Baring it all to Software: The Raw Machine*

Elliot Waingold, Michael Taylor, Vivek Sarkar, Walter Lee, Victor Lee,
Jang Kim, Matthew Frank, Peter Finch, Srikrishna Devabhaktuni,
Rajeev Barua, Jonathan Babb, Saman Amarasinghe, Anant Agarwal

MIT Laboratory for Computer Science
Cambridge, MA 02139

Abstract

Rapid advances in technology force a quest for computer architectures that exploit new opportunities and shed existing mechanisms that do not scale. Current architectures, such as hardware scheduled superscalars, are already hitting performance and complexity limits and cannot be scaled indefinitely. The Reconfigurable Architecture Workstation (Raw) is a simple, wire-efficient architecture that scales with increasing VLSI gate densities and attempts to provide performance that is at least comparable to that provided by scaling an existing architecture, but that can achieve orders of magnitude more performance for applications in which the compiler can discover and statically schedule fine-grain parallelism.

The Raw microprocessor chip comprises a set of replicated tiles, each tile containing a simple RISC like processor, a small amount of configurable logic, and a portion of memory for instructions and data. Each tile has an associated programmable switch which connects the tiles in a wide-channel point-to-point interconnect. The compiler statically schedules multiple streams of computations, with one program counter per tile. The interconnect provides register-to-register communication with very low latency and can also be statically scheduled. The compiler is thus able to schedule instruction-level parallelism across the tiles and exploit the large number of registers and memory ports. Of course, Raw provides backup dynamic support in the form of flow control for situations in which the compiler cannot determine a precise static schedule. The Raw architecture can be viewed as replacing the bus architecture of superscalar processors with a switched interconnect and accomplishing at compile time operations such as register renaming and instruction scheduling.

This paper makes a case for the Raw architecture and provides early results on the plausibility of static compilation for several small benchmarks. We have implemented a prototype Raw processor (called RawLogic) and an associated compilation system by leveraging commercial FPGA based logic emulation technology. RawLogic supports many of the features of the Raw architecture and demonstrates that we can write compilers to statically orchestrate all the communication and computation in multiple threads, and that performance levels 10x-100x over workstations is achievable for many applications.

* Also appears as MIT Laboratory For Computer Science TR-709, March 1997.

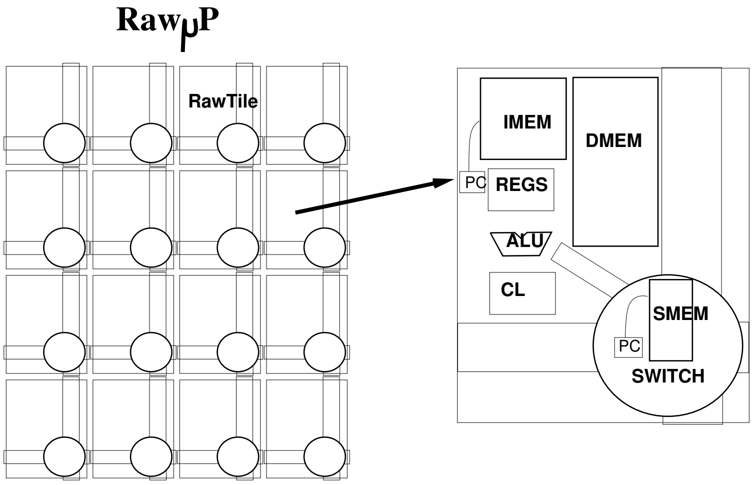

Figure 1: RawμP composition. Each RawμP comprises multiple tiles.

1 Introduction

Rapidly evolving technology places a billion transistors on a chip within reach of the computer architect. Although several approaches to utilizing the large amount of silicon resources have been put forth, they share the single important goal of obtaining the most performance out of approximately one square inch of silicon. All approaches exploit more parallelism in one or more instruction streams, and include more aggressive superscalar designs, simultaneous multithreading, multiprocessors on a chip, VLIWs, and integrating a processor with huge amounts of DRAM memory.

The Raw project's approach to achieving this goal is to implement a simple, highly parallel VLSI architecture, and to fully expose the low-level details of the hardware architecture to the compiler so that the compiler or the software can determine, and implement, the best allocation of resources for each application. As displayed in Figure 1, the Raw microprocessor (RawμP) is composed of a set of interconnected tiles, each tile comprising instruction and data memory, an ALU, registers, some configurable logic, and a programmable switch. Wide channels couple near-neighbor tiles. The hardware architecture is exposed fully to the compiler and the software system, and the switch is programmable via a sequence of instructions, also determined by the compiler. The Raw design uses both distributed register files and distributed memory. Communication between tiles is slightly less expensive than memory access on the same tile, slightly more expensive than register reads and occurs over high handwidth point-to-point channels. Figure 2 shows a multiprocessor system environment for the RawμPs, where each multiprocessor node is constructed out of a RawμP, a dynamic router, and some high-bandwidth RDRAM.

Our approach leverages the same two features that make application-specific custom hardware systems superior in performance for the given application to extant general purpose computers. First, they implement fine-grain communication between large numbers of replicated processing elements and, thereby, are able to exploit huge amounts of fine-grain parallelism in applications, when this parallelism exists. For example, custom vision chips that implement hundreds of small processing tiles interconnected via a point-to-point, near-neighbor interconnect can achieve performance levels far exceeding that of workstations in a cost effective manner. Second, they expose the complete details of the underlying hardware architecture to the software system (be it the software CAD system, the applications software, or the compiler), so the software can carefully orchestrate the

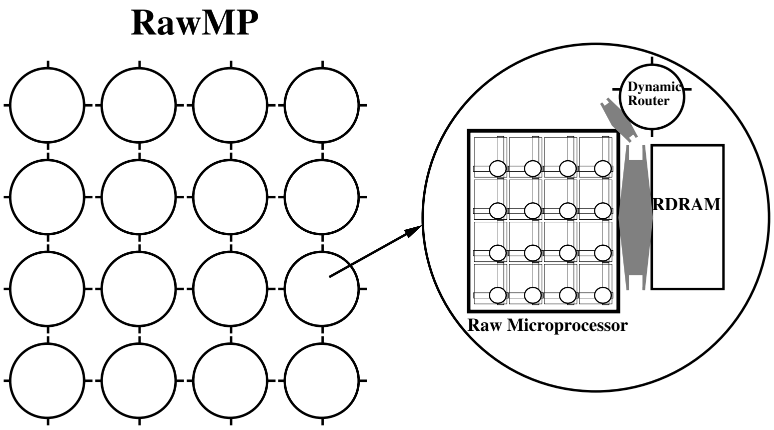

Figure 2: Raw multiprocessor comprising multiple Raw μPs.

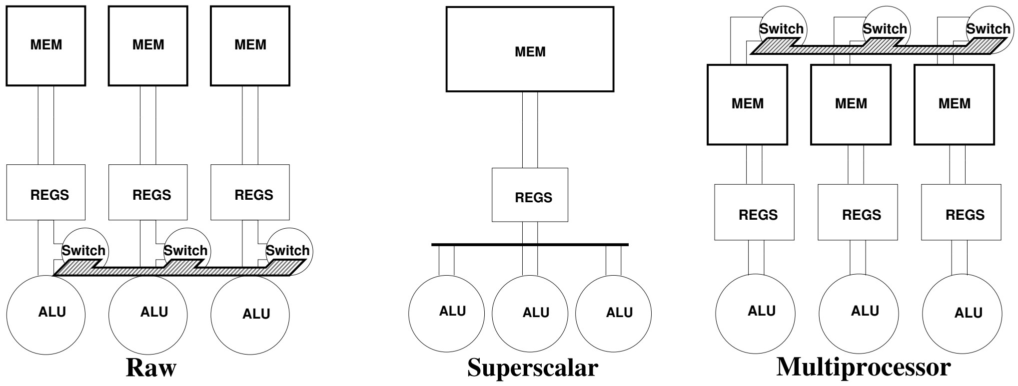

Figure 3: Raw microprocessors versus superscalar processors and multiprocessors. The Raw microprocessor distributes the register file and memory ports and communicates between ALUs on a switched, point-to-point interconnect. In contrast a superscalar contains a single register file and memory port and communicates between ALUs on a global bus. Multiprocessor's communicate at a much coarser grain through the memory subsystem.

execution of the application by applying techniques such as pipelining, synchronization elimination through careful cycle counting, and conflict elimination for shared resources by static scheduling and routing.

In keeping with the general philosophy of building a simple replicated architecture and exposing the underlying implementation to the compiler, Raw has the following features.

Simple replicated tile As shown in Figure 1, the Raw machine comprises an interconnected set of tiles. Each tile contains a simple RISC-like pipeline, and is interconnected over a pipelined, point-to-point network with other tiles. Unlike current superscalar architectures, Raw does not bind specialized logic structures such as register renaming logic, dynamic instruction issue logic, or caching into hardware. Instead, it focuses on keeping each tile small, and maximizes the number of tiles it can implement on a chip, thereby increasing the amount of parallelism it can exploit, and the clock speed it can achieve.

As Figure 3 displays, Raw does not use buses, rather, it uses a switched interconnect. Communication between tiles is pipelined over this interconnect, and occurs at the register level. Thus the clock frequency is higher than in chips that uses global buses. No signal in Raw travels a distance greater than a single tile width within a clock cycle. Furthermore, because the interconnect allows communication between tiles to occur at nearly the same speed as a register read, compilers can use instruction-level parallelism in addition to coarser forms of parallelism to keep the tiles busy.

In addition, memory distributed across the tiles will eliminate the memory bandwidth bottleneck and provide significantly lower latencies to each memory module. Wide channels will allow significantly higher bandwidth for both I/O and on-chip communication. A large number of distributed registers will eliminate the small register name space problem allowing the exploitation of instruction-level parallelism (ILP) to a greater degree. Significantly higher switching speeds internal to a chip compared to off-chip communication and DRAM latencies will allow software sequencing to replace specialized hardware functionality. Software sequencing also exposes the opportunity for customization to applications. As contrasted in Figure 3, with multiprocessors on a chip, inter-tile communication in Raw is at the register level and can be compiler orchestrated, allowing Raw to exploit fine-grain ILP to keep the tiles busy.

Of course, a simple architecture also results in faster time to market, and eliminates the verification nightmare that threatens modern day superscalars. The passage of time has not changed the benefits of simple architectures superbly articulated in the paper by Patterson and Ditzel [8] making a case for RISCs.

Programmable, integrated interconnect The switched interconnect between the tiles provides high bandwidth, low latency, communication between neighboring tiles. Integrated interconnect allows channels with hundreds of wires instead of the traditional tens. VLSI switches are pad limited and rarely dominated by internal switch or channel area. The switches support dynamic routing, but are also programmable and synchronized so that a static schedule can be maintained across the entire machine should the compiler choose to do so. Static routing permits higher utilization of available bandwidth of chip-level channel resources and eliminates dynamic synchronization costs.

Multi-sequential statically-scheduled instruction streams Each tile runs a single thread of computation with its own program counter. Switches that can be programmed with static communication schedules enable the compiler to statically schedule the computations in the threads in each tile. *Static scheduling eliminates synchronization*, and thus provides the cheapest form of synchronization – one that is free.

Section 5 discusses how Raw software can handle cases such as dynamic routing and dependence checking, where traditional systems devote specialized hardware resources for them. Of course, Raw also provides basic flow control support on each channel so that a tile can stall if a neighboring tile violates the static schedule due to an unforeseen dynamic event. As discussed in Section 3, Raw also allows apportioning channel bandwidth between static routing and dynamic routing.

Multigranular operations The Raw architecture implements mechanisms such as wide-word ALUs and is thus coarser than traditional FPGA based computers. In order to achieve the levels of parallelism an FPGA computer can attain, the Raw architecture is multigranular. Multigranular support includes the ability of a single tile to exploit bit or byte-level parallelism when it exists in an application. Logic simulation applications, for example, require bit level operations. FPGA based emulation engines are simply special purpose computers that can exploit bit-level parallelism.

The Intel MMX design and several signal processing architectures have similar mechanisms to allow their ALUs to be used for a few wide word operations or many narrow word computations.

Multigranular support in Raw is achieved through the use of a small amount of configurable logic in each tile which can be customized to the right datapath width.

Configurability Configurability allows the customization of the hardware to support special operations to support specific application demands. Each Raw tile includes a small amount of FPGA-like configurable logic that can be customized at the bit-level granularity to perform special operations. The key with configurability is that both bit or byte-level parallelism can be combined with special communication between each of the individual bits or bytes, and is thus significantly more powerful than MMX-like multigranularity. One way to view configurable logic is as a mechanism to provide the compiler a means to create customized single or multicycle instructions without resorting to longer software sequences.

1.1 Comparison with other Architectures

Raw builds on several previous architectures. IWarp [3] and NuMesh [4] both share with Raw the philosophy of building point-to-point networks that support static scheduling. However, the cost of initiating a message in both systems was so high that compilers could exploit only coarse grain parallelism, or had to focus on very regular signal processing applications so the startup cost could be amortized over long messages. Note that even though iWarp supported very low-latency processor access to the message queues, it required an initial set up of a pathway before word-level data could be sent. Due to the difficulty of predicting statically events over long periods of time, static scheduling in a coarse grain environment is very difficult. The Raw architecture improves on NuMesh and iWarp in several ways. The register like latencies of communication in Raw allows it to exploit the same kinds of instruction-level parallelism that superscalars can exploit. Predicting latencies over small instruction sequences, followed by statically scheduling of multiple tightly coupled instruction streams is much simpler. Just like the processor, the switches in Raw are programmable via an instruction stream produced by the compiler, and are thus significantly more flexible than either iWarp or the multiple finite-state-machine architecture of the NuMesh switches.

Like an FPGA-based computer, the Raw architecture achieves the speeds of special purpose machines by exploiting fine-grain parallelism and fast static communication, and exposing the low-level hardware details to facilitate compiler orchestration. Compilation in Raw is fast, unlike FPGA-computers, because it binds into hardware commonly used compute mechanisms such as ALUs, registers, word-wide data channels and memory paths, and communication and switching channels, thereby eliminating repeated low-level compilations of these macro units. The onerous compile times of our multi-FPGA based Raw prototype computer (see Section 6) amply demonstrates this compile speed problem. Binding of common mechanisms into hardware also yields significantly better execution speed, lower area, and better power efficiency than FPGA systems.

Many features in Raw are inspired by the VLIW work. Like the VLIW Multiflow machine [5], Raw has a large register name space and a distributed register file. Like VLIWs, Raw also provides multiple memory ports, and relies heavily on compiler technology to discover parallelism and statically schedule computations. Unlike traditional VLIWs, however, Raw uses multiple instruction streams. Individual instruction streams give Raw significantly more flexibility to perform independent but statically scheduled computations in different tiles, such as loops with different bounds.

It is natural to compare this work with the natural evolution of multiprocessors, *viz*, integrate on a chip a multiprocessor that uses a simple RISC chip [7]. Like Raw, such a multiprocessor would use a simple replicated tile and provide distributed memory. However, the cost of message startup and synchronization would hamper its ability to exploit fine-grain instruction-level parallelism. A compiler can exploit Raw's static interconnect to schedule ILP across multiple tiles without the need for expensive dynamic synchronization.

Raw architectures also promise better balance than the processor in memory or IRAM architectures. Current proposals for the latter architectures combine within a DRAM chip a single processor, and as such suffer from the same problems as extant superscalar processors, although to a lesser extent. As Dick Sites notes [12], extant superscalars spend three out of four CPU cycles waiting for memory for many applications. Long memory latencies are one of the causes for this and are not significantly reduced in the IRAM approach. The Raw approach directly addresses the memory latency problem by implementing many smaller memories, so there are multiple independent ports to memory, and each memory has lower latency. The parallelism allowed by multiple independent memory ports allows the compiler to issue several memory operations simultaneously and helps mask the latency. Furthermore, if IRAM architectures hold the physical area of memory constant and increase the amount of memory, the latency problem will only be exacerbated. Processing and switching speeds will increase significantly in future generation chips, making delays due to the long memory bitlines and wordlines relatively much worse. The Raw approach can be viewed as an IRAM approach in which *multiple* processing units are sprinkled with distributed memory, where each independent memory module is sized so its access speed scales as the processor clock.

1.2 Roadmap

The rest of the paper is arranged as follows. The next section discusses a simple example to demonstrate the benefits that derive from the Raw architecture. The following section describes the Raw architecture in more detail. Section 3 describes a hardware implementation of the Raw architecture. Sections 4 discusses the Raw compiler, and Section 5 discusses how data-dependent events such as dynamic routing and dependence checking can be supported within a statically scheduled framework. In Section 6 we give results from an early Raw prototype which show that Raw can achieve massive instruction level parallelism with extremely low latency. Section 7 concludes the paper.

2 An Example

This section uses a small Jacobi program fragment to discuss how the features of Raw yield performance improvements over a traditional multiprocessor system. Our discussion focuses on the fine-grain parallelism exploited by a single-chip RawμP. Of course, as in extant multiprocessors, a Raw multiprocessor can exploit coarse grain parallelism using several RawμP chips, a dynamic interconnect and external DRAM, by partitioning the program into multiple coarse-grain threads. The goal of the Raw microprocessor is to extract a further degree of fine-grain parallelism out of each of these threads.

Let us start with a traditional fine-grain implementation of a Jacobi relaxation step in which each processor "owns" a single value, receives a single value from each of four neighbor processors, computes an average value and updates its own value with this average, and transmits its new value to its neighbors. Figure 4 shows successive versions of this Jacobi code, each one yielding progressive improvements in runtime as we integrate various features of Raw. In the first version, most of the time for such a fine-grain step in a traditional multiprocessor implementation would go into communication latency and synchronization delays between the nodes.

The Raw architecture significantly reduces communication and memory access delays through its tightly integrated register-level inter-tile communication. Explicit synchronization is also elim-

Original Code

```
;; MPP VERSION
loop:
send(msg, NDIR) ; send value to the north
receive(s, SDIR) ; wait for value from the south
.
. ; COMMUNICATION LATENCY
.
send(msg, SDIR) ; send value to the south
receive(n, NDIR) ; wait for value from the north
.
. ; COMMUNICATION LATENCY
.
send(msg, EDIR) ; send value to the east
receive(w, WDIR) ; wait for value from the west
.
. ; COMMUNICATION LATENCY
.
send(msg, WDIR) ; send value to the west
receive(e, EDIR) ; wait for value from the east
.
. ; COMMUNICATION LATENCY
.
;; Move data into registers
nreg <- [n] ; n = &msg[x][y-1]
sreg <- [s] ; s = &msg[x][y+1];
ereg <- [e] ; e = &msg[x+1][y];
wreg <- [w] ; w = &msg[x-1][y];

;; Do computation
t0 <- nreg + sreg
t1 <- ereg + wreg
t2 <- t0 + t1
r  <- t2 >> 2 ; r = (((n+s)+(e+w))/4);

;; Move result back to memory
[msg] <- r ; *to = r;

barrier()
.
. ; SYNCHRONIZATION LATENCY
.
iter <- iter + 1 ; while (++iter < max_iter)
t4 <- (iter < max_iter)
if t4 goto loop
```

Second Version

```
;; TIGHTLY COUPLED COMMUNICATION
;; and MULTI-SEQUENTIAL STATIC
loop:
send(r, NDIR)
receive(sreg, SDIR), send(r, SDIR)
receive(nreg, NDIR), send(r, EDIR)
receive(wreg, WDIR), send(r, WDIR)
receive(ereg, EDIR)

;; Do computation
t0 <- nreg + sreg
t1 <- ereg + wreg
t2 <- t0 + t1
r  <- t2 >> 2 ; r = (((n+s)+(e+w))/4);

;; NO BARRIER NEEDED, ALL NODES ARE IN LOCKSTEP

iter <- iter + 1 ; while (++iter < max_iter)
t4   <- (iter < max_iter)
if t4 goto loop
```

Third Version

```
;; CONFIGURABLE
;; (APPLICATION SPECIFIC OPERATION ((a+b+c+d)/4))
loop:
send(r, NDIR) ; send value to the north
receive(sreg, SDIR), send(r, SDIR)
; wait for value from the south
receive(nreg, NDIR), send(r, EDIR)
receive(wreg, WDIR), send(r, WDIR)
; send value to the west
receive(ereg, EDIR) ; wait for value from the east

;; Do computation
r  <-  (((n+s)+(e+w))>>2); APPLICATION SPECIFIC OP

iter <- iter + 1 ; while (++iter < max_iter)
t4   <- (iter < max_iter)
if t4 goto loop
```

Figure 4: Several variations on the Jacobi kernel

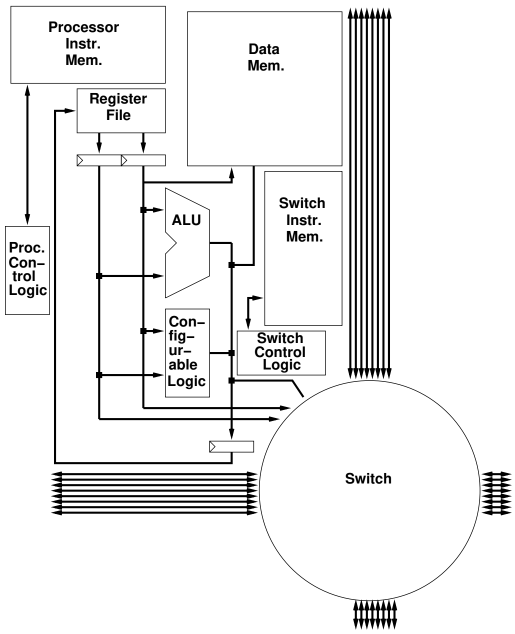

Figure 5: Major datapaths of each individual tile.

inated because the multiple instruction streams are *statically scheduled*. These two benefits are reflected in the second Jacobi code sequence.

The third benefit Raw conveys is support for single-cycle multigranular operations. If we assumed that our Jacobi algorithm demanded 16-bit integer precision, we could possibly pack each register with two of the Jacobi values. We could then perform the 16-bit multigranular operation (addition, in this case) which, in one cycle, would perform two separate additions, one on the top halves of the registers, and the other on the bottom halves.

The fourth benefit arises from Raw's configurable logic. A custom instruction can be instantiated for the application as in the final Jacobi code sequence.

3 Raw Hardware

As discussed in Section 1 the approach used by the Raw architecture is to expose all underlying hardware mechanisms and move scheduling decisions into the compiler. This section describes a Raw hardware architecture which directly exposes a set of mechanisms, including ALUs, instruction sequencing hardware, communication interconnect switches, physical pipeline registers, and network flow control bits, to allow high-quality static scheduling. Additional hardware support is provided for handling dynamic events.

Physical Organization Globally the Raw chip is organized as a mesh of processing tiles. The major datapaths of each tile are shown in Figure 5. The tile contains all the mechanisms found in a typical pipelined RISC processor. These include a register file, ALU, SRAM for software-managed instruction and data caches and pipeline registers. In addition each tile also includes some configurable logic to support bit and byte level parallel operations, and a programmable switch to communicate with neighboring tiles. The switch supports both dynamic routing and compiler orchestrated static routing. The switch is integrated directly into the processor pipeline to support

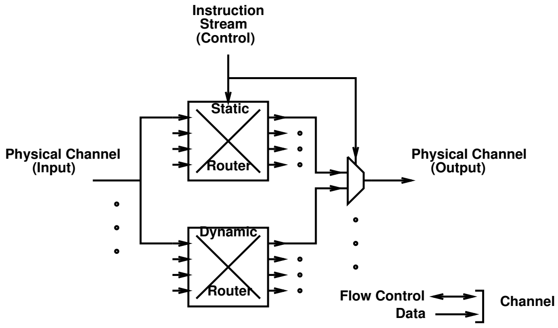

Figure 6: Structure of the Raw switch. The switch multiplexes two different networks, one static and one dynamic, across the same physical wires. The physical channels also provide wires for flow control.

unassigned timeslot available to dynamic router

processor	north	south	east	west
north	d	d	d	d
d	west	proc	west	d
d	d	d	d	north
east	d	d	south	d
d	east	d	d	d
d	d	d	d	d
d	d	north	d	d

Statically assigned route from north to west

time

output channel assignment

Figure 7: Logical information in the communication timetable implemented by the switch program. Each output channel is statically assigned an input. When no static assignment is pre-scheduled, time slots are available to the dynamic router.

single cycle message injection and receive operations.

Sequencing Each tile includes two sets of control logic and instruction memories. The first of these controls the operation of the processing elements. The second is dedicated to sequencing routing instructions for the switch. Separating the control of the processor and the switch allows the processor to take arbitrary data dependent branches without disturbing the routing of independent messages that are passing through the switch. The switch schedule can be changed by loading a different program into the switch instruction memory.

Communication The switch structure is shown in more detail in Figure 6. Physical wires are a critical resource, so the switch multiplexes two logically distinct networks, one static, and one dynamic, over the same set of physical wires. The static network is controlled by the sequence of switch instructions (Figure 7) which describe, for each cycle, which switch inputs are connected to each switch output. Not shown are the additional control bits to support static branches in the switch instruction stream. Because the communication patterns are orchestrated at compile time, messages do not require header interpretation. Thus there is no routing overhead, and efficient single word data transfers are possible. Single word messages allow tiles to take advantage of registers and function units in neighboring tiles to support distributed register files and very fine-grain ILP parallelism.

Any slot in the routing timetable not allocated to static scheduling is available to the dynamic router. The dynamic router is a traditional wormhole router, making routing decisions based on the head of each message. Each word following the header simply follows the same path as the header. The dynamic router includes the required additional lines for flow control in the case that messages become temporarily blocked due to contention or the need to wait for empty slots in the static schedule. Thus in Figure 6, when a physical channel is reserved for the static router, the dynamic router will have a stalling flow control signal.

The processor/network interface is shared between the two networks. The processor uses distinct opcodes to distinguish between accesses to the two networks. The interface performs different actions depending on which network the message is injected into. For the static network the message is injected directly into, or copied directly out of, the static switch. The internals of the dynamic network include send and receive FIFOs. The flow control bits for the dynamic network are visible to the processor so the software can determine whether the send queue is full or the receive queue is empty. This allows the processor to utilize the dynamic network without hardware stalls and permits flexibility in the rate at which individual nodes poll for dynamic events.

Although compiler orchestration ensures that the static switch stalls infrequently, the switches admit the possibility of failing to meet a static schedule because they allow interaction between statically scheduled and dynamic events. Notice that while the static and dynamic network are logically disjoint, data dependencies between statically scheduled code and dynamic messages may exist in the processor code. Such a system must support some form of *flow control* even in the static network for correctness. Our design provides minimal hardware-supported flow control in the static network. This flow-control information is also exported to the software. If the processor or switch attempts to use a busy resource, it is stalled until the resource is freed. A stalled processor or switch, in turn, may cause others to stall as well. While the static router is stalled, the dynamic router is allowed to use all timeslots until the static router is able to make forward progress. We are also exploring higher level end-to-end software flow control strategies to avoid the requirement of flow control on the static network altogether.

Memory Each Raw tile contains a small amount of data SRAM that can be locally addressed. The amount of data memory is chosen to roughly balance the areas devoted to processing and memory, and to match within a factor of two or three the memory access time and the processor clock. Memory accesses will be pipelined and scheduled by the compiler. As with the communication network, the demands of static scheduling require that dynamic events, like cache misses in a conventional processor, do not stall the processor pipeline. The data memory on each tile, therefore, is directly physically addressed. A hardware cache tag file will export a hit/miss status to the software rather than stalling the processor pipeline, so that the software can check the status if it is uncertain a datum is in the cache. Higher level abstractions like caching, global shared memory and memory protection are implemented by the compiler [10], and dynamic events, like cache misses, are handled entirely in software.

Discussion The Raw hardware architecture provides a number of benefits over conventional superscalar designs. Because of its tiled distributed structure Raw can achieve large I-fetch, data memory and register file bandwidth. Since each tile contains its own set of resources, multiple memory units and registers can be accessed in parallel without resorting to building expensive and slow multi-ported register files or memories.

Because individual hardware mechanisms are exposed directly to the compiler, applications with static structures can take advantage of the raw speed available from the static network. Messages that can be statically structured can be injected and removed from the network with no packaging or routing overhead and messages move through the network at the lowest possible latency. Because the Raw design exposes all possible dynamic events to the software, dynamic and static communication

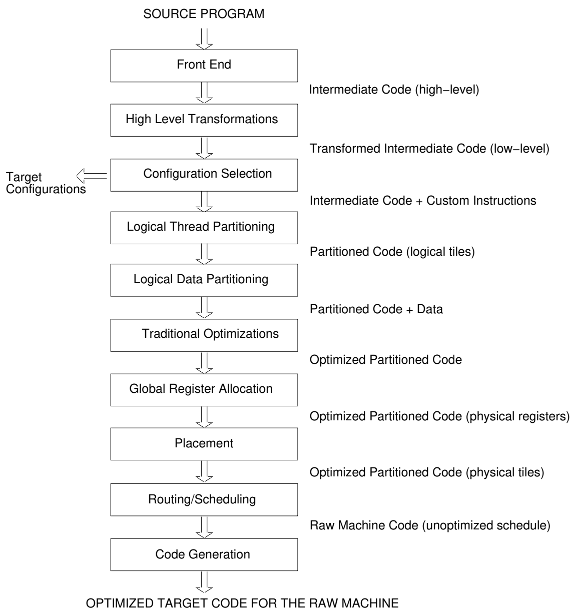

Figure 8: Compilation phases in Raw compiler

patterns can coexist, and unexpected stalls are rare.

Finally, the tiled organization on a single chip permits extremely wide and short message paths. Unlike traditional routers that face pin limits, the data paths in an on-chip router can be several words wide, permitting extremely high-bandwidth communication. The tiled point-to-point communication organization further permits the avoidance of long chip-crossing data-paths. Each signal need travel no more than the width of a single tile each cycle, reducing the effort required to create very high clock-rate designs.

4 Raw Compilation System

The goal of the Raw compiler is to take a single-threaded (sequential) or multi-threaded (parallel) program written in a high-level programming language and map it onto Raw hardware. Raw's explicitly parallel model makes it possible for the compiler to find and express program parallelism directly to the hardware. This overcomes a fundamental limitation of superscalar processors, that the hardware must extract instructions for concurrent execution from a sequential instruction stream. The issue logic that dynamically resolves instruction dependencies in superscalars is a system bottleneck, limiting available concurrency [13].

The Raw machine avoids the bottleneck resulting from a single sequential instruction stream by supporting a *multi-sequential* architecture in which multiple sequential instruction streams execute on multiple tiles (one stream per tile). Unlike the medium-grain or coarse-grain parallelism exploited on multiprocessor systems, the compiler views the collection of N tiles in the Raw machine as a collection of functional units for exploiting instruction-level parallelism.

In this section, we outline a sequence of compilation phases for the Raw machine. These phases are shown in Figure 8. While we may consider merging some of the phases in the future, this is the phase ordering that we will use in building the first version of the Raw compiler. The compilation phases are discussed in the following paragraphs, along with the rationale for this phase ordering.

Front-end The job of the front-end is to generate *intermediate code* for an abstract Raw machine with unbounded resources *i.e.*, unbounded number of *virtual tiles* and *virtual registers* per tile. The generated code reflects the structure of the source program, and may be sequential with a single address space, or multithreaded with a shared address space or a distributed address space, depending on whether the source program is sequential or parallel. Regardless of the form of the source program, the problem of co-locating instructions and data within a tile (Partitioning), and mapping instructions and data to specific tiles (Placement) are performed by later phases in the compiler and not by the front-end.

Data references are kept at a high level and are classified as private or shared. Private data is only accessible by the virtual tile that it is allocated in. Shared data is accessible by all virtual tiles. Data references are also classified by the compiler as being *predictable* ("static") or *unpredictable* ("dynamic") using type information and program analysis information. A predictable data reference is one for which the compiler will be able to predict the destination tile, even though the base address may be unknown. This issue is discussed in detail in the Logical Data Partitioning phase. The conservative assumption for an unanalyzable data reference is to classify it as unpredictable.

For a traditional sequential program with no explicit parallelism or data distribution, the intermediate code generated by the front-end for the abstract Raw machine will look very similar to intermediate code generated in compilers for traditional uniprocessors. The restructuring into multi-sequential code for a physical Raw machine is performed by later phases of the Raw compiler.

High-Level Transformations This phase performs high-level program transformations to expose greater levels of instruction-level parallelism and register locality. The transformations include loop unrolling, procedure inlining, software pipelining, code duplication, scalar replacement, speculative execution, and iteration-reordering loop transformations such as loop interchange, loop tiling, and unimodular loop transformations [14]. As in high-level optimizers in today's compilers for high-performance processors, this phase includes a deep intraprocedural and interprocedural analysis of the program's control flow and data flow. Alias analysis is especially important for reducing the number of unpredictable (dynamic) memory references on the Raw machine.

Branch instructions and array data references in the intermediate code are lowered to machine level instructions after this phase, but they still use symbolic labels as address arguments. High-level transformations is the first optimization phase performed after the front-end, so that the remaining compiler phases can operate on the lower level intermediate code.

Configuration Selection This phase selects a configuration to be loaded into the configurable logic in the Raw machine, for the given input program. For the sake of simplicity the first version of the Raw compiler assumes that each tile is loaded with the same configuration and only considers implementing combinational logic as custom instructions. Future versions of the compiler may lift these restrictions. The compiler will be capable of performing this phase automatically but will also give the programmer the option of specifying custom instructions.

As shown in Figure 9, for each custom operation the configuration phase must both output a specification for the configurable logic and rewrite the intermediate code so that each subtree (compound operation) is replaced by a call to the appropriate custom instruction. The compiler will invoke a logic synthesis tool to translate a custom operation specification into the appropriate bit sequence for the configurable logic in the Raw machine. Our current Raw prototype system described in Section 6 requires that these special operators be identified by the programmer and specified in behavioral Verilog.

The compiler will extend the dynamic programming algorithms used in tree pattern-matching

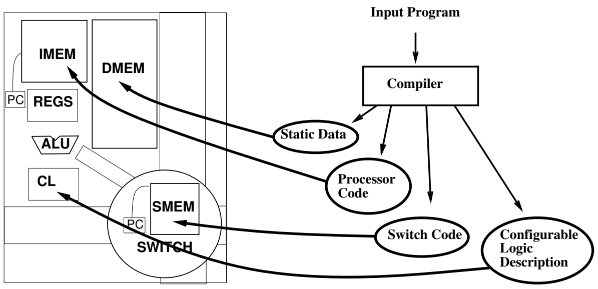

Figure 9: Compiler interaction with the architecture

systems [1] to identify commonly occurring subtree patterns that might be good candidates for implementation as custom instructions. To support multigranular operation the compiler will search for important special cases, such as bitwise and multiple shortword (*e.g.*, 8-bit) arithmetic operations that can be packed together into a single custom instruction.

Partitioning into Logical Threads The Partitioning phase generates parallel code for an idealized Raw machine with the same number of tiles as the physical architecture, but assuming an idealized fully-connected switch so that tile placement is not an issue. This idealized Raw architecture also has an unbounded number of virtual registers per tile and and supports symbolic data references like the abstract Raw machine. The structure of the code after partitioning is similar to that of an single-program-multiple-data (SPMD) program. A single program is assumed for all threads, along with a *my_id* primitive that is used to guide control flow on different threads.

Along with traditional multiprocessor and vector loop parallelization techniques [9] this phase identifies and partitions for fine grain instruction level parallelism. The partitioner must balance the benefits of parallelism versus the overheads of communication and synchronization. Because these overheads are much lower in the Raw microprocessor than in multiprocessors the partitioning can occur at a much finer grain than in conventional parallelizing compilers.

Note that the Raw compiler performs logical thread partitioning before logical data partitioning, which is counter to the approach used by compilers for data-parallel languages such as High Performance Fortran. This phase ordering is motivated by a desire to target a much wider class of applications with instruction level parallelism versus static data-parallelism. In the future, we will explore the possibility of combining thread partitioning and data partitioning into a single phase.

Logical Data Partitioning This phase determines the co-location of data and computation by binding symbolic variables to logical threads. In addition, symbolic names are translated to lower level offsets and locations in the local address spaces of the logical threads. The storage allocation is selected to enhance data locality as much as possible, and may include replication of frequently-used variables/registers. For array variables that have no storage aliasing, this phase can also include selection of data layouts (*e.g.*, row-major or column-major) and of data distributions (*e.g.*, block, cyclic, etc. as in HPF).

This phase assumes that each tile's address space is divided into a *private* section and a *shared* section. An access to the private section is always local. The compiler implements a shared memory abstraction on top of the distributed physical memory using techniques similar to those described in [10].

As far as possible, we would like the Raw compiler to statically predict the destination tile for a memory access to the shared section, so that even an access to a remote tile can be statically routed.

We observe that the *offset* for many memory operations is often known at compile-time (*e.g.*, the stack frame offset for a local variable or the field/element offset in a heap-allocated object) even though the base address may be unknown. Therefore, by using *low-order interleaving* and $N \times B$ *alignment* we can calculate the tile on destination tile for each memory request.

1. *Low-order interleaving* — we assume that the address space is divided into blocks of $B = 2^b$ bytes (typical values of B are 8, 16, 32, 64), and that the addressing of blocks is low-order-interleaved across tiles. This means that the bit layout of a data address in Raw is as follows, where $N = 2^n$ is the number of tiles:

```
                                <---- n -----> <---- b ---->
+----------------------------+--------------+--------------+
| Block address in node/tile | Node/tile id | Block offset |
+----------------------------+--------------+--------------+
```

   This address format makes it easy for the compiler to extract the tile id from a low order address offset.

2. $N \times B$ *alignment* — We assume the base address of commonly used program entities (stack frames, static arrays, heap-allocated structures, etc.) is aligned with a $N \times B$ boundary (*i.e.*, has zeroes in the lowest $n + b$ bits). This enables the compiler to statically predict the destination tile for a known offset, even though the actual base address is unknown.

For base pointers such as function parameters that may not be known at compile-time to be $N \times B$ aligned, we suggest generating dual-path (multi-version) code with a runtime test that checks if all unknown base addresses are $N \times B$ aligned. The code for the case when the test returns true will contain only predictable references (this should be the common case). The code for the false case will contain unpredictable references and thus be less efficient but will serve as a correct fallback code for executing with unaligned arguments.

Traditional Optimizations This phase performs traditional back-end optimizations such as constant propagation, common-subexpression elimination, loop-invariant code motion, strength reduction, and elimination of dead code and unreachable code. The output of this phase is optimized intermediate code with operations at the level of the target instruction set.

Traditional optimizations are performed after Logical Data Distribution so that all address computations in the program are exposed for optimization, but before Global Register Allocation so that register allocation can be performed on optimized code.

Global Register Allocation This phase assigns the unbounded number of virtual/symbolic registers assumed in previous phases to the fixed $N \times R$ registers in the target machine, where R is the number of registers in a single tile. Our motivation in performing Global Register Allocation after Partitioning is to give priority to instruction-level parallelism over register locality. Had we performed Register Allocation earlier, we would be in danger of reducing instruction-level parallelism by adding "false" dependences on physical registers before the Partitioning phase.

The register allocation phase is guided by hardware costs for accessing local/nonlocal registers and local/nonlocal memory locations. Our initial assumption is that the costs will be in increasing order for the following:

1. Access to a local register

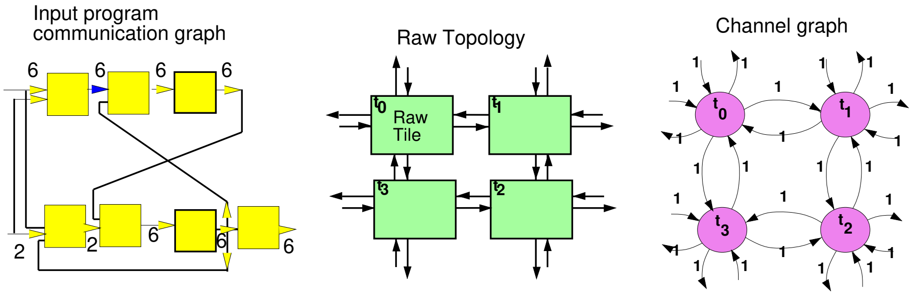

Figure 10: Input program communication graph, the Raw microprocessor topology, and the Raw communication channel graph showing the initial values of the available channel capacities.

2. Access to a nonlocal register (in a neighboring tile)
3. Statically predictable access to a local memory location
4. Statically predictable access to a nonlocal memory location
5. Unpredictable (dynamic) access to a memory location

Placement The only remaining idealization in the machine model at this stage of compilation is the hypothetical crossbar interconnect among threads. The Placement phase removes this idealization by selecting a 1-1 mapping from threads to physical tiles based on the inter-thread communication patterns and the physical structure of the static interconnect among tiles in the Raw machine. Because the earlier data partitioning phase co-located data variables and arrays with the threads, the thread placement phase results in physical data placement as well.

The placement algorithm attempts to minimize a latency and bandwidth cost measure and is a variant of a VLSI cell placement algorithm. This algorithm is used in our current multi-FPGA compiler system [2]. The placement algorithm attempts to lay close together threads that communicate intensely or threads that are in timing critical paths.

As shown in Figure 10, the Placement algorithm uses a dependence graph for the program to determine critical paths and the volume of communication between threads. Each edge is annotated both with the length of the longest critical path on which it is (shown in the figure), and the volume of communication in words it represents (not shown). The placement algorithm also uses a physical channel graph whose edges are annotated with the physical bandwidth they represent. The channel graph captures the topology and the available bandwidth of the RawμP.

The goal of placement is to determine an optimal mapping of the input program graph to the channel graph. Our prototype system uses simulated annealing to optimize a placement cost function. A suitable cost function sums over all pairs of threads the product of the communication volume between a pair of threads and the number of hops between the pair of tiles on which the individual threads are placed, divided by the available bandwidth between the tiles. To account for latency, the cost function also folds in a latency related weight.

Routing and Global Scheduling This phase inserts routing instructions for nonlocal data accesses, and selects the final ordering of computation and routing instructions on individual tiles with the goal of minimizing the overall estimated completion time. This phase will automatically give priority to routing/compute instructions on a critical path, without requiring special-case treatment for either class of instructions.

Routing uses the same input program graph and the channel graph used by placement and assumes that a data value can be communicated between a neighboring pair of tiles in a single cycle. For each value communicated between threads, the scheduler prescribes both a sequence of inter-tile channels and a corresponding sequence of time slots (or cycles) in which each channel will be used. The following is a brief summary of the TIERS (Topology Independent Pipelined Routing and Scheduling) routing and scheduling algorithm that we will use [11]. This algorithm is implemented in our emulation system, and we propose to adapt this for use in Raw.

Given the channel graph and the dependence-analyzed input program communication graph, the TIERS routing approach uses a greedy algorithm to determine a time-space path for each nonlocal communication event and a schedule for the instructions in each thread. The scheduler relies on available channel capacities for each cycle maintained in each edge of the channel graph. The greedy algorithm performs a topological sort of the communication edges in the input program graph so that edges appear in partial dependence order.

The routing algorithm picks edges in order from the sorted list and routes each edge using the sequence of channels in the shortest path between the source and destination tiles. The scheduler determines the cycle on which a value can be scheduled for communication based on three quantities: the arrival times of the edges on which it depends, the amount of time required for any computation or memory accesses that produce the value, the availability of channel capacity along the shortest part during a subsequent sequence of cycles. The scheduler decrements available channel capacities for each channel along the route taken by the edge, and marks a channel unavailable during any cycle on which its capacity is exhausted. Routing of each edge is accompanied by the production of a precise schedule for computations and memory accesses within each thread.

Final Code Generation This phase emits the final code in the appropriate object code format for the Raw Machine. It also emits the appropriate bit sequence for implementing the selected configuration within each tile.

5 Software Support for Dynamic Events

As in any approach advocating exposure of hardware details, one raison de'etre of the Raw Machine is that compilation techniques exist which can manage and indeed take advantage of the resulting explicit control. For static program regions, the compilation process described in Section 4 provides plausible grounds for leveraging this control to achieving attractive performance gains when compared to more traditional architectures. For efficient execution of the full spectrum of general-purpose programs, however, we must complement this compilation paradigm with techniques that support dynamic and unpredictable events.

Dynamic events typically occur when data dependencies and/or latencies cannot be resolved at compile time. Examples of unresolvable data dependence include undecidable program input and non-linear (or indirect) array indexing. Latencies unknown at compile time may be a function of the memory system (cache misses, page faults), data-dependent routing destination, or simply a region of code which executes for a data-dependent number of cycles. A final class of dynamic events includes unpredictable situtations that arise as a function of the external environment of the processor, such as a bus interrupt.

Raw's philosphy for handling dynamic situtations that occur while running a program is to:

- ensure correct execution of all dynamic events,
- handle infrequent events in software (i.e. via polling),
- use dynamic hardware as a last resort[1].

We have discovered that under the right circumstances certain static techniques can actually outperform dynamic protocols at their own game. When static techniques fail, we have discovered software solutions for the Raw architecture which approach the performance of additional hardware without the entailing overhead and complexity. And finally, when we do resort to hardware, such as in the dynamic routing of non-uniform message streams, we must be careful to insure that the unpredictable dynamic situations are handled without interfering with any carefully predicted static events taking place at the same time. This section continues by describing some of these techniques.

5.1 Software Dynamic Routing

While some applications can be compiled in a completely static fashion, many applications have at least some amount of dynamic, data-dependent communication. We use the term *dynamic message* to denote any act of communication that cannot be routed and scheduled by the compiler. Raw uses one of two approaches to handle dynamic messages.

The first is a software approach that preserves the predictability of the program in the face of dynamic messages by using a static, data-independent routing protocol. This approach statically reserves channel bandwidth between nodes that may potentially communicate. In the worst case, this conservative approach will involve an all-to-all personal communication schedule [6] between the processing elements. All-to-all schedules effectively simulate global communication architectures, such as a cross-bar or bus, on our static mesh. This approach is effective for a small number of tiles. It is also effective when messages are short, so that the amount of communication bandwidth wasted is small.

The second approach uses the dynamically routed network. Raw supports a dynamically routed network in which communication is grouped into multi-word packets with a header that includes the destination address. Hardware interprets the message header in a packet at each switching point and dynamically routes the packet in an appropriate direction. By conservatively estimating the delivery time of dynamic messages and allowing enough slack, the compiler can still hope to meet its static schedule. Correctness is preserved in this approach through software checks of the flow control bits. This approach is more effective when the dynamic messages are long, so that the overhead of the message headers and the routing decision making can be amortized over a larger volume of communication.

5.2 Software Memory Dependency Checking

Dynamically checking for dependencies between every pair of memory operations is expensive. Most superscalar processors can sustain only one or at most two (*e.g.*, Alpha 21264) memory instruction issues per cycle. Since, in most programs, memory operations account for at least 20% of the total dynamic instructions, this limit on memory issue sharply limits the total parallelism that a superscalar processor can achieve.

[1]While our pedagogical "vanilla" Raw Machine will include the minimal hardware mechanisms required, our research directions include evaluation of various levels of hardware support for the kinds of dynamic events found in real applications.

```
loop:   st w, [a+v]          ; a[b[i-3]] = w
        ld [a+u], w          ; w = a[c[i-2]]
        shl t, 2, v          ; v = t * 4 (from iteration i-2)
        shl s, 2, u          ; u = s * 4 (from iteration i-1)
        ld [b+i], t          ; t = b[i-i]
        add i, 4, i          ; i++
        ld [c+i], s          ; s = c[i]
        sub n, i, x          ; loop test
        bnz x, loop
```

Figure 11: Software pipelined code for the loop `for (i = 0; i < n; i++) a[b[i]] = a[c[i]]`. With register renaming a superscalar could achieve 4 cycles per iteration on this loop and 2 cycles per iteration only if it can permit 2 memory instructions to issue every cycle. The memory instruction issue constraint limits the processor from making any further improvement.

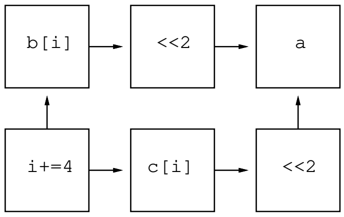

Figure 12: Because the compiler knows that accesses to the `a`, `b` and `c` arrays are all independent, it can create a 6 tile systolic structure on the Raw hardware that achieves 2 cycles per iteration. A value is read out of each of the `b` and `c` arrays every cycle, while the tile handling the `a` array alternates each cycle between handling a load or a store. The limit is a true value dependence between each load from the `a` array and the store in the following cycle.

Figure 11 shows a software pipelined instruction schedule that might be generated by a compiler for the loop

`for (i = 0; i < n; i++) a[b[i]] = a[c[i]]`

Because the loop requires 4 memory operations per iteration the limit on performance is the number of load/store functional units.

A compiler, however, can easily determine that the references to each of the arrays `a`, `b` and `c` are independent from each other. A typical superscalar RISC instruction set provides no method by which the compiler can transmit this information to the hardware. All memory references must be checked for dependencies, even if they can be statically determined to be independent.

In a Raw architecture, on the other hand, the compiler can allocate a seperate tile to act as the memory system for each independent data structure. Figure 12 demonstrates a systolic structure that maintains all the true value dependencies of the original loop, but allows independent memory references to proceed in parallel. In the figure each box represents a Raw computation node. The tile labeled `i+=4` increments the index variable every other cycle and sends the new value to its two neighbors. The tiles labeled `b[i]` and `c[i]` take in a value for i on their input and do a load every other cycle. The two nodes labeled << 2 just take their input and shift it every other cycle. Finally, the node labeled `a` alternates between reading the location given by its lower input and storing the value back at the location given by its left hand input.

The Raw implementation achieves 2 cycles per iteration, without resorting to the "heroic efforts" that are required to provide multiple load/store functional units and dual ported caches in superscalar processors. Providing hardware for dynamic dependency checking is less important in the Raw architecture because the architecture provides a method by which the compiler can provide memory dependence information to the hardware.

For those applications that require high bandwidth runtime dependency checking, software dependence checking structures can be constructed by the compiler from multiple Raw tiles. As with VLSI implementations of dependency checking, these software structures scale poorly in terms of both size and latency. On a superscalar every application pays the cost of dynamic dependency checking, even for those memory references which are known, statically, to be independent. In the Raw architecture it is completely plausible to consider implementing multiway memory issue in software, because only those applications that require the feature need dedicate resources for it.

6 Results

To assess the difficulty of writing compilers that can statically orchestrate all the communication and computation in multiple threads, and to determine the level of performance achievable, we have implemented a prototype Raw processor (called RawLogic) and an associated compilation system by leveraging commercial FPGA based logic emulation technology. RawLogic supports some (but not all) of the features of the Raw architecture described in earlier sections. Specifically, it supports tightly integrated communication and computation, multigranularity and configurability, and comprises simple replicated tiles. However, it supports only a very limited form of instruction sequencing. Multiple static instruction instruction sequences are converted into individual hardware state machines and *hardwired* in RawLogic. Because it uses FPGAs, RawLogic compilation is time consuming.

This section describes the RawLogic system and the results of a benchmark suite consisting of a set of twelve general purpose computing programs, including sorting, searching, matrix and graph computations. To compile these programs to RawLogic, we have developed a common framework based on *raw computation structures*, RawCS. Each application is implemented with multiple copies of operator primitives selected from a computation library primitive written in behavioral verilog. These computation elements are then connected based on the dependence structure of the unrolled loops, to generate a behavioral Verilog netlist. While in our current implementation the generate procedures are hand-coded by application of known compilation techniques such as partial evaluation, loop unrolling, and speculative parallelization, our goal for a more faithful implementation of Raw is to develop a high-level compiler based on SUIF to automate this task. Executing these benchmarks programs on our prototype RawLogic machine, running at 25MHz and built with FPGAs, achieves 10x-100x speedup over a commercial Sparc20/71.

Raw Computation Structures Programs compiled in the RawCS framework have two parts: a set of operator primitives, and an application-specific computation structure generator written in C. RawCS operator primitives such as multi-input adds are provided in a library. The primitive operators are written in behavioral verilog. Programmers can add to this library components specialized to their own application. A component specialized to the Jacobi relaxation example discussed earlier produces an output which is the average of its five input values.

The RawCS computation structure generator is written in C by the applications programmer. The generator reflects a graph whose edges represent the data dependences in an unrolled version of the program. The input to this generate program includes parameters specifying problem size,

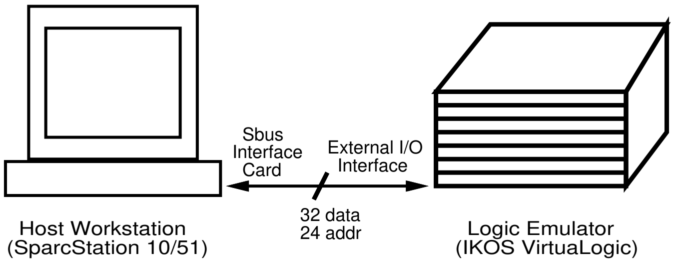

Figure 13: RawLogic: Prototype Reconfigurable Computing System

Tool Function	Software Used	Rough running times
RawCS Generator	New C Programs	seconds
Behavioral Compiler	Synopsys	minutes
RTL Compiler	Synopsys	ten minutes
Virtual Wires Compiler	IKOS VirtuaLogic	two hours per board
FPGA Compiler	Xilinx	two hours per board (ten workstations)

Figure 14: Experimental Software System

datapath width, and degree of parallelism. Given these inputs, the generate program makes calls to the RawCS operator library to instantiate computation structure components and data channels. The output is a single design in synthesizable behavioral verilog. At this point the verilog code is application-specific but architecture-independent – It can be targeted to any multi-FPGA system that accepts a single design netlist.

In our system, the output behavioral Verilog netlist is then automatically synthesized to a gate-level verilog netlist. This gate level netlist is then processed by a VirtualWires compiler, and includes the steps of partitioning, placement and scheduling [11, 2]. These steps are akin to the corresponding phases in our compiler flow described in Section 4 process the resulting verilog netlist and produce a binary that maps into the multi-FPGA hardware substrate. This binary comprises individual state machines representing programs both the computation threads and for statically scheduled communication switching.

Benchmark Results We have targeted a Benchmark Suite to RawLogic. As depicted in Figure 13, RawLogic is a prototype hardware system consisting of a VirtuaLogic Emulator (VLE) from IKOS Systems coupled with a Sun SparcStation 10/51 via a SLIC S-bus interface card from DAWN VME Products. Not shown is a SCSI interface to the emulator for downloading configurations and controlling clock speed. We are currently using a production VLE system consisting of five arrays of 64 Xilinx 4013 FPGAs each. The FPGAs in the system are connected in a direct, point-to-point interconnect. The system has over a thousand I/Os.

Figure 14 lists the software used for each tool step and rough running times on SparcStation 20 class machines. By far the most computationally expensive step is the last FPGA compile. We have successfully compiled most of the smaller designs, all the way down to configuration binaries, and run the designs on our prototype emulation system. The execution of these designs has been validated with I/O across our memory-mapped Sbus interface. The emulation software provides accurate estimates of execution speed, so we report those numbers for larger programs due to the onerous FPGA compile times.

Benchmark Category	Benchmark Name	element count	gate count	FPGA count	speedup (vs software)
Binary Heap	bheap15	15	10K	19	0.62
	bheap127	127	89K	64	0.80
Bubble Sort	bubble64	64	59K	64	6
	bubble108	108	100K	64	10
	bubble256	256	240K	319	12
	bubble512	512	484K	x	22
Jacobi Relaxation	jacobi8x8	64	22K	36	29
	jacobi16x16	256	98K	64	127
	jacobi32x32	1024	426K	320	306
Conway's Game of Life	life32x6	192	45K	34	127
	life64x16	1024	239K	129	289
	life96x16	1536	358K	216	371
Integer Matrix Multiply	matmult4x4	16	8K	19	65
	matmult8x8	64	33K	43	41
Merge Sort	merge8	14	9K	19	6
	merge64	126	79K	62	2.96
	merge256	510	317K	287	2.11
Single Source Shortest Path	ssp16	16	44K	14	10
	ssp64	64	181K	56	27
	ssp64-mesh	64	159K	46	32
	ssp128	128	366K	118	41
	ssp256	256	814K	261	52
Multiplicative Shortest Path	spm16	16	156K	36	14
	spm32	32	310K	90	25
Transitive Closure	tc512	512	187K	48	398

Figure 15: Benchmark Results

By using different problem sizes we generated cases for each benchmark ranging in size from a few FPGAs to hundreds of FPGAs. Figure 15 compares execution results for each of these benchmark cases to a software only version executing on a 125 SpecInt SparcStation 10 processor. We also list gate count and the total number of Xilinx 4013 FPGAs required.

7 Conclusion and Current Research

Raw is a software-exposed VLSI architecture comprising a replicated set of processing tiles, and a compiler system that discovers and statically schedules instruction-level parallelism across the tiles. A prototype multi-FPGA system supporting some of the features of Raw has been implemented. The prototype demonstrates that it is feasible for compilers to statically orchestrate all the communication and computation in many parallel applications, and that performance levels 10x-100x over workstations is achievable. Our applications work also indicates that the multi-sequential static model permits the expression of massive parallelism on a number of simple integer benchmarks. The implementation of a more faithful simulator for Raw and a compiler for high-level languages based on SUIF are in progress. A faithful FPGA-based emulation of Raw and a VLSI implementation are also planned.

8 Acknowledgements

The research reported in this paper was funded by DARPA contract #DABT63-96-C-0036 and by an NSF Presidential Young Investigator Award. The Virtualogic emulation system was donated by Ikos Systems.

References

[1] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *ACM Transactions on Programming Languages and Systems*, 11(5):491–516, October 1989.

[2] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal. Logic emulation with virtual wires. In *To appear in IEEE Transactions on CAD*, 1997.

[3] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting systolic and memory communication in iWarp. In *Proc. 17th Annual Symposium on Computer Architecture*, Seattle, May 1990.

[4] David Shoemaker, Frank Honore, Chris Metcalf, and Steve Ward. NuMesh: An Architecture Optimized for Scheduled Communication. *The Journal of Supercomputing*, 10:285–302, 1996.

[5] Joseph A. Fisher. Very Long Instruction Word Architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 140–150, Stockholm, Sweden, June 1983.

[6] Susan Hinrichs, Corey Kosak, David R. O'Hallaron, Thomas M. Stricker, and Riichiro Take. An architecture for optimal all-to-all personalized communication. Computing Science Tech Report CMU-CS-94-140, CMU, Pittsburgh, PA, September 1994.

[7] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, Cambridge, Massachusetts, October 1–5, 1996.

[8] D. A. Patterson and D. R. Ditzel. The Case for the Reduced Instruction Set Computer. *Computer Architecture News*, 8(6):25–33, October 1980.

[9] Vivek Sarkar and John Hennessy. Compile-time Partitioning and Scheduling of Parallel Programs. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, 21(7):17–26, June 1986.

[10] 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.

[11] Charley Selvidge, Anant Agarwal, Matt Dahl, and Jonathan Babb. TIERS: Topology independent pipelined routing and scheduling for VirtualWire compilation. In *1995 ACM International Workshop on Field-Programmable Gate Arrays*, pages 12–14, Berkeley, CA, February 1995. ACM.

[12] Dick Sites. Architects look to processors of future, applications, instruction sets, memory bandwidth are key issues. *Microprocessor Report*, pages 18–24, August 1996.

[13] Michael D. Smith, Mike Johnson, and Mark A. Horowitz. Limits on multiple instruction issue. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 290–302, Boston, Massachusetts, April 3–6, 1989.

[14] Michael E. Wolf and Monica S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

A Matrix Multiply Example

In this section, we use a matrix multiply program as an example to illustrate the compilation phases of the Raw compiler outlined in section 4. A simple and effective approach to executing the matrix multiply program on the Raw machine is to use well-known loop parallelization techniques for exploiting loop-level parallelism across multiple tiles. However, to better illustrate issues related to instruction-level parallelism and register locality, the compilation example shown below is based on fine-grain parallelism rather than loop-level parallelism. This example does not use the configurable logic in the Raw machine, but it should give the reader a good idea of how the Raw compiler can perform static allocation and scheduling for all kinds of programs including programs that are not amenable to loop-level parallelization.

Input Program: Recall that the input program to the Raw compiler may be sequential or explicitly parallel. In this example, we assume that the input is a sequential program:

```
int i, j, k;
float a[M,M], b[M,M], c[M,M];

for ( i = 0 ; i < M ; i++ )
    for ( j = 0 ; j < M ; j++ )
        a[i,j] = 0 ;

for ( i = 0 ; i < M ; i++ )
    for ( j = 0 ; j < M ; j++ )
        for ( k = 0 ; k < M ; k++ )
            a[i,j] += b[i,k] * c[k,j]
```

Front-end: Since the input program is sequential, the intermediate code generated by the front-end for the abstract Raw machine looks very similar to intermediate code generated in compilers for traditional uniprocessors. For simplicity, we just show the intermediate code generated for the inner `k` loop. It uses an unbounded number of symbolic/virtual registers (e.g., k.val, b.val, b.addr, b.stride, _temp1) as follows:

```
                    ...
          loadi   k.val,      0
label L1:
          loadf   b.val,      [b.addr]
          add     b.addr,     b.addr,    b.stride
          loadf   c.val,      [c.addr]
          add     c.addr,     c.addr,    c.stride
          mulf    _temp1,     b.val,     c.val
          loadf   a.val,      [a.addr]
          addf    _temp2,     a.val,     _temp1
          storef  [a.addr],   _temp2
          add     a.addr,     a.addr,    a.stride
          add     k.val,      k.val,     1
          blt     k.val,      M.val,     L1
                    ...
```

High Level Transformations: After translating the input program to abstract intermediate code, the Raw compiler performs high-level program transformations to expose greater levels of instruction-level parallelism and register locality. For the matrix multiply example, the important transformations are unrolling of the outer parallel `i` and `j` loops ("unroll-and-jam") and loop-invariant scalar replacement of the `a[i,j]` array references. For convenience, we show a "source-level" representation of the transformed matrix multiply loop nest below, even though the high-level transformations are performed by the compiler on the intermediate code.

In this example, an unroll factor of four was used for loops `i` and `j`, resulting in an innermost unrolled loop body containing $4 \times 4 = 16$ copies of the original loop body. Since the loops being unrolled are parallel, the 16 copies are all independent and provide a copious amount of instruction-level parallelism. Larger unroll factors would further reduce the execution time of this example, but would lead to more cumbersome code for the human eye. Note that all the `a[i,j]` array references are *loop-invariant* and have been replaced by scalar temporaries `_sum0`, . . ., `_sum15` in the inner `k` loop. In addition, outer loop unrolling improved register locality by reusing `b[i,k]` and `c[k,j]` values within an unrolled iteration. The transformed code shown below includes remainder loops which are created by the loop unrolling transformation to correctly handle the case when M is not a multiple of 4:

```
for ( i = 0 ; i < M ; i += 4 )
    for ( j = 0 ; j < M ; j += 4 ) {
        _sum0 = a[i,j] ;
        _sum1 = a[i+1,j] ;
        _sum2 = a[i+2,j] ;
        _sum3 = a[i+3,j] ;
        _sum4 = a[i,j+1] ;
        _sum5 = a[i+1,j+1] ;
        _sum6 = a[i+2,j+1] ;
        _sum7 = a[i+3,j+1] ;
        _sum8 = a[i,j+2] ;
        _sum9 = a[i+1,j+2] ;
        _sum10 = a[i+2,j+2] ;
        _sum11 = a[i+3,j+2] ;
        _sum12 = a[i,j+3] ;
        _sum13 = a[i+1,j+3] ;
        _sum14 = a[i+2,j+3] ;
        _sum15 = a[i+3,j+3] ;
```

```
        for ( k = 0 ; k < M ; k++ ) {
           _sum0  +=  b[i,k]   * c[k,j] ;
           _sum1  +=  b[i+1,k] * c[k,j] ;
           _sum2  +=  b[i+2,k] * c[k,j] ;
           _sum3  +=  b[i+3,k] * c[k,j] ;
           _sum4  +=  b[i,k]   * c[k,j+1] ;
           _sum5  +=  b[i+1,k] * c[k,j+1] ;
           _sum6  +=  b[i+2,k] * c[k,j+1] ;
           _sum7  +=  b[i+3,k] * c[k,j+1] ;
           _sum8  +=  b[i,k]   * c[k,j+2] ;
           _sum9  +=  b[i+1,k] * c[k,j+2] ;
           _sum10 +=  b[i+2,k] * c[k,j+2] ;
           _sum11 +=  b[i+3,k] * c[k,j+2] ;
           _sum12 +=  b[i,k]   * c[k,j+3] ;
           _sum13 +=  b[i+1,k] * c[k,j+3] ;
           _sum14 +=  b[i+2,k] * c[k,j+3] ;
           _sum15 +=  b[i+3,k] * c[k,j+3] ;
        }
        a[i,j]     = _sum0 ;
        a[i+1,j]   = _sum1 ;
        a[i+2,j]   = _sum2 ;
        a[i+3,j]   = _sum3 ;
        a[i,j+1]   = _sum4 ;
        a[i+1,j+1] = _sum5 ;
        a[i+2,j+1] = _sum6 ;
        a[i+3,j+1] = _sum7 ;
        a[i,j+2]   = _sum8 ;
        a[i+1,j+2] = _sum9 ;
        a[i+2,j+2] = _sum10 ;
        a[i+3,j+2] = _sum11 ;
        a[i,j+3]   = _sum12 ;
        a[i+1,j+3] = _sum13 ;
        a[i+2,j+3] = _sum14 ;
        a[i+3,j+3] = _sum15 ;
    }
    for ( ; j < M ; j++ )
        for ( k = 0 ; k < M ; k++ )
            a[i,j] += b[i,k] * c[k,j] ;

for ( ; i < M ; i++ )
    for ( j = 0 ; j < M ; j++ )
        for ( k = 0 ; k < M ; k++ )
            a[i,j] += b[i,k] * c[k,j] ;
```

Partitioning into Logical Threads: We concentrate on the body of the innermost loop of the transformed program and show how it can be mapped onto a Raw machine. For this example, we assume a small target Raw configuration consisting of a 2×2 mesh of Raw tiles. The partitioning phase will partition the 16 copies of the original loop body onto 4 threads. Let's say that the compiler uses locality of reference to array b to guide its partitioning decisions, and places all uses of b[i,k] on to the same thread. That leads to the following partitioning (again shown at the source-level, for convenience):

Thread 0	Thread 1
_sum0 += b[i,k] * c[k,j] ;	_sum1 += b[i+1,k] * c[k,j] ;
_sum4 += b[i,k] * c[k,j+1] ;	_sum5 += b[i+1,k] * c[k,j+1] ;
_sum8 += b[i,k] * c[k,j+2] ;	_sum9 += b[i+1,k] * c[k,j+2] ;
_sum12 += b[i,k] * c[k,j+3] ;	_sum13 += b[i+1,k] * c[k,j+3] ;
Thread 2	**Thread 3**
_sum2 += b[i+2,k] * c[k,j] ;	_sum3 += b[i+3,k] * c[k,j] ;
_sum6 += b[i+2,k] * c[k,j+1] ;	_sum7 += b[i+3,k] * c[k,j+1] ;
_sum10 += b[i+2,k] * c[k,j+2] ;	_sum11 += b[i+3,k] * c[k,j+2] ;
_sum14 += b[i+2,k] * c[k,j+3] ;	_sum15 += b[i+3,k] * c[k,j+3] ;

Logical Data Partitioning: The compiler can now choose a storage layout and data distribution to reduce non-local memory references for the thread partitioning. In fact, in some cases it would be better for the compiler to combine the thread partitioning and data distribution phases into a single phase.

For this example, we can get the best locality by storing array `b` in a column-major layout and array `c` in a row-major layout. The layout of array `a` does not have a big performance impact because there are no loads and stores to array `a` in the innermost loop. We can just choose a row-major layout for array `a` as a default layout.

Assuming that the row/column size of each matrix (4×M bytes) is a multiple of $N \times B$, and that the start of each array is aligned with an $N \times B$ boundary, the storage layout for arrays `a`, `b`, `c` across threads will look as follows for $N = 4$ and B = 4 bytes (in a real Raw machine, we expect B to be $\geq$ 8 bytes):

Thread 0	Thread 1	Thread 2	Thread 3
...	...	...	...
a[0,0]	a[0,1]	a[0,2]	a[0,3]
a[0,4]	a[0,5]	a[0,6]	a[0,7]
...	...	...	...
a[0,M−4]	a[0,M−3]	a[0,M−2]	a[0,M−1]
...	...	...	...
a[M−1,M−4]	a[M−1,M−3]	a[M−1,M−2]	a[M−1,M−1]
b[0,0]	b[1,0]	b[2,0]	b[3,0]
b[4,0]	b[5,0]	b[6,0]	b[7,0]
...	...	...	...
b[M−4,0]	b[M−3,0]	b[M−2,0]	b[M−1,0]
...	...	...	...
b[M−4,M−1]	b[M−3,M−1]	b[M−2,M−1]	b[M−1,M−1]
c[0,0]	c[0,1]	c[0,2]	c[0,3]
c[0,4]	c[0,5]	c[0,6]	c[0,7]
...	...	...	...
c[0,M−4]	c[0,M−3]	c[0,M−2]	c[0,M−1]
...	...	...	...
c[M−1,M−4]	c[M−1,M−3]	c[M−1,M−2]	c[M−1,M−1]
...	...	...	...

Traditional Optimizations: After performing traditional optimizations, the optimized code in the inner `k` loop looks as follows for the four logical threads:

Thread 0				Thread 1	Thread 2	Thread 3			
	...			...	...		...		
loada	b.addr0,	address of b[i,k]		...	...	**loada**	b.addr3,	address of b[i+3,k]	
loada	c.addr0,	address of c[k,j]		...	...	**loada**	c.addr3,	address of c[k,j+3]	
loada	c.limit0,	address of c[M-1,j]		...	...	**loada**	c.limit3,	address of c[M-1,j+3]	
	...			...	...		...		
label L0:				...	...	**label L3:**			
loadf	b.val0,	[b.addr0]		...	...	**loadf**	b.val3,	[b.addr3]	
add	b.addr0,	b.addr0,	b.stride	...	...	**add**	b.addr3,	b.addr3,	b.stride
loadf	c.val0,	[c.addr0]		...	...	**loadf**	c.val3,	[c.addr3]	
add	c.addr0,	c.addr0,	c.stride	...	...	**add**	c.addr3,	c.addr3,	c.stride
				...	...	**waitfor**	Thread 0 to compute c.val0		
mulf	_temp0,	b.val0,	c.val0	...	...	**mulf**	_temp3,	b.val3,	c.val0
addf	_sum0,	_sum0,	_temp0	...	...	**addf**	_sum3,	_sum3,	_temp3
waitfor	Thread 1 to compute c.val1			...	...	**waitfor**	Thread 1 to compute c.val1		
mulf	_temp4,	b.val0,	c.val1	...	...	**mulf**	_temp7,	b.val3,	c.val1
addf	_sum4,	_sum4,	_temp4	...	...	**addf**	_sum7,	_sum7,	_temp7
waitfor	Thread 2 to compute c.val2			...	...	**waitfor**	Thread 2 to compute c.val2		
mulf	_temp8,	b.val0,	c.val2	...	...	**mulf**	_temp11,	b.val3,	c.val2
addf	_sum8,	_sum8,	_temp8	...	...	**addf**	_sum11,	_sum11,	_temp11
waitfor	Thread 3 to compute c.val3			...	...				
mulf	_temp12,	b.val0,	c.val3	...	...	**mulf**	_temp15,	b.val3,	c.val3
addf	_sum12,	_sum12,	_temp12	...	...	**addf**	_sum15,	_sum15,	_temp15
ble	c.addr0,	c.limit0,	L0	...	...	**ble**	c.addr3,	c.limit3,	L3
	...			...	...		...		

The optimized code reflects the high-level transformations (loop unrolling and scalar replacement) that had been performed earlier. This optimized code is still for an idealized Raw machine with a shared register set of unbounded size. The **waitfor** pseudo-instructions reflect the inter-thread dependences dictated by the *program dependence graph*. For example, the data dependence constratint requires that the use of register C.val0 in Thread 3 wait for the load into register C.val0 in Thread 0 to complete. For the matrix multiply example, all **waitfor** synchronization constraints will be satisfied by static scheduling of computation and routing instructions. In other programs, we may have to use software flow control techniques to satisfy some of the synchronization constraints.

Global Register Allocation: The compiler can now assign physical registers to the local variables and temporaries for each thread as follows, based on the thread partitioning. One possible register allocation is as follows:

Register	Tile 0	Tile 1	Tile 2	Tile 3
r1	b.addr0	b.addr1	b.addr2	b.addr3
r2	c.addr0	c.addr1	c.addr2	c.addr3
r3	b.stride	b.stride	b.stride	b.stride
r4	c.stride	c.stride	c.stride	c.stride
r5	c.limit0	c.limit1	c.limit2	c.limit3
f1	b.val0	b.val1	b.val2	b.val3
f2	c.val0	c.val1	c.val2	c.val3
f3	_temp0	_temp1	_temp2	_temp3
	_temp4	_temp5	_temp6	_temp7
	_temp8	_temp9	_temp10	_temp11
	_temp12	_temp13	_temp14	_temp15
f4	_sum0	_sum1	_sum2	_sum3
f5	_sum4	_sum5	_sum6	_sum7
f6	_sum8	_sum9	_sum10	_sum11
f7	_sum12	_sum13	_sum14	_sum15

Placement and Global Scheduling: For this example, we will assume a simple one-to-one placement of threads onto tiles in which thread i is mapped onto tile i. The routing of data among tiles is accomplished by the code for the node program and the switch program shown below. For convenience, we assume a virtual ring topology in describing the routing of data to the left and from the right. The "left" and "right" virtual directions can be replaced by the appropriate hardware physical directions in the mesh topology. For this example, all tiles have identical code in the inner loop thus lending itself to an SMPD-style code generation in which all tiles execute the same code on different data. However, in general, the output program will contain branches to different instruction sequences for different tiles, implemented as a switch-style multi-way branch on the tile id.

This code executes the 16-way unrolled loop body in 12 cycles on a 4-processor Raw configuration, which amounts to an average of one matrix multiply iteration every 0,75 cycles. A larger Raw configuration would deliver a proportionately larger throughput.

Node Program

label	L0		
loadf	f1,	[r1]	
add	r1,	r1,	r3
loadf	f2,	[r2]	
add	r2,	r2,	r4
mulf	f3,	f1,	f2
addf	f4,	f4,	f3
mulf	f3,	f1,	in
addf	f5,	f5,	f3
mulf	f3,	f1,	in
addf	f6,	f6,	f3
mulf	f3,	f1,	in
addf	f7,	f7,	f3
ble	r2,	r5,	L0

Switch Program

push result f2 Right
copy in from Left
route in from Left to Right
copy in from Left
route in from Left to Right
copy in from Left