Parallelizing Applications into Silicon

Jonathan Babb, Martin Rinard, Csaba Andras Moritz, Walter Lee,
Matthew Frank, Rajeev Barua, and Saman Amarasinghe
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
jbabb@lcs.mit.edu

Abstract

The next decade of computing will be dominated by embedded systems, information appliances and application-specific computers. In order to build these systems, designers will need high-level compilation and CAD tools that generate architectures that effectively meet the needs of each application. In this paper we present a novel compilation system that allows sequential programs, written in C or FORTRAN, to be compiled directly into custom silicon or reconfigurable architectures. This capability is also interesting because trends in computer architecture are moving towards more reconfigurable hardware-like substrates, such as FPGA based systems. Our system works by successfully combining two resource-efficient computing disciplines: *Small Memories* and *Virtual Wires*.

For a given application, the compiler first analyzes the memory access patterns of pointers and arrays in the program and constructs a partitioned memory system made up of many small memories. The computation is implemented by active computing elements that are spatially distributed within the memory array. A space-time scheduler assigns instructions to the computing elements in a way that maximizes locality and minimizes physical communication distance. It also generates an efficient static schedule for the interconnect. Finally, specialized hardware for the resulting schedule of memory accesses, wires, and computation is generated as a multi-process state machine in synthesizable Verilog.

With this system, implemented as a set of SUIF compiler passes, we have successfully compiled programs into hardware and achieve *specialization* performance enhancements by up to an order of magnitude versus a single general-purpose processor. We also achieve additional *parallelization* speedups similar to those obtainable using a tightly-interconnected multiprocessor.

1 Introduction

When the performance, cost, or power requirements of an application cannot be met with a commercial off-the-shelf processor, engineers typically turn to custom and/or reconfigurable hardware. Performance oriented hardware projects have generated excellent results: IBM's Deep Blue chess computer recently defeated Garry Kasparov, the World Chess Champion; the Electronic Frontier Foundation's DES cracking machine recently cracked the encryption standard of our banking systems in less than a week. Low-power and low-cost *embedded* applications in cars and hand-held devices have indirectly become an everyday part of human existence. Similarly, information appliances promise to be one of the dominating applications of the next decade.

All of these hardware systems perform well, in part, because they exploit large amounts of concurrency and specialization. Currently, hardware architects exploit these aspects by hand. But as more and more complex applications are mapped into hardware, the difficulty of exploiting concurrency and specialization by hand makes it more and more difficult to design, develop and debug hardware. This paper demonstrates a parallelizing compiler that automatically maps applications directly into a hardware implementation. We expect that the combination of such high-level design tools with the added capability of reconfigurable hardware substrates (e.g. FPGA-based systems), will enable hardware designers to develop application-specific systems that can achieve unprecedented price/performance and energy efficiency. This paper explains how such a high-level compilation system works.

For this sytem to generated efficient computing structures, we need to make two fundamental attitude changes: 1) A shift of focus from computation alone to also including memory, and wires; and 2) A shift of focus from a low-level, hardware-centric design process to a high-level, software-based development process.

The first shift is driven by the key performance trend in VLSI: the performance of basic arithmetic units, such as adders and multipliers, is increasing much more rapidly than the performance of memories and wires. Wire lengths affect the performance in that the clock cycle time must be long enough for signals to propagate the length of the longest wire. As designers exploit feature size decreases to put more functionality on a chip, the wire lengths become the limiting factor in determining cycle time. It is therefore crucial to deliver designs sensitive to wire length. We therefore choose to organize the compiler to optimize the memories and wires, as well as distributing the computation, in order to achieve good overall performance.

The second shift is driven by both VLSI and the complexity of applications we can now implement in hardware. In the past decade, hardware design languages such as Verilog and VHDL enabled a dramatic increase in the sophistication of the circuits that designers were able to build. But these languages still require designers to specify the low-level hardware components and schedule the operations in the hardware on a cycle-by-cycle basis. We believe that these languages present such low-level abstractions that they are no longer adequate for the large, sophisticated systems that are now possible to implement in hardware. We believe that the next revolution in hardware design will come from compilers that are capable of implementing programs written in a high-level language, such as C, directly into hardware.

While these shifts may at first appear to require a radically different approach, our results indicate that the solution lies in the extension of known techniques from parallel computing and parallelizing compilers. Specifically, we believe that performance gains will be made possible by splitting large, relatively slow memories into arrays of small, fast memories, and by statically scheduling the interconnection between these memories. In this paper computation is implemented by custom logic interspersed among these memories. However, our approach for managing memory and wires is applicable whether the interspersed computation is implemented in reconfigurable hardware, configurable arithmetic arrays, or even small microprocessors.

The direct application of the research presented in this paper will allow programmers to design complex distributed-memory hardware which is correct-by-construction. That is, if the applications can be successfully executed and tested on a sequential processor, our compiler will generate functionally correct hardware.

In the remainder of this paper we will show, step-by-step, the basic compiler transformations needed to generate a distributed-memory hardware design from a sequential program. This design will contain hardware and memory structures carefully matched to the program requirements. At the end, we will present initial favorable results from a working compiler which automatically implements the set of transformations we describe.

The organization is as follows. Section 2 motivates our compilation strategy. Section 3 overviews the compilation process and introduces a running example subsequently used for illustration. Section 4 then outlines two analyses we use for decomposing the application data into multiple clusters of small memories. Next, Section 5 describes how application instructions are assigned to memory clusters and how a space-time scheduler creates virtual interconnections between clusters. Section 6 shows how we generate hardware from the resulting schedule. Section 7 then presents both specialization and parallelization results for several benchmark applications. Finally, Section 8 describes related work in the field, and Section 9 makes concluding remarks.

2 Motivation

When compiling applications to hardware, we consider three basic primitives: memory, wires, and logic. The following sections describe and motivate our compilation approach for each primitive.

2.1 Small Memories

Our efficient memory architecture contains many small memories, with the computation distributed among the memories to maximize parallelism while minimizing the physical communication distance. This organization has significant benefits when compared with a standard architecture, which segregates the memory from the computation. First, a system of small memories have shorter effective memory access latencies. A small memory can produce a value quicker than a large memory can; once the value is produced, it is closer to the point where it will be consumed. Multiple small memories also have a higher aggregate memory bandwidth than a single large memory, and each reference takes less power because the wire capacitance in small memories is less than the capacitance in a large memory.

Compiler technology is an important element in exploiting the potential of small memories. If the compiler can statically determine the memory location that will satisfy each data reference, it can then generate more efficient computation structures. Also, the compiler can leverage the correspondence between computation and memory to place the computation close to the memory that it accesses. When possible, static disambiguation also allows the compiler to statically schedule the memory and communication wires, eliminating the need to implement an expensive dynamic interconnect capable of delivering data from any memory to any piece of computation. Finally, static disambiguation allows the compiler to simplify address calculations. The bits in the address that would otherwise select the memory bank disappear, replaced by directly wired connections.

In our hardware implementation of small memories, we cluster one or more memories together into directly-connected tiles arranged in a space-efficient interconnect. Roughly speaking, a *tile* is a hardware region within which wire delay is less than one clock cycle. For simplicity, we also choose to correlate tiles with sequencing control, with one sequencer per tile.

Besides memory banks, each tile contains custom logic as well as the ability to schedule communication between neighbors. Unlike traditional memory banks, which serve as sub-units of a centralized memory system, the memory banks in each tile function autonomously and are directly addressable by the custom logic, without going through a layer of arbitration logic. This organization enables memory ports which scale with the hardware size. While we will focus on custom hardware generation throughout this paper, this approach is generally application to reconfigurable systems as well.

2.2 Virtual Wires

Our architectures are designed specifically to keep all wire lengths within a fixed small size. This mechanism allows the compiler to tightly control the constraints that wire lengths place on the clock cycle time. To maximize performance, the compiler uses a spatially-aware partitioner that maps the data and computation onto specific physical locations within the memory array. The compiler heuristically minimizes the wire length between communicating components.

Of course, applications also require communication between remote regions of the chip. The architecture implements such communications by implementing a long "virtual wire" out of multiple short physical wires. Conceptually, a virtual wire is a pipelined connection between the endpoints of the wire. This connection takes several clock cycles to traverse, but allows pipelined communication by occupying only one short physical wire in each clock cycle.

Virtual wires are a key concept for a compiler that optimizes wire performance. Our compiler generates architectures that connect the small memories using virtual wires; short physical wires are statically scheduled for maximum efficiency and minimum hardware interconnect overhead.

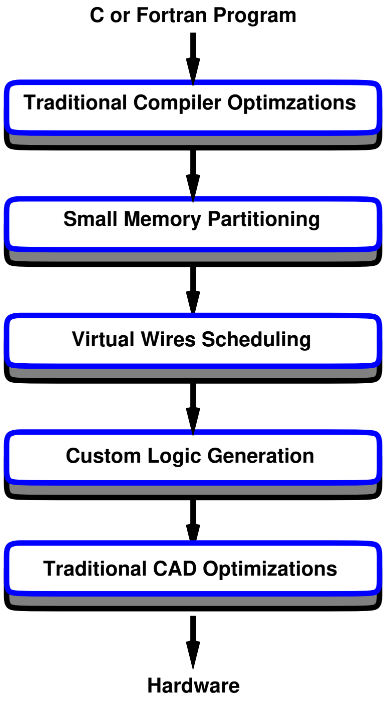


Figure 1: Compiler Flow

2.3 Custom Logic

To maximize the efficiency of the final circuit, our compiler implements the computation using custom logic. This logic consists of a datapath and finite state control. Because this logic is customized for the specific computation at hand, it can be smaller, consume less power, and execute faster than more general purpose implementations. Examples of specific optimizations include arithmetic units optimized for the specific operations in the program and reductions in datapath widths for data items with small bitwidths. The use of a small finite state control eliminates the area, latency, and power consumption of general-purpose hardware designed to execute a stream of instructions.

Conceptually, the compiler generates hardware resources such as adders for each instruction in the program. In practice, the compiler reuses resources across clock cycles: if two instructions require the same set of resources but execute in different cycles, the compiler uses the same resource for both instructions. This approach preserves the benefits of specialized hardware while generating efficient circuits.

While generating custom logic provides the maximum performance, our approach is also effective for other implementation technologies. Specifically, the compiler could also map the computation effectively onto a platform with reconfigurable logic or programmable ALUs instead of custom logic.

3 Compilation Overview

Our compilation system is responsible for translating an application program into a parallel architecture specialized for that application. To make this translation feasible, our compilation system incorporates both the latest code optimization and parallelization techniques as well as state-of-the-art hardware synthesis technology. Figure 1 shows the overall compiler flow. After reading in the program and performing traditional compiler optimizations, the compiler performs three major sets of transformations. Each set of transformations handles one class of primitives: memory, wires, or logic. Following these transformation, traditional computer-aided-design (CAD) optimization can be applied to generate the final distributed-memory hardware.

Phase ordering is determined by the relative importance of each primitive in determining overall application performance. First, we analyze and partition program data into memories, and assign computation to those memories to create. We then schedule communication onto wires. Finally, we generate custom logic to perform all the computation. The following sections describe each component in turn.

Figure 2 introduces a running example which illustrates the steps the smart memory compiler and synthesizer perform. Figure 2(a) shows the initial code and data fed to the compiler. The code is a simple *for* loop containing affine references to two arrays, A and B. Both the data arrays are initially mapped to the same monolithic memory bank. Subsequent sections will discuss how this program is transformed by smart memory compilation, scheduling and hardware generation.

4 Small Memory Formation

The first phase of our compilation system is small memory formation. This phase consists of two main steps. First, the total memory required per application is decomposed into smaller memories based on the application's data access pattern. Second, the computation is assigned to the appropriate memories partitions.

4.1 Memory decomposition

The goal of memory decomposition is to partition the program data across many small memories, but yet enforce the attribute that each program data reference can refer to only a single memory bank. Such decomposition enables high aggregate memory bandwidth, keeps the access logic simple, and enables locality between a memory and the computations which access it.

Our memory decomposition problem is similar to that for Raw architectures [20]. The Raw compiler uses *static promotion* techniques to divide the program data across the tile in such a way that the location of the data for each access is known at compile time [2, 3]. Our memory compiler leverages the same techqniues. At the coarse level, different objects may be allocated to different memories through *equivalence class unification*. At the fine level, each array is potentially distributed across several memories through the aid of *modulo unrolling*. We overview each technique below.

Equivalence Class Unification Equivalence class unification [2] is a static promotion technique which uses the pointer analysis to collect groups of objects, each of which has to be mapped to a single memory. The compiler uses SPAN [15], a sophisticated flow-sensitive, context-sensitive interprocedural pointer analysis package. Pointer analysis is used to provide *location set lists* for every pointer in the program, where every location set corresponds to an abstract data object in the program. A pointer's location set list

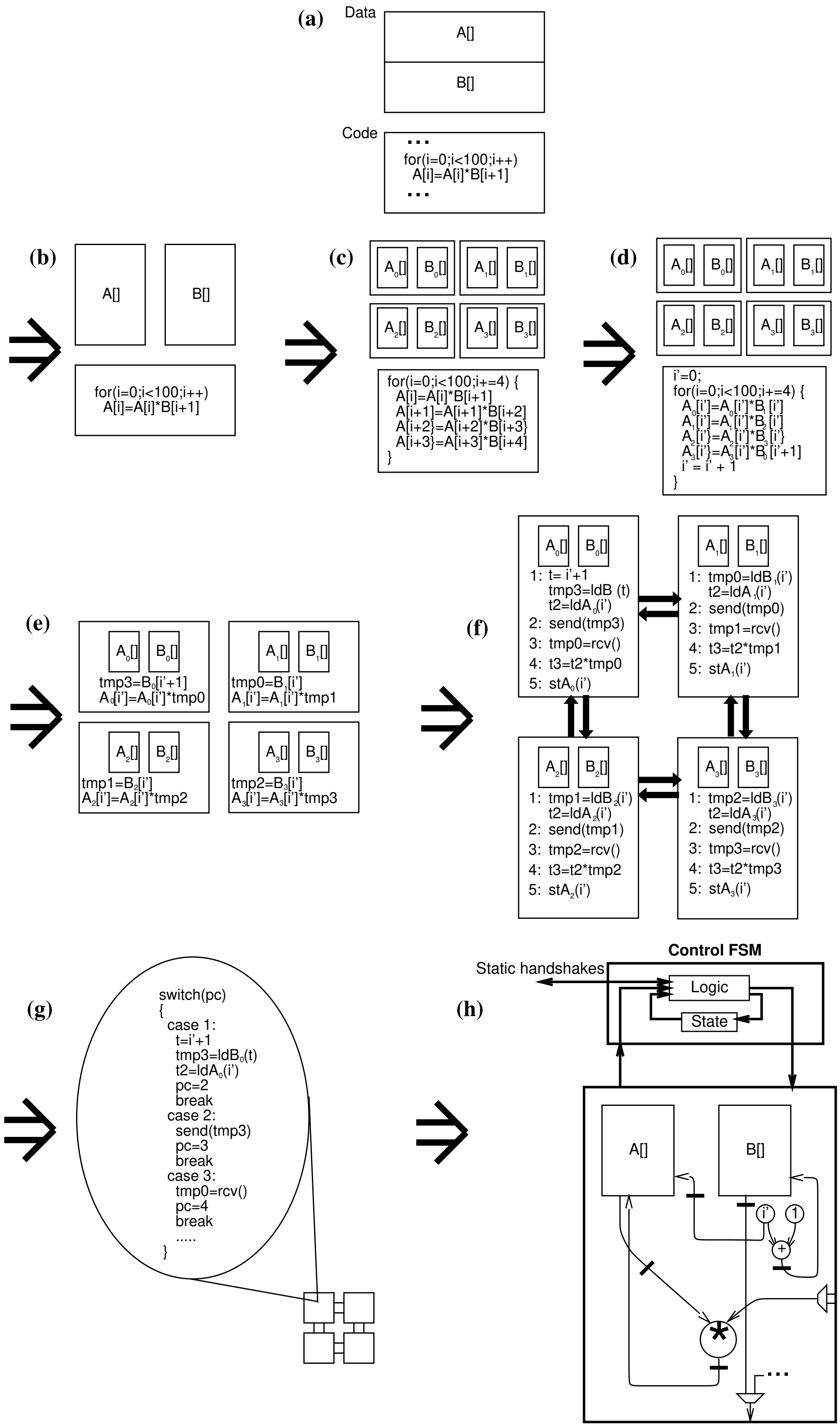


Figure 2: Example. (a) initial program; (b) after equivalence class specialization; (c) after modulo unrolling step 1; (d) after modulo unrolling step 2; (e) after migration of computation to memories; (f) after scheduling of interconnect between memories; (g) after state machine generation; (h) resulting hardware architecture.

is a list of abstract data objects to which it can reference. We use this information to derive *alias equivalence classes*, which are groups of pointers related through their location set lists. [1] Pointers in the same alias equivalence class can potentially alias to the same object, while pointers in different equivalence classes can never reference the same object.

Once the alias equivalence classes are determined, equivalence class unification places all objects for an alias class onto a single memory. This placement ensures that all pointers to that alias class refer to one memory. By mapping objects for every alias equivalence class in such a manner, all memory references can be constrained to addressing a single memory bank. By mapping different alias equivalence classes to different banks, memory parallelism can be attained.

Figure 2(b) shows the results of equivalence class unification on the initial code in Figure 2(a). Since no static reference in the program can address both $A[]$ and $B[]$, pointer analysis determines that $A[]$ and $B[]$ are in different alias equivalence class. This analysis allows the two arrays to be mapped to different memories, while ensuring that each memory reference only addresses a single memory.

Modulo Unrolling The major limitation of equivalence class unification is that arrays are treated as single objects that belong to a single alias equivalence class. Mapping an entire array to a single memory sequentializes accesses to that array and destroys the parallelism found in many loops. Therefore, we use a more advanced strategy called modulo unrolling [3] to allow arrays to be distributed across different memories.

Modulo unrolling applies for array accesses whose indices are affine functions of enclosing loop induction variables. First, arrays are partitioned into smaller memories through *low-order interleaving*. In this scheme, consecutive elements of the array are interleaved in a round-robin manner across memory banks on successive tiles. Next, modulo unrolling shows that it is always possible to unroll loops by certain factors such that in the resulting code all affine accesses go to only one of the smaller memories. In certain cases, additional transformations may be required. One feature of modulo unrolling is that it does not affect the disambiguation of accesses to any other equivalence class. Details on determining the symbolic derivations of the minimum unroll factors required, and the code generation techniques needed, are given in [3].

Figure 2(c) shows the result of low-order interleaving and unrolling phases of modulo unrolling on the code in Figure 2(b) when the number of desired memories is four. Low order interleaving splits each array into four sub-arrays whose sizes are a quarter of the original. Modulo unrolling uses symbolic formulas to predict that the unroll factor required for static dissambiguation is four. Now we can see that each reference always goes to one tile. Specifically, the A[i], A[i+1], A[i+2] and A[i+3] references access sub-arrays 0, 1, 2 and 3. The B[i+1], B[i+2], B[i+3] and B[i+4] references access sub-arrays 1, 2, 3 and 0. Figure 2(d) shows the code after the code generation required by modulo unrolling. The array references have been converted to references to the partitioned sub-arrays, with appropriate updated indices.

[1] More formally, alias equivalence classes are the connected components of a graph whose nodes are pointers and whose edges are between nodes whose location set lists have at least one common element.

4.2 Computation Assignment

The previous memory partitioning phase has decomposed program data structures into separate memories, with groups of memories clustered together into tiles. We next locate computation close to the most appropriate memory. The load and store instructions that access a memory are always assigned to the tile containing that memory. The remaining computation can be assigned to any tile; its actual assignment is selected based on two factors: minimize communication between tiles and minimizing the latency of the critical path.

Our algorithm for assigning computation to memory tiles directly leverages the non-uniform resource architecure (NURA) algorithms developed for the Raw Machine [11]. In this algorithm, instruction-level parallelism within a basic block is orchestrated across multiple tiles. This work is in turn an extension of the original MIT Virtual Wires Project [1] work, in which only circuit-level, or *combinational*, parallelism with a clock cycle was orchestrated across multiple tiles.

The compiler performs computation assignment in three steps: clustering, merging, and placement. Clustering groups together instructions, such that instructions within a cluster have no parallelism that can profitably be exploited across tiles given the cost of communication. Merging merges the cluster to reduce the number of clusters down to the number of tiles. Placement performs a bijective mapping from the merged clusters to tiles, taking into account the topology of the interconnect.

Figure 2(e) shows the results of computation assignment in our example. As we described in the previous section, each tile contains two memories, one for each array. Computation is assigned directly into the small memory array. In the figure, each tile has been assigned a subset of the original instructions.

5 Virtual Wires Scheduling

While the previous phase of the compiler produces the small memories, this next phase of the compiler is responsible for managing inter-tile communication. We term this phrase *Virtual Wires Scheduling*.

Although moving program instruction into the memory system eliminates long memory communication latencies in comparison to monolithic processor design, new inter-tile communication paths will now be required for the program to execute correctly. Namely, there are data dependencies between the instructions assigned to each tile.

In Figure 2(e) the data dependences between tiles are explicitly shown as temporary variables introduced in the code. For example, *tmp*0 needs to be communicated between the upper-left tile and the upper-right tile. Besides data dependencies, there are also control dependencies introduced. However, this control flow between basic blocks is explicitly orchestrated by the compiler through *asynchronous global branching* [11], an asynchronous mechanism for implementing branching across all the tiles using static communication and individual branches on each tile. Thus control dependencies will be turned into data dependencies as well.

In the same manner as the original virtual wires scheduling algorithm for logic emulation [1], our static scheduler will multiplex logical communications such as *tmp*0 with other communications between the same source and destination, sharing the same physical channel. The multiplexing of this physical channel will be directly synthesized in the target hardware technology by a following hardware generation phase. Note that communication required between non-neighbor tiles must be routed through an intermediate tile.

In contrast to the Raw space-time scheduler, which targets a simple processor and static switch in each tile, our compilation target is more flexible, allowing multiple instructions to be executed in parallel, with the only constraint being that each memory bank within each tile can be accessed at most once per cycle. This additional flexibility is possible because we are targeting custom hardware instead of a fixed processing unit.

In constrast to compilation for a VLIW, which also involves statically scheduling multiple instructions, we are not constrained by an initial set of function units, register files, and busses. Our scheduler minimizes the execution latency of the program by scheduling to virtual function units as dictated by the application.

As a final note, during scheduling we have essentially relaxed the resource allocation problem for both registers and function units, while retaining the constraints for memory accesses and inter-tile wires. This strategy is similar to virtual register allocation strategies for sequential processors, except that we relax all computation resource constraints. Because we are parallelizing sequential code and thus predominately limited by memory and communication costs, this relaxation is feasible. Resource allocation is delayed until the later custom logic generation phase.

Figure 2(f) shows the results of communication scheduling. The send instruction in each tile represents multiplexed communications across the inter-tile channels. The following custom logic generation phase will convert these instructions to pipeline registers and multiplexers controlled by bits in the state machine in each tile.

6 Custom Logic Generation

By the time the custom logic generation phase executes, the previous phases have mapped the data onto the memories, extracted the concurrency, and generated a parallel thread of instructions for each tile. They have also scheduled the execution: each memory access, communication operation and instruction has been mapped to a specific execution cycle. Finally, the space-time scheduler has mapped the computation and communication to a specific location within the memory array.

The custom logic generation phase is responsible for generating a hardware implementation of the communication and computation schedules. First the system generates a finite state machine that produces the cycle-by-cycle controls for all the functional units, registers and memories that make up the tile. Then the finite state machine is synthesized into a technology independent register-transfer-level specification of the final circuit. We discuss each of these activities below.

6.1 Finite State Machine

As shown in Figure 2(g), the finite state machine (FSM) generator takes the thread of scheduled instructions that has been generated for each tile, and produces a state machine for each thread. This state machine contains a state register, called `pc`, which serves a function similar to the program counter in a conventional processor. Each state of the FSM corresponds to the work that will be done in a single cycle in the resulting hardware. The resulting FSM will generate both the proper control to calculate the next state, including any calculation to generate branch destinations, and also the proper control signals to the virtual functional units that will be generated in the next phase.

Each state of the FSM contains a set of possibly dependent operations. The FSM generator turns the set of operations in each state into combinational logic. Any dependences between operations in the same state are connected by specifying wires between the dependent functions. For any data values that are *live* between states (produced in one state and consumed in a different state), the FSM generator allocates a register to hold that value.

6.2 Register Transfer Level

The final phase of compilation involves generating a Register Transfer Level (RTL) model from the finite state machine. The resulting RTL specification for each tile is technology-independent and application-specific. Each tile contains a datapath of inter-connected functional units, several memories, and control elements. Figure 2(h) shows a logic diagram for the circuit that would be generated from the FSM. The datapath is synthesized out of registers (represented by solid black bars in the figure), higher order functional units (such as adders and multipliers) and multiplexers. A controller is specified in RTL to control the actions of the datapath and the inter-processor I/O.

The next step synthesizes logic for the address, data, and enable signals for each memory reference in the FSM. For example the address signals are shown as arrows going into the `A[]` and `B[]` memories in Figure 2(h). Whenever a reference has been statically mapped to a specific memory, the address calculation is specialized to eliminate the bits that select that memory as the target, as described further in Section 6.3. This specialization simplifies the address calculation hardware, so it consumes less power and less area.

The compiler-generated architecture must include communication channels between each FSM for inter-tile communication. The communication synthesis step creates these channels from the inter-tile communication dependencies and synthesizes the appropriate control logic into the FSM. This logic is shown as the multiplexors on the bottom and right hand side of Figure 2(h). In addition, the control logic includes static handshaking signals between each pair of communicating FSMs as shown in the upper left corner of Figure 2(h).

Finally, the compiler outputs technology independent Verilog RTL. This step is a modification of the Stanford VeriSUIF Verilog output passes. SUIF data structures are first converted into Verilog data structures and then written as Verilog RTL to an output file. The output fully describes the cycle-level behavior of the circuit, including all state transitions and actions to be take every cycle.

We have described the steps that are required to compile a program into hardware. In the next section we describe some of the additional optimization opportunities that are available when producing hardware.

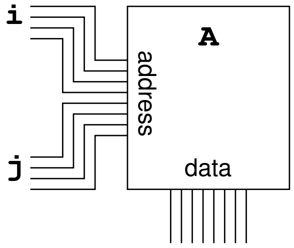


Figure 3: Example of address specialization. The address calculation for the array reference `A[i][j]` can be entirely eliminated. Instead, the values of `i` and `j` are placed on the corresponding address wires.

6.3 Other Custom Logic

Along with the parallelism that is available in custom logic, an additional advantage is that when the bit-level operations are exposed to the compiler, the compiler can perform additional optimizations beyond those found by traditional software optimizers. These include the possibility of folding constants directly into the circuitry and using other compile-time knowledge to completely eliminate gates.

Address Calculation. The primary example of hardware specialization involves simplification of address calculation. As an example, consider what would happen if the code in Figure 2 had referenced a two dimensional array, like `A[i][j]`. The naive code for this address calculation would be `A + (i * Y_DIM + j) * 4`. In the case that `Y_DIM` was a power of 2, most compilers would strength reduce the multiplication operations to shifts to produce `A + (i << LOG2_Y_DIM + j) << 2`.

In most software systems this is the minimal calculation to reference a two dimensional array. In hardware systems, on the other hand, we can specialize even further. First, since we perform equivalence class unification, as described in Section 4.1, the `A` array will live in its own private memory. Then every reference will lie at an offset from location 0 of the memory for array `A`, so the first addition operation can be eliminated. Furthermore, the memory for `A` can be built to be of the same width as the data word, so that the final multiplication by 4 can be completely eliminated, leaving `(i << LOG2_Y_DIM + j)`.

While in a conventional RISC processor logical operations require the same number of cycles as an addition operation, in hardware a `logical or` operation is both faster and smaller than an adder. Our compiler performs *bit-width optimization* to transform these `add` operations into `or` operations. The compiler calculates the maximum bit-width of `j`, and finds that since `j < Y_DIM`, the maximum value of `j` never consumes more than `LOG_Y_DIM` bits, so the bits of `j` and the bits of `(i << LOG2_Y_DIM)` never overlap. The addition in this expression can then be optimized away to produce `(i << LOG_Y_DIM | j)`. Finally, since `oring` anything with 0 produces that value, the `or` gates can be eliminated, and replaced with wires. The result is that the two values, `i` and `j`, can be concatenated to form the final address. This final transformation is shown in Figure 3.

Floating Point. For each floating point operation in the program the compiler replaces the operation with a set of simpler integer and bit level micro-operations using a technique similar to [4]. These resulting micro-operations can then be optimized and scheduled by the datapath generator. Because the constituent exponent and mantissa micro-operations are exposed to the compiler, the compiler can exploit the parallelism both inside individual floating point operations and also between different floating point operations.

As an example of the specialization that can occur, consider dividing a floating point number with a constant that can be written as a factor of two, *e.g.* $y = x/2.0$. Executing this division operation would take many cycles on a traditional floating-point execution unit.

However, by exposing the micro-operations required to do the floating-point computation in terms of operations on the exponent and mantissa and generating specialized hardware for the computation, the cycle count of this operation can be optimized by an order of magnitude. The required micro-operation to perform such a division is to subtract 1 from the exponent of x. The time to execute the floating-point division operation reduces to the latency of a single fixed-point subtraction.

7 Experimental Results

We have implemented a compiler that is capable of accepting a sequential application, written in either C or Fortran, and automatically generating a specialized architecture for that application. The compiler contains all of the functionality described in the previous sections of this paper.

This section presents experimental results for an initial set of applications that we have compiled to hardware. For each application, our compiler produces an architecture description in RTL Verilog. We further synthesize this architecture to logical gates with a commercial CAD tool. The current target technology in which we report gate counts is the reference technology provided with the IKOS VirtuaLogic System [9]. This system is a logic emulator that can be used to validate designs of up to one million gates, as well as additional custom memories.

Our current timing model, enforced during scheduling, limits the amount of computation that may be assigned to any one clock cycle to be less than the equivalent delay of a 32-bit addition or the latency of a small memory access. The exact clock period of each circuit will be technology-dependent, but due to this constraint will be similar to that of a simple processor implemented in the same technology. We report execution times as total cycle counts for each application.

Table 1 gives the characteristics of the benchmarks used for the evaluation. These applications include one traditional scientific program and three multimedia applications. The input programs are all sequential. Jacobi is an integer version of a iterative relaxation algorithm. Adpcm-encode is the compression part of the compression/decompression pair in Adpcm. MPEG-kernel is the portion of MPEG which takes up 70% of the total run-time. SHA implements a secure hash algorithm.

For reference, we also compare our results with previously published simulation results from a parallel Raw machine with one to 16 processors. Because our compiler is based on the same frontend infrastructure for parallelization as the Raw compiler, this comparison allows us to isolate the effect of customizing the tile for an application.

Benchmark	Type	Source	Lines of code	Seq. RT (cycles)	Primary Array size	Description
Jacobi	Dense Mat.	Rawbench	59	2.38M	32×32	Jacobi Relaxation
Adpcm-encode	Multimedia	Mediabench	133	7.1M	1000	Speech compression
MPEG-kernel	Multimedia	UC Berkeley	86	14.6K	32×32	MPEG-1 Video Software Encoder kernel
SHA	Multimedia	Perl Oasis	608	1.0M	512×16	Secure Hash Algorithm

Table 1: Benchmark characteristics. Column *Seq. RT* shows the run-time for the uniprocessor code generated by the Machsuif MIPS compiler.

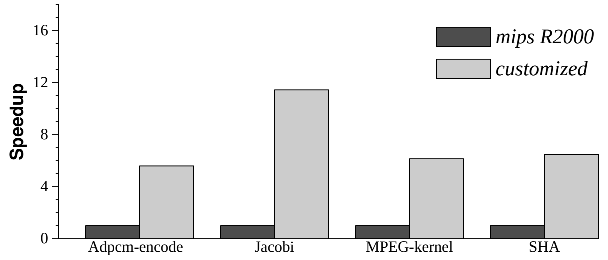


Figure 4: Base comparison

7.1 Base Comparison

Figure 4 presents execution times for each application in comparison with execution time on a single MIPS R2000 processor (the basic component of one Raw tile). Roughly speaking, customization results in an order of magnitude fewer clock cycles (5 to 10) required to perform the same computation. As we expect the clock cycles to be similar, this reduction will translate directly into proportional performance increases for custom hardware.

Table 2 presents the gate counts for the single tile compilation. We assume that one gate is approximately equivalent to one byte of memory for SRAM comparisons. Note that the logic and register gate counts for each application except SHA are smaller than the size of a simple processor (about 20K gates).

Benchmark	Logic	Registers	Memory	Total
Jacobi	3830	5064	65536	74430
Adpcm-encode	3786	3704	3210	10700
MPEG-kernel	1833	2312	3584	7729
SHA	35100	12680	16384	64164

Table 2: Gate counts for one tile.

7.2 Parallelized Comparison

For the parallelized case, we report results for two compilation strategies: a hardwired system and a virtual wired system. The hardwired system allows direct, dedicated wires between arbitrary memories and assumes that signals can propagate the full length of each wire in one clock cycle. The total cycle count is always smaller for the hardwired system because there are no cycles dedicated to scheduling the wires. Additionally, the hardwired systems do have additional hardware synthesized for multiplexing the interconnect. It therefore provides a good comparison point that allows us to isolate the costs of virtual wire scheduling. Bear in mind, however, that for large circuits the hardwired system would contain long wires with large propagation delays. These delays slow down the clock, degrading the overall performance of the application. The scaling factors in future VLSI system will only exacerbate this phenomenon, making virtual wires scheduling a necessity.

Figure 5 presents the resulting speedups for Jacobi and MPEG, the two application which are parallelizable, as we increase the number of tiles. Note that for both cases each tile already has multiple memory banks and takes advantage of ILP even for the single tile case. For both the hardwired and the virtual wired case, performance continues to increase as we add more hardware. The virtual wired case must pay the penalty of communication costs, but nevertheless, absolute performance remains well above even a 16 processor Raw machine. Raw achieves better scalability for Jacobi because of it does not take advantage of initial amount of instruction-level parallelism in the one processor case.

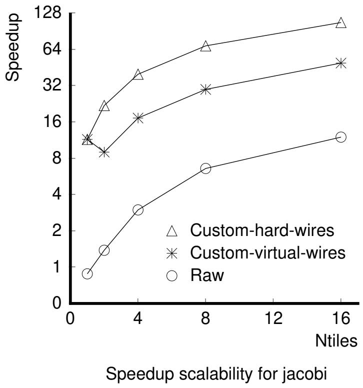
Figure 6 reports the increase in hardware area, including memory, as the number of tiles are increased. Note that for the virtual wires case, the hardware areas grow more rapidly because of the increasing amount of communication logic that is required. In Figure 7, we show the hardware composition, in percentages, of each architecture. Notice how the memory starts out taking a fairly large percentage of the area, but as we decrease the granularity of the memory system by adding more tiles, the system becomes more balanced. As we totally dominate the memory area with additional hardware, the speedup curves tail off.

Implications for Configurable Hardware

Note that while we have not directly considered reconfigurable hardware, such as an FPGA-based system described in the related work, if a reconfigurable system were to implement virtual wires and small memories directly in the fabric, we believe similar performance gains might be achievable. However, we should note that in reconfigurable logic-based architectures the potential clock speeds will most likely *not* be on an equal basis and an appropriate speed penalty for the custom logic will need to be taken into account.

8 Related Work

Silicon compilation has existed in one form or another since the early days of designing chips. Even before schematic capture, designers wrote programs which directly generated the geometry and layout for a particular design. In the past twenty years various research has proposed to compile PASCAL, FORTRAN, C, and Scheme into hardware. We must be careful to distinguish between compiling a program into hardware, and describing hardware with a higher-level



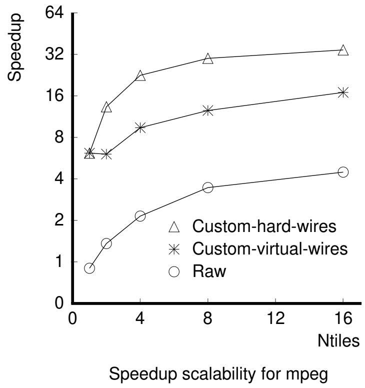
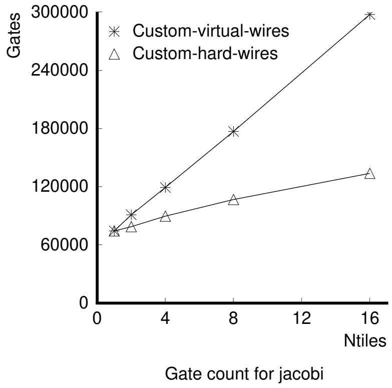


Figure 5: Speedup for jacobi and mpeg.



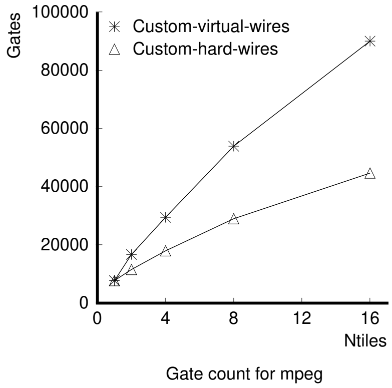
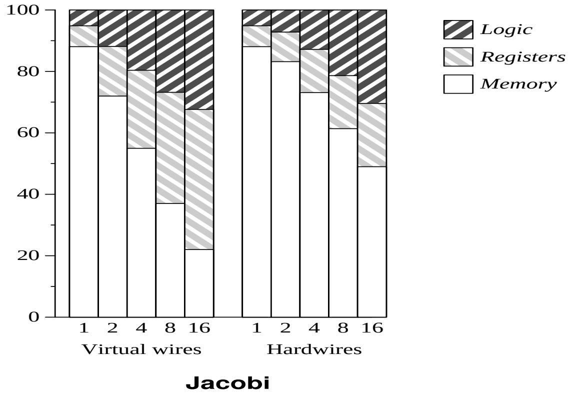


Figure 6: Hardware Size



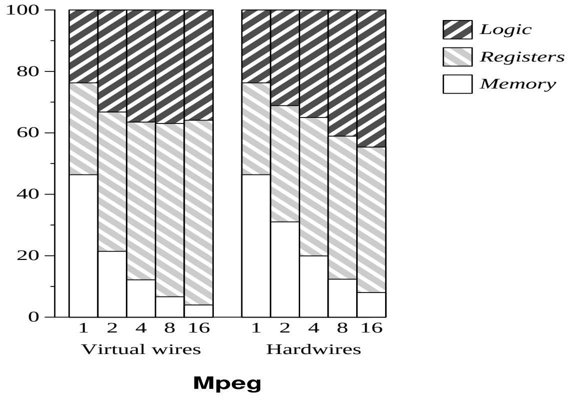


Figure 7: Hardware composition.

language – they are not the same. In our work, we compile sequential programs that describe an *algorithm* in a manner easy for the programmer to understand.

Recent work in RTL and behavioral synthesis [18] involves synthesizing higher-level algorithms into hardware. Where possible, we leverage this work in our compilation process. For example, we leave resource allocation to be performed during the following synthesis pass. However, to our knowledge this is the first system which can manage memory, multiplex wires, and transform general high-level programming languages directly into custom hardware.

We continue by discussing two additional areas of related work: parallel architectures and reconfigurable architectures.

8.1 Parallel Architectures

Other researchers have parallelized some of the benchmarks in this paper. Automatic parallelization has been demonstrated to work well for dense matrix scientific codes [8]. In contrast to this work, our approach to generating parallelism stems from an ability to exploit fine-grain ILP, rather than the coarse-grain parallelism targeted by [8]. Multiprocessors are mostly restricted to such coarse-grain parallelism because of their high costs of communication and synchronization. Unfortunately, finding such coarse grain parallelism often requires whole program analysis by the compiler, which works well only in restricted domains. In a custom architecture, we can successfully exploit ILP because of the register and wire-level latencies in hardware. Of course, hardware can exploit coarse-grain parallelism as well.

Software distributed shared memory schemes on multiprocessors (DSMs) [16] [5] are similar in spirit to our approach for managing memory. They emulate in software the task of cache coherence, one which is traditionally performed by complex hardware. In contrast, this work turns sequential accesses from a single memory image into decentralized accesses across multiple small memories. This technique enables the parallelization of sequential programs into a high bandwidth memory array.

Both the Berkeley IRAM research project [10] and Stanford's new Smart Memories Project [12] focus on building future-generation computing system that are more tightly coupled with memory. The IRAM's approach is to improve performance of the memory system by fitting more data on a chip. They achieve this goal by using high-density dynamic RAM (DRAM) memory instead of lower-density SRAM caches and treating the on-chip DRAM memory as the main memory instead of a redundant copy. The Smart Memories Project's stated purpose is to build a future-generation computing system that provides efficiency, generality, and programmability in a single system.

8.1.1 Reconfigurable Architectures

A reconfigurable computing system, comprised of an array of interconnected Field Programmable Gate Array (FPGA) devices are common hardware targets for application-specific computing. Splash [6] and PAM [19] are the first substantial reconfigurable computing systems. As part of the Splash project, a team lead by Maya Gokhale ported data-parallel C [14] to the Splash reconfigurable architecture. This effort was one of the first to actually compile programs, rather than design hardware, for a reconfigurable architecture. Designs reported to take months to design could be written in a day. While data-parallel C extended the language to handle bit-level operations and systolic communication, all control flow is managed by the host. Hardware compilation was only concerned with basic blocks of parallel instructions. This approach has been ported to National Semiconductors new processor/FPGA chip based on the CLAy architecture [13].

Programmable Active Memories [19], designed at Compaq Paris Research Lab, interfaces to a host processor via memory-mapped I/O. The programming model is to treat the reconfigurable logic as a memory capable of performing computation. The actual design of the configuration for each PAM application was specified in a C-syntax hardware description language.

In the PRISM project [17], functions derived from a subset of C are compiled into an FPGA. The PRISM-I subset included if-then-else as well as for loops of fixed count. The PRISM-II subset included variable length for loops, while, do-while, switch-case, break, and continue.

Other projects include compilation of Ruby [7] - a language of functions and relations, and compilation of vectorizable loops in Modula-2 for reconfigurable architectures [21].

9 Conclusion

In this paper we have described and evaluated a novel compilation system that allows sequential programs written in C and Fortran to be compiled directly into application-specific hardware substrates.

Our approach extends known techniques from parallelizing compilers, such as memory disambiguation, static scheduling and data partitioning. We focus on memory and wires first, and then computation. We start by partitioning the data structures in the program into and array of small, fast memories. In our model, computation is performed by custom logic computing elements interspersed among these memories. The compiler leverages the correspondence between memory and computation to place the computation close to the memory that it accesses, such that communication costs are minimized. Similarly, a static schedule is generated for the interconnect that optimizes wire utilization and minimizes interconnect latency. Finally, the specialized hardware for smart memories, virtual wires, and other custom logic is compiled to hardware in the form of a multi-process state machine in synthesizable Verilog.

With this compilation system we have obtained specialization performance improvements by up to an order of magnitude versus a single general purpose processor and additional parallelization speedups similar to those obtainable using a tightly interconnected multiprocessor.

References

[1] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal. Logic Emulation with Virtual Wires. *IEEE Transactions on Computer Aided Design*, 16(6):609–626, June 1997.

[2] Rajeev Barua, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Maps: A Compiler-Managed Memory System for Raw Machines. Technical

report, M.I.T. LCS-TM-583, July 1998. Also http://www.cag.lcs.mit.edu/raw/.

[3] Rajeev Barua, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Memory Bank Disambiguation using Modulo Unrolling for Raw Machines. In *Proceedings of the ACM/IEEE Fifth Int'l Conference on High Performance Computing(HIPC)*, Dec 1998. Also http://www.cag.lcs.mit.edu/raw/.

[4] William J. Dally. Micro-optimization of floating-point operations. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 283–289, Boston, Massachusetts, April 3–6, 1989.

[5] Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 186–197, Cambridge, Massachusetts, October 1–5, 1996.

[6] Maya Gokhale, William Holmes, Andrew Kopser, Sara Lucas, Ronald Minnich, Douglas Sweeney, and Daniel Lopresti. Building and using a highly parallel programmable logic array. *Computer*, 24(1), January 1991.

[7] Shaori Guo and Wayne Luk. Compiling Ruby into FPGAs. In *Field Programmable Logic and Applications*, August 1995.

[8] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. *COMPUTER*, 29(12):84–89, December 1996.

[9] IKOS Systems, Inc. *VirtuaLogic Emulation System Documentation*, 1996. Version 1.2.

[10] Christoforos E. Kozyrakis, Stylianos Perissakis, David Patterson, Thomas Anderson, Krste Asanovic, Neal Cardwell, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, Randi Thomas, Noah Treuhaft, and Katherine Yelick. Scalable processors in the billion transistors era: IRAM. *IEEE Computer*, pages 75–78, September 1997.

[11] Walter Lee, Rajeev Barua, Matthew Frank, Devabhatuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, San Jose, CA, October 1998.

[12] Mark Horowitz, personal communications. Stanford University Smart Memories Project. http://velox.stanford.edu/smart_memories.

[13] Maya B. Gokhale, Janice M. Stone, Matthew Frank, Sarnoff Corporation. NAPA C: Compiling for a Hybrid RISC/FPGA Architecture. In *FCCM98*, Napa Valley, California, April 1998.

[14] Maya Gokhale and Brian Schott. Data-Parallel C on a Reconfigurable Logic Array. *Journal of Supercomputing*, September 1995.

[15] Radu Rugina and Martin Rinard. Span: A shape and pointer analysis package. Technical report, M.I.T. LCS-TM-581, June 1998.

[16] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Cambridge, Massachusetts, October 1–5, 1996.

[17] A. Smith, M. Wazlowski, L. Agarwal, T. Lee, E. Lam, P. Athans, H. Silverman, and S. Ghosh. PRISM II Compiler and Architecture. In *Proceedings IEEE Workshop on FPGA-based Custom Computing Machines*, pages 9–16, Napa, CA, April 1993. IEEE.

[18] Synopsys, Inc. *Behavioral Compiler User Guide, V 1997.08*, August 1997.

[19] J.E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. H. Touati, and P. Boucard. Programmable Active Memories: Reconfigurable Systems Come of Age. *IEEE Transactions on VLSI Systems*, 4(1), March 1996.

[20] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, September 1997. Also available as MIT-LCS-TR-709.

[21] M. Weinhardt. Compilation and Pipeline Synthesis for Reconfigurable Architectures - High Performance by Configware. In *Reconfigurable Architecture Workshop*, 1997.