defects and may enable simple and effective methodologies for dealing with one-time events. ASH uses reconfigurable-hardware devices as a substrate on which the computation is implemented. One important quality of such devices is that parts of the circuit are essentially interchangeable, so a defective part can be replaced by a working one at compile time. Research in the Teramac project [HKSW98] has shown this approach to be viable.

To deal with transient errors, we can imagine the synthesis of circuits containing embedded fault-tolerance features [Spi96]. Certainly, such a solution is also applicable to traditional CMOS-based devices, but the ASH framework may enable the synthesis of application-specific fault-tolerance devices, which can be more lightweight than a general-purpose solution.

3.3 Skyrocketing Cost of CMOS Manufacturing

Crossover: Cost required for fabrication dwarfs the resources available to most companies.

The cost of manufacturing an integrated circuit has two components: the cost of the manufacturing plant and the Non-Recurring Engineering (NRE) cost, which is on a per-design basis. The NRE cost includes the testing and verification cost mentioned in Section 3.1.

Moore's second law describes the exponential increase of the cost of a manufacturing plant with time; a state-of-the-art installation now costs more than 3 billion dollars. This cost comes mostly from the very fine mechanical and optical lithographic devices and masks. Another problem compounding cost is the yield, as very small devices are more prone to defects.

Solution: ASH advocates the use of a reconfigurable-hardware support for implementing the computations. Such devices have smaller densities than traditional CMOS devices, but they are "universal": one Field Programmable Gate Array (FPGA) can be used to implement any circuit, so the cost of the manufacturing plant and of the masks can be eliminated completely. As mentioned in Section 3.2, the reconfigurability provides tolerance to defects, allowing lower-yield devices to be used.

3.4 Inefficient Use of Resources

Crossover: The hardware resources become mostly idle.

Only a small fraction of the area of modern microprocessors is dedicated to actual computational units. Most of the chip resources are used for auxiliary tasks: L1 and L2 caches take most of the space, but other structures such as reorder buffers, reservation stations, issue logic, multi-ported register files, forwarding logic, jump prediction, trace caches, and others consume a lot of area and energy, but only support the computation. There is a tension in the design of the processor pipeline: on one hand, designers add enough resources to support program regions that can use them (with high ILP, unpredictable branches, etc.); on the other hand, most of the time these resources switch idly. For example, 4-wide issue processors seldom can sustain 1.5 IPC [HP96].

Solution: ASH proposes the radical solution of implementing for each program portion a separate piece of hardware, with resources exactly matching the available parallelism. Because we can statically describe all the required resources, a lot of the auxiliary structures can be eliminated completely (issue logic, reorder buffers, forwarding paths or multi-ported register files, instruction caches), while others can be reduced substantially (jump predictors, possibly data caches). Moreover, in ASH the structure of computation is highly localized, making it very easy to turn off all the unused part of the circuit.

3.5 Power Consumption

Crossover: The power consumed by a CPU cannot be thermally dissipated in an efficient way.

Despite the fact that the supply voltage is decreasing for each chip generation, the total power consumption is skyrocketing. Modern CPUs have reached more than 100 W, exacerbating the difficulty of cooling [Aza00] and of supplying power (especially for mobile devices). More than half of the power is consumed just by the clock signal, which spans the whole chip surface [Man95].

Solution: ASH synthesizes circuits that use exactly as many computational units as necessary for each part of the program; thus, most of the support structures in a CPU are no longer necessary. The circuits synthesized by ASH are activated according to program locality, the circuits corresponding to inactive parts of the program being turned off. Moreover, ASH generates very simple finite-state machines, which could be implemented in large part using asynchronous circuitry, eliminating most of the global clock signals.

3.6 Global Signals

Crossover: The time between two clock ticks is not enough for a signal to cross the whole chip.

A major problem engendered by the quickly increasing clock rate is that the electromagnetic signal does not have enough time to cross too many levels of logic. Deeper pipelining is a partial solution, but which provides diminishing returns due to the overhead of the pipeline registers [KS86].

Moreover, a lot of structures on a modern CPU require global signals or very long wires: the forwarding paths on the pipeline, the multi-ported register files, the instruction wake-up logic, and others. Wires connecting different modules tend to remain relatively constant in absolute size [AMKB00, HMH01], so they will be unable to function at higher clock rates.

Solution: The circuits implemented by using ASH are based solely on local signals that match exactly the program dataflow. For long-range communication (e.g., memory access or control transfers) we use pipelined communication channels. The finite-state-machine control is implemented in a completely distributed fashion, using token passing and requiring no global signals. We expect such circuits to scale very well with technology.

3.7 Complex CAD Tools

Crossover: Compilation takes too long.

CAD tools are very slow (taking on the order of hours to days to compile) and buggy [Dal01]; they require a lot of human intervention to generate good results [HMH01].

Solution: Several factors make the design of CAD tools for ASH simpler than that of full-blown general-purpose tools:

- The structure of the circuits generated by ASH is fairly simple and homogeneous: they all have a datapath and a very simple finite-state control. Our previous research [BG99] suggests that datapath-oriented CAD tools can be much simpler and faster than traditional CAD tools, as they need to do fewer optimizations and they deal with problems much smaller in magnitude (because the unit of compilation is the operation and not the logic gate).
- Compiling to reconfigurable-hardware substrates is intrinsically simpler than generating masks.

3.8 Reaching the Limits of ILP

Crossover: Computer architecture has reached diminishing returns in the exploitation of the ILP by dynamic schemes.

Processors cannot exploit ILP behind the limited issue window. However, the complexity of superscalar processors increases quickly with the size of the issue window [CG01], some hardware structures growing quadratically.

Solution: The compiler has a global view of the program and can expose distant ILP easily; by generating parallel hardware, exploiting the ILP in the ASH framework is straightforward, requiring almost zero overhead. By generating pipelined hardware, ASH also offers a natural path for exploiting the loop-level parallelism present in programs.

4 ASH Overview

In this section I briefly describe the research goals of this project and outline a solution that would meet them.

4.1 Goals

The goals of the ASH framework are to provide the following features: generality, good scalability, acceptable performance, low cost and low power. I discuss each of these features briefly in this section.

Generality: we want to implement a general model of computation, which can execute arbitrary application code and does not impose unreasonable constraints on the programming model. We want the framework to accomodate the programming languages available today and look and feel exactly like traditional programming (and not like hardware design).

Scalability: the major goal of this research is to provide an architecture that will scale well with time. Section 3 discusses a lot of the problems faced today or in the near future by traditional computer architectures and for which no obvious general solutions seem to exist. We hope ASH will provide a framework for smooth evolution of computing for the next 50 years.

Performance: we do **not** aim to provide a very high-performance architecture; we would be very satisfied if ASH, if implemented using today's technology, could approach today's CPUs performance. If the argument about the scalability of ASH is correct, the higher performance of ASH will come for free with future technologies.

Low cost: devices built using ASH should come at a lower cost than traditional CPUs, both in terms of investment and per-unit cost. More important, the cost of ASH devices should scale better with time compared to CMOS-based devices.

Low power: we intend to aggressively reduce the power consumed for computation, by as much as one order of magnitude.

Parallelism: we provide a simple methodology to exploit the instruction-level parallelism available in programs. Our technique enables a clear path towards the exploitation of other types of parallelism, such as loop-level and thread-level parallelism.

4.2 CASH: The Compiler

ASH mandates the compilation of complete programs into reconfigurable hardware, implementing each application completely in silicon. Figure 1 summarizes the tool-flow.

Programs written in general-purpose high-level languages are the input to the CASH compiler. CASH first applies traditional program-optimization techniques. The program is next decomposed into small pieces, called Split-phase Abstract Machines, or SAMs.

Each SAM is placed and routed independently by a local placer. All the SAMs that compose the program are input to the global placer and router, which uses a defect map of the target chip to decide how to layout the machines and how to connect them. The resulting "executable" is a configuration for the reconfigurable hardware. At run time the configuration is loaded on the reconfigurable-hardware substrate and executed. If the configuration is too large, a run-time hardware-virtualization method may be used.

My thesis will mostly be concerned with the top part of this diagram, labeled "compile-time". I will study the other two parts only sketchily, enough to create a trustworthy simulation infrastructure.

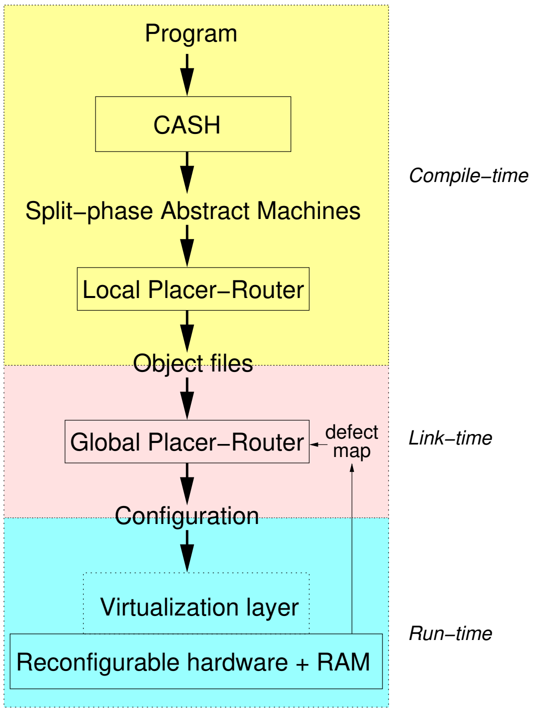


Figure 1: *The ASH tool-flow.*

4.3 Split-phase Abstract Machines

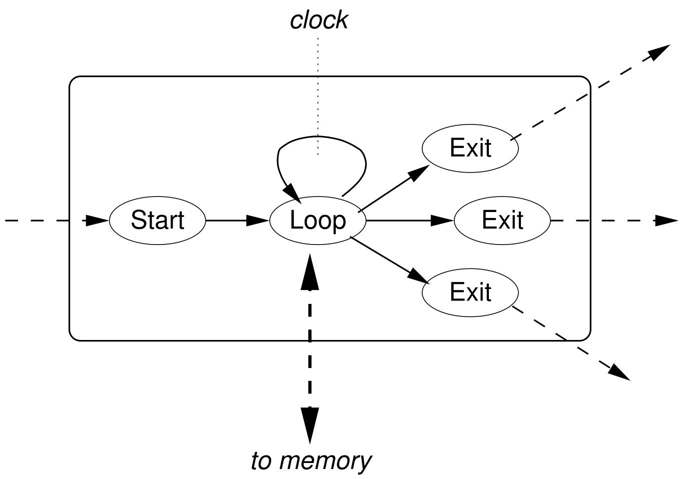


Figure 2: *The structure of a Finite-State Machine implementing a SAM. The solid lines indicate direct connections; the dashed lines indicate unpredictable-latency, pipelined communication.*

Each SAM is translated into a finite-state machine, as shown in Figure 2. The computation of the next state is a combinational circuit, which is synthesized as a dataflow machine. The local computation in a finite-state machine can access memory through a high-latency, pipelined interconnection network.

The body of each innermost loop in the source program becomes the "loop" macrostate in the figure. The loop itself is implemented as a finite-state machine, whose state is composed of the loop-carried variables. Some loops can span multiple SAMs, in the same way loops in high-level languages can span multiple procedures (by calling procedures in the loop body).

There are many possible translations of programs into SAMs; in this paper we mention two possibilities: constructing one SAM for each basic block (Section 5) and constructing a SAM from a hyperblock (Section 6).

During the program execution, at each instant we can discern three kinds of SAMs:

- **Inactive SAMs** are not being executed and do not have any live state. These SAMs do not need to consume any power and, if hardware virtualization is available, can be swapped out of the reconfigurable hardware.

- **One active SAM** is actively switching and consuming power and should be entirely swapped in; it is analogous to the procedure on the top of the stack (currently being executed) in a traditional model of computation.

- **Passive SAMs** are mostly quiescent: they store live values, but are blocked waiting for the completion of a "callee" SAM. They dissipate only static power most of the time[1] and correspond roughly to the procedures in the current call chain, which have been started in execution, but have not been completed.

4.4 An example

Figure 3 shows a simple C program and the equivalent translation into three SAMs. This figure has been automatically generated by our prototype CASH compiler using the VCG graph layout tool [LS93] and illustrates just one possible implementation.

The compiler creates three SAMs from this program:

SAM 1 implements the initialization of the variables `i` and `j` with 0. It receives as input the "program counter" (PC), which indicates the caller SAM. The shaded [red] empty oval indicates the "Start" state from figure 2, i.e. that the current FSM has just received control from the outside. The lightly shaded rectangles [green] with a sharp sign are output registers, containing data that is passed to SAM 2.

SAM 2 implements most of the computation in the procedure. It contains two additions, one for `i` and one for `j`, a comparison of `i` with 10, and two multiplexors (represented by the `?:` diamonds) that select the values for `i` and `j` based on the flow of control: either the initial value (if the machine is in the Start state) or the result from the increment operation (if the machine is in the Loop state). The multiplexors have two data inputs and two control inputs (dotted lines) each; the colors correspond: when the dark dotted line is asserted, the dark input is selected. The boxes marked with sharp signs # are registers holding the state, represented by the values of `i` and `j`.

This SAM is executed as long as the loop condition is true (i.e., `i < 10`). When the loop condition becomes false, control is transfered to the next SAM.

[1] There may be some concurrent activity between the passive SAMs and the active one, because of "instructions" that can be executed in parallel with the call.

```
int main(void)
{
  int i, j=0;
  for (i=0; i<10; i++)
    j+= i;
  return j;
}
```

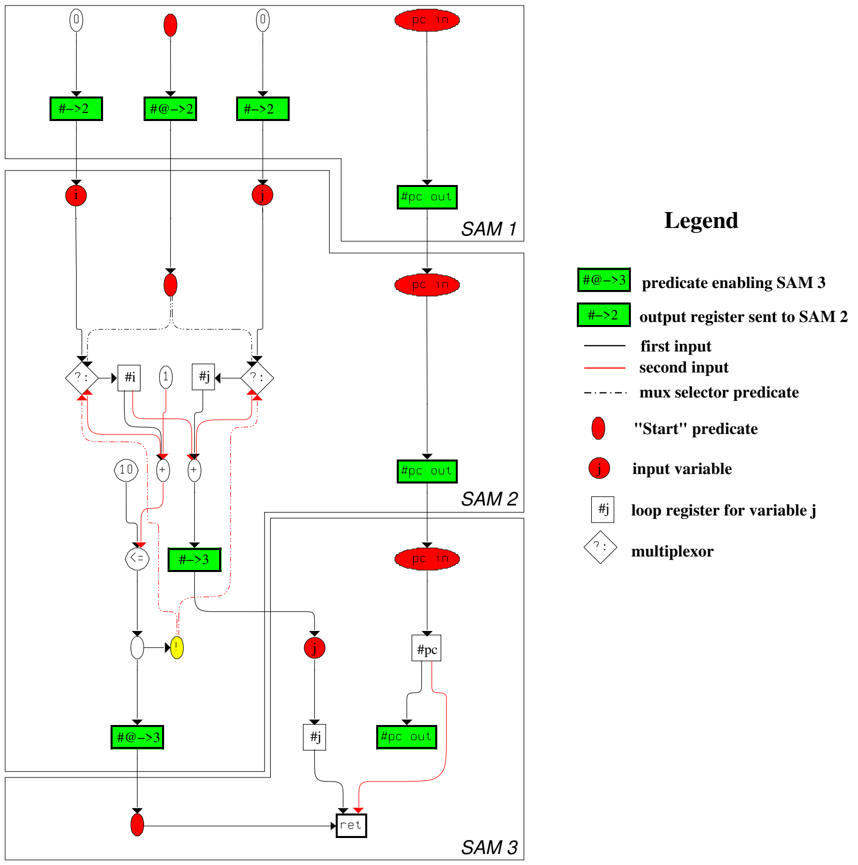


Figure 3: *A simple C program and its equivalent SAM implementation. The tokens are not explicitly represented in this figure: conceptually each wire has an associated token, indicating when the signal on the wire is valid.*

SAM 3 executes just the `return` instruction. It receives from SAM 2 the value of j and the PC and uses them as arguments to the return "instruction". Because the return instruction has a side-effect, it has a third input, a predicate, which indicates when the instruction is safe to execute. Because this return is executed unconditionally, the predicate is the same as the "Start" state for the SAM 3. This return instruction uses the PC value to return the control to the SAM that had originally invoked SAM 1.

4.5 Weaknesses of the ASH Model

In this section I address some of the weaknesses of the ASH model and discuss possible solutions.

- **ASH uses a lot of hardware resources.** Indeed, each different static operation in the program requires a different computational unit. It is unlikely that, in the near future, the amount of available hardware will be able to accommodate programs as big as Microsoft Word.

  The solution to this problem consists of hardware virtualization. This is a run-time mechanism that can provide the illusion of unlimited hardware resources, very much like virtual memory provides the illusion of unlimited RAM. An example of hardware virtualization can be found in multi-context FPGAs [deH96] and virtual pipelines [Sch97]; these systems can dynamically reprogram the hardware, changing its functionality on the fly. See also Section 7 for a discussion of an alternative CPU + ASH architecture.

- **Reconfigurable hardware has low density and speed.** Reconfigurable hardware incurs a big space and time overhead for storing configuration and for implementing wire interconnections with switches.

  We can see two mitigating factors for this problem: (1) the development of datapath-oriented reconfigurable hardware, which has a coarser granularity than traditional FPGAs (i.e. wider functional units) — for instance, using 8-bit functional units would reduce the configuration size roughly eightfold and would permit much denser implementations [GSM+99]; (2) nanotechnology promises to offer circuits that do not use extra memory to store the configuration and that have incredibly high density [CWB+99, GB01].

- **Costly place and route must be done at compile time.** In traditional CAD tools these operations are time consuming and resource intensive. Nonetheless, we believe that datapath-oriented CAD tools for datapath-oriented reconfigurable architectures, as typified for instance by our own DIL compiler [BG99, GSB+00], can achieve compilation speeds comparable to the software compilers. However, a lot of research remains to be done in this area.

- **We do not solve the memory latency and bandwidth problems.** Memory-access time is one important bottleneck for modern processors. ASH attempts to alleviate this problem by partitioning statically the program data in disjoint memory banks that can be placed close to the code using them; these banks can be accessed in parallel, providing increased memory bandwidth. However, it is unlikely that static analysis can do an efficient memory partitioning, especially in the presence of dynamic memory allocation and pointer-based code. It is also unclear how well local memories match with the requirements of hardware virtualization. Still, as Guri Sohi points out in [Soh97], even if we do not solve the memory problems, research in computer architecture can still be fruitful.

- **We do not have any multiprocessing and operating-system model.** At this stage we do not have any sound proposal on how the ASH model would be used to handle traditional multiprocessing and what its interaction with an operating system would be. I/O issues for ASH devices are also an open question.

5 A Preliminary Study

In this section I briefly describe a limit study that we carried out to assess the qualities of the ASH model. This work is more thoroughly described in our ISCA 2001 paper [GB01]. We use simulation to estimate the area and timing of applications implemented using the ASH model. We analyze the behavior of programs from the SpecInt95 [Sta95] and Mediabench [LPMS97] benchmark suites. We do not perform any reconfigurable-computing optimizations, (e.g., loops are not turned into pipelines) and we do not implement custom-size functional units. We assume no parallel execution or pipelining between independent SAMs.

5.1 Area requirements

We define the area unit as the area for the implementation of one memory word (4 bytes). We assumed that each integer operation (except multiply and divide) can be also implemented in one area unit[2]. For our benchmarks the total area (code and data) was between 2,000 and 250,000 units, a size that will be readily available in future hardware generations.

5.2 Trace Collection and Analysis

Each application to be simulated is compiled using `gcc` with maximum optimizations on an Alpha 21164 machine. The ATOM binary instrumentation tool [SE94] is used to instrument the program to generate a trace of important events. From the program trace we create a weighted graph (with weights both on nodes and edges). Each node represents either a SAM thread or a memory location; a node's weight is the estimation of its area. Each basic block of the program becomes a different SAM. A SAM exploits the complete ILP available in a basic block.

An edge between two basic blocks represents control-flow transfer and is weighted with the number of data values transferred between the two blocks. Transfers of control resulting from procedure calls are weighted with the total size of the procedure arguments. An edge between a basic-block node and a memory node is weighted with the number of accesses made from the block to that address.

After the program is run and the graph is built, the graph is placed on a two-dimensional grid. We strive to place together nodes connected by heavy edges. We carry out the placement in two stages: clustering and placement.

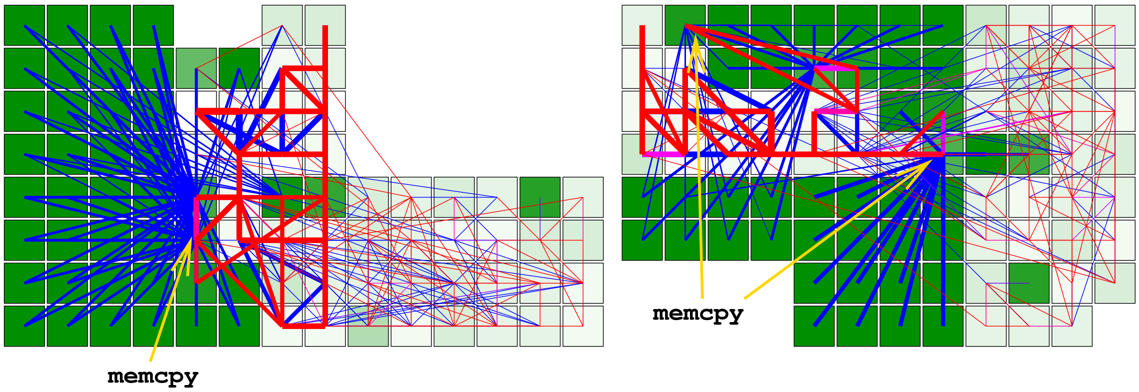
- In the clustering stage nodes are clustered together into supernodes of total weight 100, by minimizing the edge total weight between supernodes. We use the METIS graph partitioning tool [KK95] for this purpose.
- In the merging stage we place the supernodes on a two-dimensional grid. We greedily merge pairs of supernodes into larger rectangles until we are left with a single node.

Some characteristics of the resulting laid-out graph are worth noting: although it is sparse (the average node degree is less than 10 for all our programs), the node-degree distribution is very skewed (each graph has a few large "stars"). Figure 4 (a) displays one of the smallest graphs; each square is a supernode, obtained after clustering. The shading of the squares indicates how many of the objects inside are computations and how many are memory. A white square contains only code.

The edges represent communication. The edge width is the logarithm of the number of bytes transferred across the edge. The edge color indicates the mix of types of messages: dark edges indicate memory reads only, while lighter edges indicate control transfers, with intermediate shadings for mixed traffic. Despite this graph being very small, it exhibits some features typical for all our programs, such as the big "stars": code regions that touch most of the memory of the program. "Stars" are bad, because there is no way to place all adjacent nodes close to the star's center node; some have to be remote and thus will be expensive to access. For example, the `memcpy`

[2] In the cited study we used the unit of area to represent a physical device which we called "cluster". According to this definition of the unit, our area estimation is pessimistic for memory, which can be packed denser, and optimistic for instructions, which would require somewhat more space.

standard-library function was the center of one star in several programs, because it would access most of the memory used by the program. To reduce the size of these stars we have inlined `memcpy`. Each new copy then services a different set of memory locations, to which it can be placed closer (Figure 4 (b)). This significantly improves the graph layout and the program performance. The area cost of inlining is negligible (less than 1%).



(a) The graph for g721_e **(b) Same graph after inlining memcpy**

Figure 4: *A placed graph before and after inlining a procedure. Each square represents 100 units of area. A white square contains only code, while a shaded square [green] contains memory too. Light edges [red] indicate control-transfer, dark edges [blue] indicate memory reads. Edge thickness is logarithm of traffic along that edge.*

5.3 Trace-based simulation

The second phase of the simulation runs the ATOM-instrumented binary, generating a trace of all important events: transfer of control between blocks and memory-read operations. We use this trace to evaluate the running time of the program when implemented as a circuit whose layout is computed by our placement algorithm. Remote operations (i.e., memory reads and inter-SAM transfers) take time proportional to the Manhattan distance between the communication end-points.

5.4 Comments

In some respects our simulation methodology is overly optimistic, in others it is pessimistic. Some of the more important optimistic assumptions:

- The placed graph depends on the input data. In general we cannot know what locations will be used.
- We completely ignore routability issues. We assume sufficient wire bandwidth is available to connect the nodes point-to-point.
- We assume full memory parallelism: any two memory accesses can be made in parallel, with no contention, either on the network or in memory.

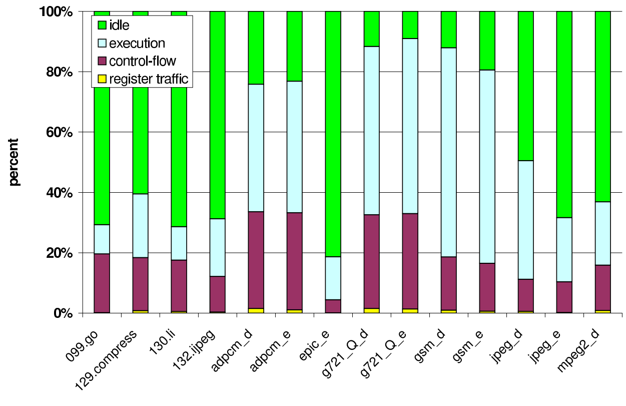


Figure 5: *Breakdown of execution time assuming the signal takes 5 cycles to transit between the centers of two neighboring squares. Idle time is the time spent waiting for memory-read operations to complete.*

The important pessimistic assumptions are:

- A lot of potential parallelism is not exploited in this scheme; only intra-basic block parallelism is available. Our new hyperblock-based scheme will expose more of it.
- The register allocation by gcc hides some ILP when spilling occurs.

The average running time for the benchmarks implemented using this model was within a factor of 3 from the running time on the Alpha processor, with a maximum slowdown of 10 (we assume the reconfigurable fabric can run at the same clock cycle as the Alpha). Figure 5 shows how the running time of each application was spent. We notice that the dominant cost is most often "idle time", spent waiting for memory reads to complete; there is indeed a good correlation between idle time and slowdown. The lack of caches in this architecture makes remote memory operations costly.

5.5 Conclusions of the Study

Here are some lessons learned from this study:

- Some important classes of programs (e.g., multimedia processing) can fit within a small amount of hardware and in future generations may be completely implementable in hardware without the need for virtualization.
- The spatial model of computation exhibits different trade-offs than a traditional architecture (CPU executing software). Some traditional compiler optimizations have new effects: for instance, procedure inlining can improve the spatial locality of programs.
- Performance of the spatial model of computation can approach the performance of traditional processors, if we assume identical performance for the underlying hardware.

6 The CASH Compiler

In this section I describe CASH, our current implementation of the ASH Compiler.

Our compiler infrastructure is built around the SUIF 1.3 research compiler [WFW+94]. For the moment, we do not use any of the parallelizing components of SUIF.

Traditional compilation. Most of the compiler front-end optimizations (e.g., dead-code elimination, copy propagation, common subexpression elimination, unreachable code removal) are beneficial in the context of the ASH framework. We also use aggressive procedure inlining. We plan to use dependence analysis to assist the loop pipelining (see below).

Width analysis and hardware cost estimation. Because we target reconfigurable hardware, we can implement arbitrary-width arithmetic at a lower cost than standard word-size computations. We have developed a compiler analysis called BitValue [BSWG00], which can discover narrow-width scalar operations in C programs[3]. This analysis is used to assess the hardware resources required for implementing each program construct.

Kernels selection. When we target a mixed system, comprised of a traditional CPU and a reconfigurable system (see Section 7), this compilation phase selects program portions that are most likely to provide benefits when executed on the reconfigurable fabric. We make this decision entirely statically now, but we plan to also use profile information. Unlike other published work [LCD+00], one of our basic assumptions is that the amount of available hardware resources will be large enough to accommodate entire procedures. As a consequence we no longer need complicated algorithms to trim down the selected kernels.

Memory partitioning. A variety of techniques can be employed to discover for each piece of code the memory regions that it will access. For array-based code this is a relatively simple task. For pointer-based code we can use a variety of techniques: type based or pointer-analysis based.

Once this kind of information is available, various techniques can be used to co-locate the memory and the code accessing it [GS99, SSM01, BLAA98, BRM+99]. An excellent overview of how memory partitioning is done in embedded systems can be found in [PDN99]; many of these techniques could prove valuable in the ASH framework.

We can also use optimistic techniques to deal with memory placement when code is unanalyzable, in the same way the RAW architecture falls back on a dynamically scheduled network when the access pattern cannot be predicted statically [LBF+98].

This part of the compiler is currently not implemented.

Hyperblock-based SAM selection. To extract a large amount of ILP from the programs we use the notion of hyperblock[4] [MLC+92], which has been introduced in the context of predicated-code generation. Each hyperblock becomes a SAM. The use of hyperblocks for reconfigurable-hardware generation has been already suggested in [CW00, SNBK01], but our approach is more general:

[3] Other similar analyses have been published concurrently [MRS+01, SBA00].

[4] A hyperblock is a part of the program control-flow graph which has a single entry point.

1. We completely cover each procedure with disjoint hyperblocks, using a linear-time algorithm. Hyperblock entry points are given by the loop entry points in the control-flow graph. Our method of hyperblock selection is completely general and deals with unstructured flow of control.

2. There are several knobs that we can turn to tune the hyperblock selection which allow us to optimize the resulting circuit for area, speed, or power: by choosing the hyperblock entry and exit points we can tune the size and power consumption. By duplicating code we can increase hyperblocks, thus enabling the exploitation of more ILP at the expense of more area and power.

   We intend to do a systematic exploration of the space of possibilities.

3. We have implemented some of the techniques described in the literature on compilation for predicated architectures: merging static single assignment and predication [CSC+00], merging speculative execution and predication [ACM+98]. Some other techniques could be applied as well, such as BDD-based predicate analysis and simplification [SAmWH00].

4. We can completely avoid some of the difficulties that classical compilers face when using predicated code: balancing the control flow [AmWHM97], register allocation [GJJS96], dataflow analysis [JS96], interference between speculation and predicated-code execution [MN99], predicate factorization [ASP+99].

Dataflow-machine generation. For each hyperblock we generate a finite-state machine. The machine has a combinational portion, which computes the next state and a sequential portion, which implements the looping. The FSM state consists of the loop-carried variables.

The combinational portion of the machine is implemented as a dataflow machine [Pap88, Tra86]. We use a technique very similar to the synthesis of dependence flow graphs [PBJ+91], which we adapted to handle predicated code; we also handle imperative languages with update-able values. In our implementation, the "tokens" are no longer explicitly represented: they become two 1-bit wires connecting the functional units (one wire is used to signal data availability, the other to confirm data consumption). There is no token store, token-matching logic or register file. The main overhead of the interpreted dataflow machines is thus completely eliminated. Static scheduling can also eliminate most of the token synchronization, as described below.

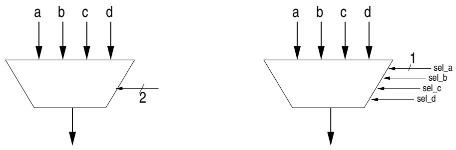
Each basic block in a hyperblock has an associated *path predicate*, as described in [CSC+00]. We use control-equivalence relations (as suggested in [GJJS96]) and some simple local structural properties of the control-flow graph to simplify the predicates. The predicate computation is implemented in hardware, using the same dataflow style (see Figure 3). The path predicates are used to guard the execution of instructions with side-effects (memory writes, memory reads that can trigger exceptions, procedure calls and returns). The path predicates are also a part of the finite-state-machine control (i.e., they compute the next state).

As in the Threaded-Abstract Machine (TAM) [CGSvE93] model, the operations having unpredictable latency (memory access, control-flow transfer, and procedure calls) are transformed into communication primitives.

Logically, each data signal has an associated "token" signal, which is used to indicate when the value signal is valid (i.e. its computation has terminated). The tokens can be implemented as 1-bit wires connecting the producer and consumers of a value; the tokens act as "enable" signals for the consumers. Each consumer has a reverse acknowledgement signal, to indicate consumption of the data value. We expect that in synchronous hardware implementations most of the token signals can be optimized away by statically scheduling the operations with known latency (more precisely, we can use a single token for all operations executed during the same clock cycle, and we can dispense with the acknowledgement altogether). Passing tokens will remain necessary between producers which have unpredictable latency (i.e. remote operations, like memory reads and procedure calls) and their consumers.

The dataflow machine is essentially a static single assignment representation (SSA) [CFR+91] of the predicated program. All instructions with no side-effects are speculatively executed; this is equivalent to the *instruction-promotion* technique described in [MLC+92]. Unlike other SSA representations for predicated code [CSC+00], we explicitly build the circuitry to compute the ϕ functions[5], which become multiplexors in hardware. The multiplexors are controlled by the path predicates. We found that this explicit representation of the complete program is extremely handy, enabling a lot of classical compiler optimizations (dead-code elimination, common-subexpression elimination, constant folding, etc.) to be carried in linear time and using very simple and clean algorithms.

Predicated-execution architectures have had to deal with the problem of balancing the control-flow paths which are speculated; for instance, if the `then` side of an `if` takes much longer than the `else` side, speculating through both will harm the execution sequences in which only the `else` side should be executed [AmWHM97]. We have a very simple solution to this problem, which consists of using *lazy, fully decoded multiplexors.*



(a) A normal MUX (b) A pre–decoded MUX

Figure 6: *Normal multiplexors vs. pre-decoded multiplexors.*

Recall that multiplexors are synthesized to select between multiple converging definitions of a variable. Our multiplexor implementation uses fully-decoded multiplexors, which have as many selector bits as there are inputs (Figure 6). These multiplexors do not need the complicated encoding/decoding logic for the selection signals and can be very cheaply implemented in hardware. When the computation leading to the multiplexor is unbalanced, we use synchronization tokens to signal the completion of the operations computing the data values and predicates. A lazy multiplexor can generate its output as soon as the "true" selector predicate and its corresponding data item have arrived. (For example, in Figure 6 (b), assume that the `a` input is valid, and the `sel_a` selector is asserted; we can then output `a` without waiting for the other signals to become valid.) Thus the circuit always computes the output in minimum time.

There is a very interesting interplay between predicates and tokens. Tokens indicate when the input values are ready, while predicates indicate when the operation should be executed. Because we are speculatively executing operations, the fact that the inputs are all ready (i.e. all tokens have arrived) does not necessarily imply that the operation should be executed: if the operation

[5]These are the diamonds in Figure 3.

has a side-effect it is also guarded by the path predicate. Only when the path predicate is true can the operation be executed.

Tokens are used not only to signal that data values are ready, but also to preserve the original program order between instructions which have side effects. The previous work on data-flow machines [Pap88, Tra86, Hel89] could dispense with this feature because it was handling functional languages. For instance, two store instructions that have no data dependency between them cannot be reordered if they may update the same memory location. Alias analysis (e.g., [HT01, Deu94, WL95, Ste96, SH97]), commutativity analysis [RD97] and other side-effect analysis (e.g., [RR00]) techniques can be used to remove unnecessary tokens between instructions with side effects which are provably independent, unleashing more parallelism.

The side-effect operations will thus execute only when both tokens and the controlling predicate are enabled. We can use this fact to reduce the number of signals, for instance by merging the token with the predicate in a single signal and setting the other signal to constant "enable". Another optimization we can do is to combine the tokens for speculated instructions with their path predicates; in this way, we effectively disable the speculative execution of that instruction and all its dependents, saving dynamic power.

In our future research we intend to explore the impact that the interplay between tokens and predicates has on the resulting circuit.

Software/hardware pipelining. A very effective method to generate high-performance implementations is to use hardware pipelining to exploit the parallelism. Various techniques can be used:

- If the memory access can be decoupled from the execution [SWP86], maximum pipeline parallelism can be exploited, like in the PipeRench architecture [GSM+99].
- If we can analyze data-dependences at compile time, we can generate static pipeline schedules, like the ones described in [LCD+00].
- Finally, if the code is not analyzable, for instance because it contains pointers or control flow, we can generate a pipeline with speculative execution and hardware to detect conflicts at run time [MH00, MTH99].

This part of the compiler is currently not implemented.

Placement and routing. The last phase of the compiler should handle placement and routing of both SAMs and the interconnection network that is used to connect SAMs to each other and to memories. This thesis will be concerned only with superficial aspects of placement and routing, which will be used just to obtain some rough estimates on the communication time. Our previous work [BG99] on datapath oriented placement and routing make us believe that this task is not insurmountable.

This part of the compiler is currently not implemented.

7 Evolutionary Path

As described in Section 4, one of the main goals of the ASH architecture is to provide good scalability with future technology. In this section we discuss how a smooth transition can be made towards an ASH architecture starting from the contemporary CPU designs.

The main problem faced by the ASH model is resource consumption; contemporary reconfigurable-hardware circuits have low densities and cannot provide enough resources to implement large programs. We imagine two different types of solutions to this problem: (1) hardware virtualization, which is a technology that simulates unbounded hardware resources by time-multiplexing the actual physical resources between computations that occur at different moments in time; (2) a combination of a reconfigurable-hardware system and a CPU; using such a system, we map only part of the program on the reconfigurable hardware, while the rest is executed on the CPU. Let us discuss these two alternatives in more detail.

7.1 Hardware virtualization

There is a wealth of work on hardware virtualization and run-time reconfiguration: [TCJW97, GF96, CWG+98, HFHK97, RLG+98, SV98, Xil96].

We believe that this paradigm will become more viable in the future due to two factors: (1) the increase in available reconfigurable-hardware resources up to a point where we can map the whole working set of the program (2) the emergence of datapath-oriented reconfigurable hardware, which can exhibit higher density and lower reconfiguration time.

Hardware virtualization is completely transparent to the application, being managed by a run-time mechanism and a configuration cache. Some hard problems exist, though, such as the interaction among defects, virtualization, and link-time placement.

7.2 ASH and CPUs

Several research projects have investigated the tight coupling of reconfigurable hardware architectures with general-purpose processors [SNBK01, CW98, WBG00, Hau00]. These architectures can offload part of the computation from a CPU to a reconfigurable fabric. One of the hardest problems in this direction of research is the automatic partitioning of the program. The scarcity of computational resources on the reconfigurable hardware has made the problem of kernel selection particularly difficult. We believe that the two factors cited above (resource increase and higher density) will make this task much simpler in the future.

7.2.1 A Preliminary Study for an ASH+CPU system

We have built the complete infrastructure for a study of the interplay between a reconfigurable-hardware fabric and a superscalar processor. Our model assumes that we have a large amount of reconfigurable hardware placed close to the processor pipeline, having direct access to the processor registers. The reconfigurable fabric behaves like an ordinary functional unit with an unpredictable latency; we call it a Reconfigurable Functional Unit (RFU).

The RFU can implement both combinational and sequential programs. The RFU can also access memory through the data cache (so we do not have to worry about coherence problems between the RFU and the main processor). The communication between the processor and the RFU is made through special instructions: instructions to pass register values to the RFU, to read register values from the RFU, to load an RFU configuration, and to start the RFU execution. The synchronization between the CPU and the RFU looks exactly like the synchronization between the CPU and a long-latency functional unit.

We have implemented several hardware–software partitioning algorithms using the SUIF compiler infrastructure. We use the CASH compiler to create the RFU configuration. We have adapted the SimpleScalar simulator [BA97] for cycle-accurate simulations of CPU+RFU combinations.

[I have almost completely built the infrastructure for this study. However, I do not yet have the simulation results. I will do my best to be able to present these at the oral presentation of this proposal.]

8 Timeframe

In this section I describe the plan for the rest of this thesis work. I intend to focus mainly on the front-end issues, with particular emphasis on the CASH compiler.

- July 2001: explore the interplay ASH+CPU using the developed SimpleScalar simulation infrastructure
- August 2001: finalize and debug the core of the CASH compiler
- Fall 2001: create realistic simulation models for important ASH components: the interconnection network, the memory controllers, I/O
- Winter 2001: create models for cost, area and power consumption
- Winter 2001–2002: explore alternative methods of hardware/software partitioning
- Spring 2002: study loop parallelization through pipelining (three variants: decoupled access, static schedule, speculative parallelism)
- Summer 2002: explore some of the architectural trade-offs: SAM selection, predicate/token duality, safe/optimistic memory partitioning, interconnection network complexity (bus/switched), etc.
- Fall 2002: write thesis
- Winter 2002: defend

References

[ACM+98] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. 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.

[AMKB00] V. Agarwal, H.S. Murukkathampoondi, S.W. Keckler, and D.C. Burger. Clock Rate Versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[AmWHM97] David I. August, Wen mei W. Hwu, and Scott A. Mahlke. A Framework for Balancing Control Flow and Predication. In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[ASP^{+}99] D.L. August, J.W. Sias, J.-M. Puiatti, S.A. Mahlke, D.A. Connors, K.M. Crozier, and W.-M.W. Hwu. The Program Decision Logic Approach to Predicated Execution. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 208–219, 1999.

[Aza00] Kaveh Azar. The History of Power Dissipation. *Electronics Cooling Magazine*, 6 (1), 2000.

[BA97] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. In *Computer Architecture News*, volume 25 (3), pages 13–25. ACM SIGARCH, June 1997.

[BG99] Mihai Budiu and Seth Copen Goldstein. Fast Compilation for Pipelined Reconfigurable Fabrics. In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, 1999.

[BLAA98] Rajeev Barua, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Memory Bank Disambiguation Using Modulo Unrolling for Raw Machines. In *Proceedings of the Fifth International Conference on High Performance Computing*, Chennai, India, December 1998.

[BRM^{+}99] 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.

[BSWG00] 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.

[Bud01] Mihai Budiu. General-Purpose Computation without General-Purpose Processors. Thesis proposal, July 2001.

[CFR^{+}91] 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.

[CG01] Ramon Canal and Antonio González. Reducing the Complexity of the Issue Logic. In *Proc. of the International Conference on Supercomputing (ICS-01)*, June 2001.

[CGSvE93] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. TAM — A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, July 1993.

[Col] Robert Collins. Dr. Dobb's Microprocessors Resources: Intel Secrets and Bugs. http://www.x86.org/secrets/intelsecrets.htm.

[CSC^{+}00] L. Carter, E .Simon, B. Calder, L. Carter, and J. Ferrante. Path Analysis and Renaming for Predicated Instruction Scheduling. *International Journal of Parallel Programming, special issue*, 28(6), 2000.

[CW98] 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, Tallinn, Estonia*, volume 1482 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1998.

[CW00] 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.

[CWB+99] C. P. Collier, E. W. Wong, M. Belohradský, F. M. Raymo, J. F. Stoddart, P. J. Kuekes, R. S. Williams, and J. R. Heath. Electronically Configurable Molecular-Based Logic Gates. *Science*, 285:391–394, July 16 1999.

[CWG+98] S. Cadambi, J. Weener, S.C. Goldstein, H. Schmit, and D.E. Thomas. Managing Pipeline-Reconfigurable FPGAs. In *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, February 1998.

[Dal01] Bill Dally. Personal communication, July 2001.

[deH96] André deHon. Dynamically Programmable Gate Arrays: A Step Toward Increased Computational Density. In *Fourth Canadian Workshop of Field-Programmable Devices*, May 1996.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *Proceedings of the ACM SIGPLAN '94 conference on Programming language design and implementation*, pages 230–241, 1994.

[GB01] Seth Copen Goldstein and Mihai Budiu. NanoFabrics: Spatial Computing Using Molecular Electronics. In *Proceedings of the 28th International Symposium on Computer Architecture 2001*, 2001.

[GF96] C. Ebeling D. C. Green and P. Franklin. RaPiD – Reconfigurable Pipelined Datapath. In R. W. Hartenstein and M. Glesner, editors, *Field-Programmable Logic: Smart Applications, New Paradigms, and Compilers. 6th International Workshop on Field-Programmable Logic and Applications*, pages 126–135, Darmstadt, Germany, September 1996. Springer-Verlag.

[GJJS96] D.M. Gillies, D.R. Ju, R. Johnson, and M. Schlansker. Global Predicate Analysis and its Application to Register Allocation. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 114–125, 1996.

[GS99] Maya Gokhale and Jan Stone. Automatic Allocation of Arrays to Memories in FPGA Processors with Multiple Memory Banks. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 63–69, Los Alamitos, CA, April 1999.

[GSB+00] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and Reed Taylor. PipeRench: A Reconfigurable Architecture and Compiler. *IEEE Computer*, 33(4), April 2000.

[GSM+99] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. PipeRench: a Coprocessor for Streaming Multimedia Acceleration. In *Published in proceedings of the 26th International Symposium on Computer Architecture ISCA 99*, 1999.

[Hau00] John R. Hauser. *Augmenting a Microprocessor with Reconfigurable Hardware.* . PhD thesis, University of California, Berkeley, 2000.

[Hel89] Steven K. Heller. Efficient Lazy Data-Structures on a Dataflow Machine. Technical Report MIT-LCS-TR-438, Massachusetts Institute of Technology, 1989.

[HFHK97] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera Reconfigurable Functional Unit. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '97)*, pages 87–96, April 1997.

[HKSW98] James R. Heath, Philip J. Kuekes, Gregory S. Snider, and R. Stanley Williams. A Defect-Tolerant Computer Architecture: Opportunities for Nanotechnology. *Science*, 280, 1998.

[HMH01] R. Ho, K. Mai, and M. Horowitz. The Future of Wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001.

[HP96] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach, second edition.* Morgan Kaufmann, 1996.

[HT01] Nevin Heintze and Olivier Tardieu. Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second. In *ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, 2001.

[JS96] R. Johnson and M. Schlansker. Analysis Techniques for Predicated Code. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '96).* Hewlett-Packard Co., Palo Alto, CA, USA, 1996.

[KK95] George Karypis and Vipin Kumar. Multilevel Graph Partitioning and Sparse Matrix Ordering. In *Proceedings of the 1995 Intl. Conference on Parallel Processing*, 1995.

[KS86] S. R. Kunkel and J. E. Smith. Optimal Pipelining in Supercomputers. In *Proceeding of the 13th Symposium on Computer Architecture*, pages 404–414, 1986.

[LBF+98] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. *ACM SIGPLAN Notices*, 33 (11):46–57, 1998.

[LCD+00] 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.

[LPMS97] 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.

[LS93] I. Lemke and G. Sander. Visualization of Compiler Graphs. Technical Report Design report D 3.12.1-1, USAAR-1025-visual, ESPRIT Project #5399 Compare, Universität des Saarlandes, 1993.

[Man95] R.T. "Tets" Maniwa. Global Distribution: Clocks and Power. *Integrated System Design Magazine*, 1995.

[MH00] Tsutomu Maruyama and Tsutomu Hoshino. A C to HDL Compiler for Pipeline Processing on FPGAs. In *2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

[MLC+92] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. 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.

[MN99] S. Mantripragada and A. Nicolau. *Innovative Architecture for Future Generation High-Performance Processors and Systems*, chapter The effects of Predicated Execution on Architectures Supporting Dynamic Speculation, pages 37–45. Silicon Graphics Inc., CA, July 1999.

[MRS+01] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth Sensitive Code Generation in a Custom Embedded Accelerator Design System. In *Proceedings of the 5th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2001)*, St. Goar, Germany, March 2001.

[MTH99] Tsutomu Maruyama, Masaaki Takagi, and Tsutomu Hoshino. Hardware Implementation Techniques for Recursive Calls and Loops. In *9th International Workshop on Field-Programmable Logic and Applications*, pages 450–455, 1999.

[Pap88] Gregory Michael Papadopoulos. Implementation of a General Purpose Dataflow Multiprocessor. Technical Report MIT/LCS/TR-432, Laboratory for Computer Science, Massachusetts Institute of Technology, 1988.

[PBJ+91] Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. Dependence Flow Graphs: An Algebraic Approach to Program Dependencies. In SIGPLAN, editor, *In Principles of Programming Languages*, volume Volume 18, 1991.

[PDN99] Preeti Ranjan Panda, Nikil Dutt, and Alexandru Nicolau. *Memory Issues in Embedded Systems-On-Chip – Optimizations and Exploration.* Kluwer Academic Publishers, 1999.

[RD97] Martin C. Rinard and Pedro C. Diniz. Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers. *ACM Transactions on Programming Languages and Systems*, 19(6):942–991, 1997.

[RLG+98] C. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. Arnold, and M. Gokhale. The NAPA Adaptive Processing Architecture. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, April 1998.

[RR00] Radu Rugina and Martin Rinard. Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, 2000.

[SAmWH00] John W. Sias, David I. August, and Wen mei W. Hwu. Accurate and Efficient Predicate Analysis with Binary Decision Diagrams. In *Proceedings of the 33rd International Symposium on Microrchitecture*, December 2000.

[SBA00] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth Analysis with Application to Silicon Compilation. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, 2000.

[Sch97] Herman Schmit. Incremental Reconfiguration for Pipelined Applications. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 47–55, 1997.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. Technical report, Digital Equipment Corporation Western Research Laboratory, 1994.

[SG99] L. Spainhower and T. A. Gregg. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. *IBM Journal of Research and Development*, 43, No. 5/6, 1999.

[SH97] M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-to Analysis. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, January 1997.

[SNBK01] K. Sankaralingam, R. Nagarajan, D.C. Burger, and S.W. Keckler. A Technology-Scalable Architecture for Fast Clocks and High ILP. In *5th Workshop on the Interaction of Compilers and Computer Architecture*, January 2001.

[Soh97] Guri Sohi. Computing With Billion Transistor Chips. ftp://ftp.cs.wisc.edu/sohi/papers/1997/sohi.billion.ps.gz, 1997.

[Spi96] Daniel A. Spielman. Highly Fault-Tolerant Parallel Computation. In *Proceedings of the 37th Annual IEEE Conference on Foundations of Computer Science 1996*, 1996.

[SSM01] Luc Séméria, Koichi Sato, and Giovanni De Micheli. Synthesis of Hardware Models in C with Pointers and Complex Data Structures. *IEEE trans. on VLSI*, 2001.

[Sta95] Standard Performance Evaluation Corp. *SPEC CPU95 Benchmark Suite*, 1995.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, 1996.

[SV98] S. M. Scalera and J. R. Vazquez. The Design and Implementation of a Context Switching FPGA. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1998.

[SWP86] J. E. Smith, S. Weiss, and N.Y. Pang. A Simulation Study of Decoupled Architecture Computers. In *IEEE Computer*, volume 35 (8), pages 692–702, August 1986.

[TCJW97] Steve Trimberger, Dean Carberry, Anders Johnson, and Jennifer Wong. A Time-Multiplexed FPGA. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 22–28, Napa, CA, April 1997.

[Tra86] Kenneth R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report MIT-LCS-TR-370, MIT, August 1986.

[WBG00] Kip Walker, Mihai Budiu, and Seth Copen Goldstein. Interfacing Reconfigurable Logic with a CPU. In *2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 317–318, 2000.

[WFW+94] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. In *ACM SIGPLAN Notices*, volume 29, pages 31–37, December 1994.

[WL95] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, 1995.

[Xil96] Xilinx Inc. XC6200: Advance Product Specification, 1996.