The Supercomputer Toolkit: A General Framework for Special-Purpose Computing

Harold Abelson[1], Andrew A. Berlin[1,4], Jacob Katzenelson[2],
William H. McAllister, Guillermo J. Rozas[1,5],
Gerald Jay Sussman[1], Jack Wisdom[3]
Analytical Medical Laboratory
HPL-94-30
March, 1994

computer architecture, parallel architectures, high performance system design, compilers, astrophysics, astronomy

The Supercomputer Toolkit is a family of hardware modules (processors, memory, interconnect, and input-output devices) and a collection of software modules (compilers, simulators, scientific libraries, and high-level front ends) from which high-performance special-purpose computers can be easily configured and programmed. Although there are many examples of special-purpose computers (see [4]), the Toolkit approach is different in that our aim is to construct these machines from standard, reusable parts. These are combined by means of a user-configurable, static interconnect technology. The Toolkit's software support, based on novel compilation techniques, produces extremely high-performance numerical code from high-level language input.

We have completed fabrication of the Toolkit processor module, and several critical software modules. An eight-processor configuration is running at MIT. We have used the prototype Toolkit to perform a breakthrough computation of scientific importance—an integration of the motion of the Solar System that extends previous results by nearly two orders of magnitude.

While the Toolkit project is not complete, we believe our results show evidence that generating special-purpose computers from standard modules can be an important method of performing intensive scientific computing. This paper briefly describes the Toolkit's hardware and software modules, the Solar System simulation, and conclusions and future plans.

[1] Department of Electrical Engineering and Computer Science, MIT [2] Department of Electrical Engineering, Technion - Israel Institute of Technology [3] Department of Earth, Atmospheric, and Planetary Sciences, MIT [4] now with Xerox Park, Palo Research Center [5] now with HP Laboratories

Published in the *International Journal of High Speed Electronics*, Vol. 3, Nos. 3 & 4, September & December 1992

The Supercomputer Toolkit: A general framework for special-purpose computing[1]

Special-purpose computational instruments will play an increasing role in the practice of science and engineering. Although general-purpose supercomputers are becoming more available, there are significant applications for which it is appropriate to construct special-purpose dedicated computing engines. The Supercomputer Toolkit is intended to make the construction and programming of such special-purpose computers routine and inexpensive, in some cases even automatic.

The Toolkit is a family of hardware modules (processors, memory, interconnect, and input-output devices) and a collection of software modules (compilers, simulators, scientific libraries, and high-level front ends) from which high-performance special-purpose computers can be easily configured and programmed. The hardware modules are intended to be standard, reusable parts. These are combined by means of a user-reconfigurable, static interconnect technology. The Toolkit's software support, based on novel compilation techniques, produces extremely high-performance numerical code from high-level language input, and will eventually automatically configure hardware modules for particular applications.

Our Supercomputer Toolkit is intended to help bring scientists and engineers back into the design loop for their computing instruments. Traditionally, scientists have been intimately involved in the development of their instruments. Computers, however, have been treated differently—scientists who require the highest performance computation are primarily users of general-purpose computers supplied by a few remote vendors. These computers must, almost necessarily, be expensive shared resources with high administrative overhead, because obtaining generality while maintaining high performance comes at the price of more complex hardware and more sophisticated software than would be required in a machine whose design has been specialized to a particular problem.

In contrast, a less expensive specialized computer can become an ordinary experimental instrument belonging to the group that constructed it. Once the machine has been constructed, the group can dedicate it to computations that run for many hours, days, or weeks without significant cost.

[1] with apologies to Joel Moses [18].

This is a social change which, we believe, has the potential of introducing a qualitative change in the way scientific and engineering computation is accomplished. (See, for example [13].)

Over the past four years, the MIT Project for Mathematics and Computation and Hewlett-Packard's Information Architecture Group have been collaborating on the design and construction of a prototype Supercomputer Toolkit. A system configured with eight processors is has been running at MIT since the spring of 1991.

In a demonstration of the success of the Toolkit approach, we used the prototype Toolkit to compute the long-term motion of the Solar System, improving upon previous integrations by two orders of magnitude. This was a computation of major scientific significance, because it confirmed that the motion of the Solar System is chaotic. A report on this analysis was published in *Science* [23], which devoted an editorial to the significance of this achievement [16].

Our prototype Toolkit is targeted at numerical computations where performance is limited by the need to integrate systems of ordinary differential equations. Such computations are characterized by a bottleneck in scalar floating-point performance rather than in I/O or in memory bandwidth. These computations are typically not easy to vectorize. Highly pipelined vector processors do not do well on them, because the state variables of the system must in general be updated by computing different expressions. In the Solar System computation, our eight-processor Toolkit—assembled from standard TTL parts that were readily available in 1989, and programmed in highly abstract Lisp code—achieves scalar floating-point performance equal to eight times a Cray 1S programmed in Cray Fortran. While this does not match the speed of the fastest available supercomputer (even in 1989), the relative price advantage of the Toolkit allows it to be used for applications that would be otherwise infeasible. The Solar System computation, for example, required running the Toolkit for 1000 hours—an amount of time that would be prohibitively expensive on a commercial supercomputer.

The Toolkit's price/performance advantage derives from two architectural principles that we followed in the hardware design, coupled with the compiler technology that we are developing to support scientific computation. The first architectural principle is the use of *problem-specific communication paths*. Starting with a particular algorithm, one can often partition

the problem among the available processors and find a static arrangement of interprocessor communication paths that admits nearly optimal utilization of processing power for that algorithm. We believe that for many important scientific applications, high-performance configurations can be generated in a straightforward way, perhaps even automatically, and easily constructed and programmed using Toolkit modules.

The second architectural principle is the use of *synchronous ultra-long instruction word* machines. Even in problems that are not vectorizable, scientific applications typically have substantial data-independent structure. One can configure a totally synchronous machine, in which most interprocessor communication is scheduled statically at compile time. In effect, the multiple VLIW execution units of the machine are programmed as a single ultra-long instruction word processor. This organization eliminates the need for (program) synchronization, bus protocols, run-time handshaking, or any operating-system overhead.

The Toolkit's compiler uses a novel strategy based upon partial evaluation [7, 9]. This exploits the data-independence of typical numerical algorithms to generate exceptionally efficient object code from source programs that are expressed in terms of highly abstract components written in the Scheme dialect of Lisp [14]. This has enabled us to develop a library of symbolic manipulation components to support the automatic construction of simulation codes. As a measure of success, our Solar-system simulation code, constructed with this library, issues a floating-point operation on 98% of the instructions.

The Toolkit approach has obvious limitations. Neither our hardware architecture nor our interconnection technology can be expected to scale to systems with many hundreds of processors. On the other hand, the Toolkit does realize a means, practical within the limits of current technology, to provide relatively inexpensive supercomputer performance for an important class of problems. Efforts with similar goals include the iWARP work at CMU [11] and the NuMesh work at MIT [25]. The iWARP cell is a building block for a variety of processor arrays. The NuMesh is a packaging and interconnect technology supporting high-bandwidth systolic communications on a 3D nearest-neighbor lattice.

Our work on the Toolkit is far from complete, and the present paper is a snapshot of an experiment in progress, rather than a report on a finished project. We have reached a significant milestone in that an eight-processor

system is working, significant software support has been implemented, and a major application has run successfully. On the other hand, we have much work yet to do, particularly on automatic compilation of parallel programs and automatic configuration of hardware modules. We have not yet attempted applications that require large memory, or for which an appropriate Toolkit configuration would be larger than a few boards. Nevertheless, we hope others will be interested in the experience we have gained so far and in the prospects for the future.

Section 1 describes the Toolkit hardware. Section 2 presents the low-level programming model. Section 3 describes the high-level programming model and the compilation technology. Section 4 gives some details on our benchmark application to long-term integrations of the Solar System. Section 5 contains conclusions and future plans.

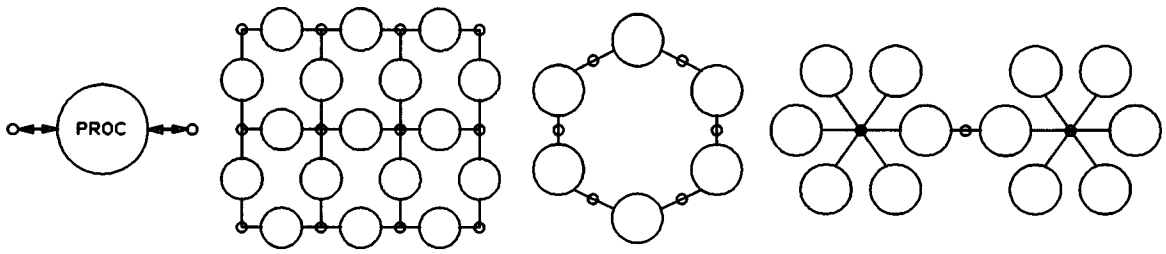


Figure 1: Interconnection Graphs. Each Toolkit processor module has two bidirectional I/O ports. Any interprocessor connection graph can be configured with each node having degree up to about 8. The figure shows how to build various network architectures: a mesh, a ring and communicating clusters.

1 Toolkit Hardware

At the highest level, the Toolkit is a network of processors, memories, and I/O modules with the topology chosen or suggested by the user, possibly with the help of the compiler. Processors have two I/O ports that are connected with fine-pitch ribbon cables to a number of other similar ports. Limiting the number of ports simplifies the hardware design, yet permits a variety of network graphs. Any connection graph may be implemented by placing a processor on each branch of the graph, as shown in figure 1. A node in the graph is actually a manually installed shared bus that we have tested with up to eight ports.

An additional one-wire "global-flag" connects between processors in a similar way. It may be set and sensed by each processor. Its primary use is for software synchronization as shown in section 2.6.

Once the customized Toolkit is assembled, it appears to the programmer that all elements run in lock step. The communication paths are under complete control of the software. There is no hardware for arbitration or bus protocol. The program specifies the source and sinks for data on each bus on each cycle. Software convention determines whether the bus value is data, address, or control information.

All cables for data communication, clocks, and host communication connect to the front edge of each processor board. A user assembles a machine by plugging in the required modules and connecting the cables appropriately. When a particular machine is no longer needed, it can be disassembled, and its modules can be reassembled into other configurations.

Each Toolkit processor is a high performance computer with its own private memory for code and data. The processor design was guided largely by two factors: the class of problems we were targeting; and the use of off-the-shelf technology available in 1989. The class of problems is the solution of systems of ordinary differential equations—computations characterized by a large number of floating-point operations on a relatively small amount of data. Given the very small design team, the technology choices were confined to readily available commercial parts.

The architecture is centered around a high-performance floating-point chip set. The remaining hardware is designed to feed operands to the floating-point chips without interruption. Figure 2 shows a block diagram of the processor.

The prototype Toolkit is housed in a minicomputer chassis borrowed from an existing Hewlett-Packard product line (HP9000/850). The processor contains about 220 integrated circuits laid out on a 14×16 inch printed-circuit board as shown in figure 3. The board is eight layers with 8–10 mil traces. All components use TTL logic levels. Our current collection of eight processor boards consumes about 1200 watts. The cycle time is 80 ns, which results in an instruction rate of 12.5 million instructions per second.

The cost of the hardware was about $3,300 per board. The cost of the hardware for the whole project was less than $40,000, including 9 boards, and chassis.

The following sections contain a brief description of the hardware. For more details the reader is referred to [2].

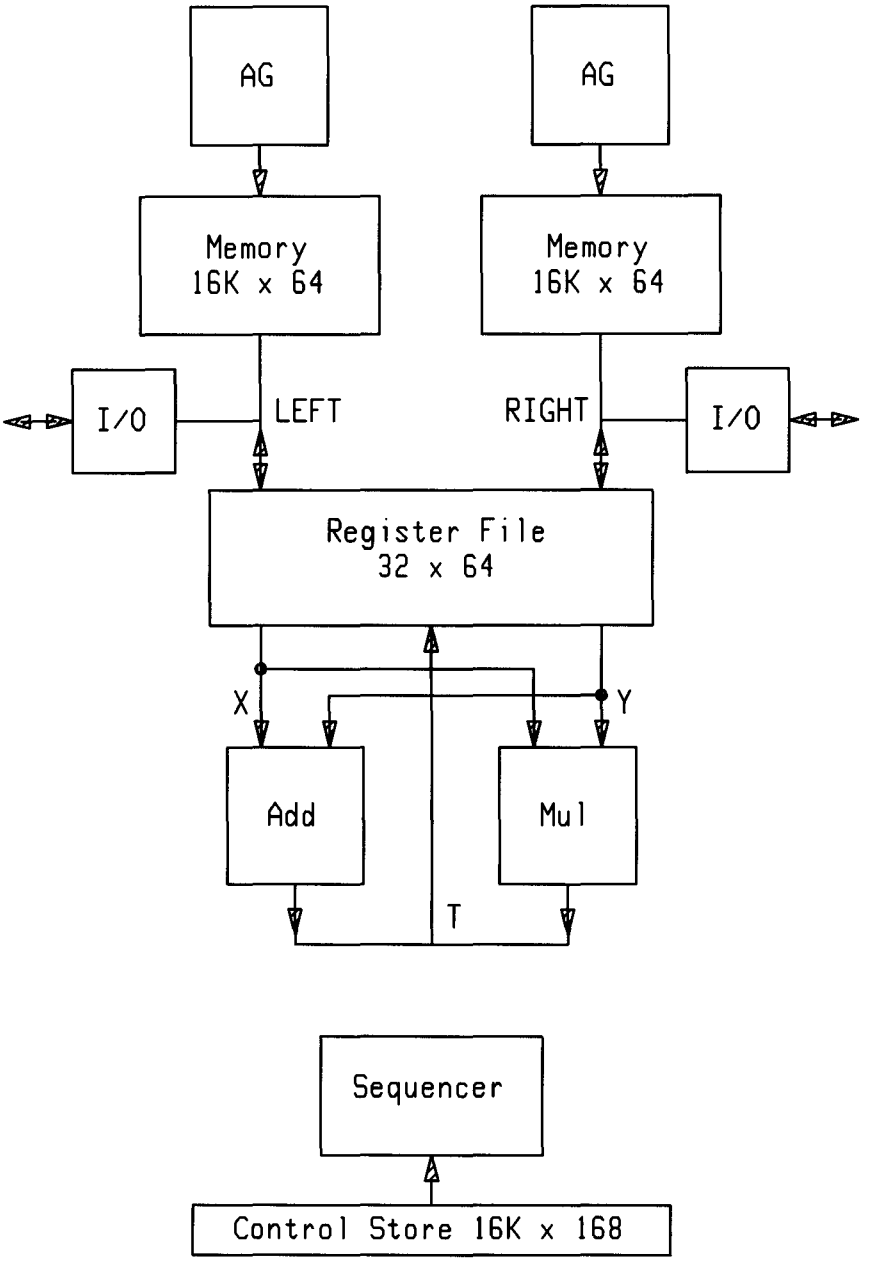
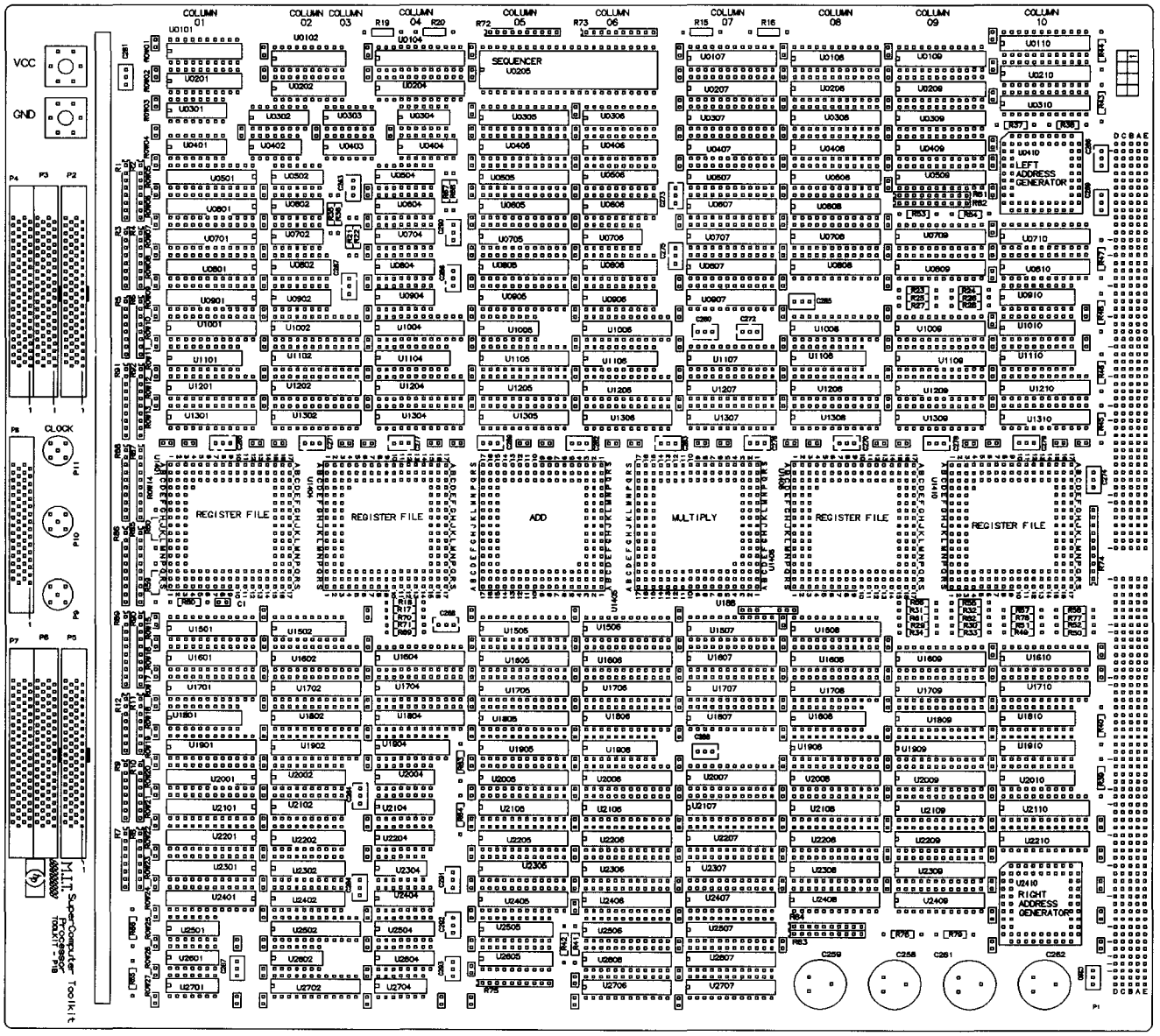


Figure 2: Processor Architecture. The Toolkit processor consists of floating-point unit (ALU and Multiplier), a register file, two data memories, each with its own address generator (AG), two I/O ports, instruction memory, and a program sequencer. The figure shows the major buses: X, Y, T, Left, and Right. Not shown are the side paths, the internal paths in the Math chips, and the host interface.



SCT-80001 5-14-90 COMPONENT SIDE SILKSCREEN

Figure 3: Toolkit processor module. The six large pin grid arrays along the centerline are the four register files, the ALU and the Multiplier. The other specialized chips are the address generators in the 68-pin square ICs, and the the sequencer in the 48-pin dual inline package. Along the left edge of the board are connectors for the clocks, I/O cables, and host cable.

1.1 Floating-point unit

The heart of the processor is a pair of Bipolar Integrated Technology (B.I.T.) floating-point math chips [10][2]. Both chips have 32-bit I/O paths and full 64-bit internal paths. To simplify the design, all operations are done to double precision. Operations complete in one cycle except for the Multiplier's divide and square root, which are multi-cycle operations.

Separate instruction bits to each math chip allow the ALU and Multiplier to be controlled independently. The peak floating-point performance is therefore two operations per cycle, or 25 MFLOPS per processor at the current 80ns clock rate. Section 2.7 below describes an inner-product routine that runs at this peak rate. In practice, we have been able to sustain roughly half this rate—about 12 MFLOPS per processor—on the applications of interest. As a point of comparison, the HP9000/835 workstation, a contemporary machine using similar technology (the same math chips, RAM, and instruction rate), was rated at about 2 MFLOPS on a common floating-point benchmark.

1.2 Register File and Buses

As illustrated in figure 2, the processor contains a central register file that communicates with the data memories, math chips, and I/O ports. The main buses, the data memories, the register file, the I/O ports, and the communication cables are protected by one parity bit for each byte of data. The register file is built with four register-file chips that logically hold thirty-two double-precision entries.

There are other data paths in the architecture that are not shown in figure 2. Recirculation paths inside the math chips allow results to bypass the register file to become operands on either math chip. This feature is illustrated in Section 2.7 with a program that computes inner-product.

There are two "side paths" that permit transfers between the address generators, floating-point unit, and microinstruction register. In our VLIW architecture, these buses are the communciation path between the four distinct functional units. An important design consideration was to insure adequate functionality while avoiding limits on the cycle time. The 16-bit

[2]The floating-point chip set consists of the B2120A-25 ALU, the B2110A-55 Multiplier, and four B2210A Register Files.

buses link the X and Y register file outputs with the immediate input on the left and right address generators. They allow floating-point values to be used in address computations. In the other direction, constants in the instruction can be driven onto the X and Y buses, to be used as operands by the math chips. This was the minimum functionality that we expected to need to support our applications.

1.3 Data Memory

To keep the system balanced with respect to bus bandwidth and to keep the math chips as busy as possible, we chose to have two independent data memories. Each holds 16K double-precision values for a total of 256K bytes on each processor. The data memory is implemented with 20ns 16K $\times$ 4-bit static RAMs. The RAMs are accessed once in each instruction cycle.

The address generators are simple 16-bit single-chip microprocessor ALU slices [3]. The chip contains a general purpose ALU backed by a 64-entry address-register file.

1.4 Sequencer and Instruction Memory

Overall program flow is controlled by the sequencer chip. Conditional jumps are based on a number of floating-point flags, the global flag, and a flag from the host processor. The pipelined sequencer calculates the address of the instruction that will be executed two cycles later. Thus, a branch instruction issued in cycle N takes effect in cycle $N + 2$. The instruction directly following a jump instruction is always executed.

Independent instruction fields control the operation of the four functional units: floating-point, left and right address generators, and sequencer. Instructions are 168 bits (21 bytes) in length, implemented with the same RAMs as the data memory. A processor holds 16K instructions or 336Kbytes.

1.5 Input/Output Ports

Each I/O port is a double precision register and a very high current TTL transceiver. A port transmits a word between processors in two cycles.

[3]The address generator and sequencer chips are 16-bit CMOS versions of the familiar AMD 2901 microprocessor slice and the AMD 2910 microprogram controller.

There is no hardware arbitration on the I/O ports so the programmer must develop a convention for controlling access to each communication channel.

The inter-board connections are designed to achieve a "first wavefront" communication path. The transmission lines flow continuously from point to point with no branches or stubs. On the board, they are routed in a continuous path from an input connector, to the transceiver, and then to an output connector. Cables, connectors, traces, and terminators all have a controlled impedance of 80 ohms. This interconnection scheme requires twice as many connectors as a traditional design but provided a superior electrical environment for signals.

1.6 Clocks

System clocks are generated from a single crystal on a separate clock/host-interface board. Copies of the master clock are carefully buffered and distributed to hold clock skew to ± 1 ns. This margin is quite adequate for our design.

The processors use a tapped delay line to create controlled clock edges at 7ns intervals. Two programmable logic arrays combine various taps to create different clock waveforms. This clocking methodology proved to be very flexible during the design phase. It was easy to achieve good performance with a wide variety of clocking requirements for different off-the-shelf parts. It promoted reliable design of setup and hold times. On the other hand, the large number of clocks were complex to modify, terminate, and control.

1.7 Host Interface

Processors communicate with the host workstation via a 16-bit general purpose parallel interface. This serial protocol runs at about 1 μsec for each transaction. The interface controls a 29-byte-long scan path that threads through the instruction register and address registers. This interface is very slow compared to the rest of the machine, and it restricts the range of applications of our prototype machine. The host interface is one of the first areas of the design that should be improved.

2 Toolkit Low-level Programming Model

The Supercomputer Toolkit processor is programmed as a very-long instruction word (VLIW) computer. The programmer has full control of all of the hardware resources discussed in section 1. Thus, in every cycle, the following operations can be performed in parallel:

- Two floating-point operations, one in the ALU and one in the Multiplier. The ALU and Multiplier share input and output ports so two values can be fetched from the register file and one result can be written back.
- Two memory-I/O bus transactions, one on the Left bus and one on the Right bus. Data is exchanged between memory and the main register file, or between the register file and the I/O port.
- Two address computations, one in the left address generator and one in the right address generator. The addresses will be used to access the data memories in the following cycle. The address generators have internal register files to support these operations.
- One sequencer operation, to generate an instruction address. Typical sequencer instructions include conditional branch, jump, continue, and call.
- Two flags may be set: the global flag goes to neighboring processors; the host flag goes to the host workstation.

To program the Toolkit at this level, we use a primitive symbolic assembly language. An instruction appears as a list of tagged fields, listed any order. The assembler supplies default values for omitted fields.

```
((flop ...)                  ;floating point operations
 (lmio ...) (rmio ...)       ;left and right I/O bus transactions
 (lag ...)  (rag ...)        ;left and right address computations
 (sequencer ...)             ;sequencer operations
 (flags ...))                ;set flags
```

The fields may be listed any order. The assembler supplies default values for omitted fields.

The remainder of this section briefly describes the fields and provide some programming examples. Additional details and more extended examples can be found in [2].

2.1 Floating-point Operations

The ALU and Multiplier operate simultaneously. Each Toolkit instruction has a separate opcode field for the ALU and Multiplier. Any ALU opcode can appear together with any Multiplier opcode. Most floating-point, integer, and logical operations require one cycle. Divide requires 4 cycles and square root requires 7 cycles. The entire list of available operations can be found in the B.I.T. data sheet for this chip set [10].

The six-cycle sequence shown in figure 4 illustrates the pipelining of these operations, as well as the assembler syntax for the floating-point portion of a Toolkit instruction. As illustrated here and in the examples to follow, the processor pipeline is controlled partly in hardware and partly in software. For example, floating-point opcodes are specified in the same instruction as the operand register numbers. Pipeline registers hold and delay the opcodes while the register read takes place. Then, the opcodes and operands enter the math chips together on the following cycle. The command to latch the math chip result register (`&latch`) is specified in another instruction, and the command to write the result back to the register file is specified in a third (`t` ...).

2.2 Bus Operations

The two memory-I/O buses can perform a 64-bit transfer either between registers and memory, or between registers and an I/O port. There are no direct register to register operations or memory to memory operations. Register-memory transfers require an address to have been generated during the previous cycle. For example, the instruction

```
((lmio m->r r23) (rmio r->m r17))
```

loads r_{23} with the contents of the left-memory address and stores r_{17} into the right memory address.

```
Cycle                    Instruction

  1      ((flop (x r10) (y r11) (* dmult x y)))

  2      ((flop (x r12) (y r13) (* dmult x y &latch)
                                (+ dadd x y)))

  3      ((flop (x r14) (y r15) (* dmult x y &latch)
                        (t * r5) (+ dadd t z &latch)))

  4      ((flop (* &latch) (+ &latch)))

  5      ((flop (t + r6)))

  6      ((flop (t * r7)))
```

Figure 4: Pipelining. This six-cycle sequence illustrates the pipelining of floating-point operations. The instruction in cycle 1 places the contents of r_{10} on the X bus, the contents of r_{11} on the Y bus, and passes the double-precision multiply opcode to the Multiplier. In cycle 2, the multiplier computes the product requested in cycle 1, and then latches the result, $r_{10}r_{11}$, into its result register. Also during cycle 2, instructions and data are passed to the Multiplier and ALU requesting both the product $r_{12}r_{13}$ and the sum $r_{12} + r_{13}$. In cycle 3, the product $r_{10}r_{11}$ is driven from the Multiplier onto the T bus and is written into r_5. This value is also specified as an operand for the next ALU operation (the t after *dadd*). Simultaneously, the product $r_{12}r_{13}$ and the sum $r_{12} + r_{13}$ are computed and latched into the result registers of the Multiplier and ALU. Data for the next multiplication is transferred from registers r_{14} and r_{15}. The second ALU operand refers to the ALU's result; The result, $r_{12} + r_{13}$, is transferred over the ALU's internal feedback path from the result register back to an operand register. In cycle 4 the new sum $(r_{10}r_{11} + r_{12} + r_{13})$ is computed and latched by the ALU. The Multiplier computes and latches $r_{14}r_{15}$. In cycle 5, the ALU result is written to r_6 via the T bus. In cycle 6 the Multiplier result computed and latched during cycle 4 is written to r_7.

2.3 Input/Output Operations

Communication between boards is accomplished by transferring 64-bit quantities between registers and the I/O ports. For example, suppose that the left I/O port of processor 1 is connected to the right port of processor 2 and to the left port of processor 3. The following code fragment transmits the contents of r_{10} in processor 1 to r_{20} in processor 2 and to r_{25} in processor 3:

```
Cycle   Processor 1       Processor 2      Processor 3

  1   (lmio r->io r10)         ...              ...
  2         ...                ...              ...
  3         ...          (rmio io->r r20)  (lmio io->r r25)
```

2.4 Address Generation

The two address generators produce a new data memory address every cycle, which is used in the following cycle. Each address generator has its own internal ALU, and 64 16-bit registers to store addresses. Source operands come from the internal registers, from a 16-bit immediate field in the instruction word, or from the main register file via the side paths.

ALU operands are denoted by A and B, the immediate constant field by D, a zero source by Z. Available ALU operations include addition, subtraction, and Boolean operations. The result of an operation can be passed to the address generator output, and may be written back to the internal address-register selected by B.

Here are two examples that illustrate the use of the address-generator field. More details can be found in [2].

`(lag (add dz nop low) (d 1117))` The left address generator performs the addition of D and Zero with carry-in set low. The result, 1117, is passed to the output, without writing it back to the address-register file (`nop`).

`(rag (add zb ramf high) (b ag-r0))` This performs pre-increment indexed addressing using register `ag-r0` as the index register. The right address generator increments the contents of `ag-r0` by adding it to

zero with the carry-in set high. The incremented result becomes the address-generator output and is stored back into `ag-r0` as specified by the destination operation `ramf`.

2.5 Instruction Sequencing

The sequencing operations [15] include conditional branches, subroutine calls, and looping, all maintained through a 33-deep on-chip stack. For example, loops are implemented using the sequencer operations `push` and `rfct`. `Push` pushes the next address onto the internal stack, and sets an internal counter to the number of loop iterations minus one. `Rfct` decrements the counter and keeps branching to the saved address until the count becomes negative, at which point it pops the stack and falls through the loop. Sequencer operations are pipelined and take effect one cycle after they appear in the instruction stream. At this point, when no jump is present, the sequencer location counter (PC) contains the instruction's location plus 2. For example, the first instruction in the body of a `push/rfct` loop will be two cycles after the appearance of the `push` in the instruction stream.[4]

Figure 5 shows a subroutine that uses the loop instruction to upload to the host a known-length vector by repeatedly calling the `upload` subroutine, which uploads the floating-point number in `r1`. The vector is stored in consecutive locations in left memory. The caller initializes the left address-register named `ptr` to point to the first of these locations. The vector length minus one is the constant `n-1`, which must be known at assembly time. The subroutine is called with a `cjs` instruction (which stacks its own location plus 2) followed by a `nop`. The subroutine returns with a `crtn` (which jumps to the address at the top of the stack and pops the stack) followed by a `nop`.

2.6 Multiprocessing and Synchronization

Since all Toolkit processors are driven by a single master clock, the individual processors are electrically synchronized in terms of when cycle boundaries occur. However, each processor has its own sequencer and instruction

[4]The instruction directly following a jump is referred to as the "jump slot" or "delay slot" in modern reduced instruction set computers.

```
Cycle
     (label upload-vector)
1    ((sequencer push true n-1))          ;setup loop index
2    ((lag (add zb nop low) (b ptr)))     ;address for first element

     ;;Body of loop is cycles 3 through 6
3    ((lmio m->r r1)                      ;move vector element to r1
      (sequencer cjs true upload))        ;subroutine call to upload
4    ()                                   ;delay slot for the cjs
5    ((sequencer rfct))                   ;test/decrement loop counter
6    ((lag (add zb ramf high) (b ptr)))   ;increment the vector pointer

7    ((sequencer crtn true))              ;return from upload-vector
8    ()                                   ;delay slot for the crtn
```

Figure 5: A Toolkit subroutine to upload a vector to the host, using a simple loop counter in the sequencer. Note in this example that the sequencer operations **push**, **rfct**, **cjs** (conditional jump to subroutine), and **crtn** (conditional return from subroutine) are here all conditionalized with **true**, so that they always take effect.

memory, so the operations performed by the processors are, in general, independent of one another. When data is transferred among processors the programmer must arrange that the receivers all read the data two cycles after the transmitter drives it out. This can be passively arranged by cycle counting or by an explicit synchronization action.

The one-bit **global** flag can be used to maintain synchronization among processors during the course of a computation. One programming technique is to maintain synchronization between processors at all times by having each processor execute the same instruction sequence. All processors branch simultaneously, based on the condition being asserted on the **global** flag, in effect, behaving as if there was one central controller governing all processors. This approach works well on vectorizable, data-independent problems, but breaks down when local data-dependent decisions must be made.

A more complex alternative is to allow each processor to execute branches independently, based on its own local data, with synchronization occurring only occasionally when communication is required.

For example, a step in a multiprocessor program might require each

```
Cycle
        (label wait-for-all-boards)
1       ((sequencer ldct true wait-for-all-boards-done))
2       ((sequencer cjp true wait-for-all-boards-assert-ready))
        (label wait-for-all-boards-assert-ready)
3       ((sequencer jrp global wait-for-all-boards-assert-ready)
         (flags (global . low)))
        (label wait-for-all-boards-done)
4       ()
5       ((sequencer crtn))
6       ()
```

Figure 6: Processor Synchronization. This routine uses the **global** flag to synchronize processors so that they will all return in the same cycle when the flag is deasserted by all boards. The Toolkit assembler default for the **flags** field asserts the **global** flag, ensuring that it will remain asserted until all processors execute the instruction in cycle 3. As each processor calls the routine, it waits at (3) in a 1-cycle loop and deasserts the flag. The loop is implemented with the sequencer **jrp** conditional jump instruction, which branches either to the address specified in the instruction, or to the address stored in the internal counter, depending on the state of the flag. The counter register is loaded by the **ldct** instruction in cycle 1. Note that a one-cycle loop is attained even though the sequencer is pipelined, by including a jump instruction (3) in the "delay slot" of a previous jump (2) (**cjp** is a conditional jump to its label). Interested readers should trace through the control structure here in detail, noting that when the loop terminates, the no-op at 4 is executed twice.

processor P_i to perform a computation C_i, where different C_i may require different numbers of cycles. In this case, the **global** flag can be used as a "busy" indicator: As long as each processor is busy, it asserts the flag. When all processors are done, they release the flag, informing all processors that they can proceed to the next phase of the computation. The subroutine **wait-for-all-boards**, shown in figure 6, can be used to implement this kind of synchronization. Any processor calling this routine will remain in a wait loop until all processors have called the routine, whereupon all processors return in the same cycle.

2.7 Benchmark Examples

Figure 7 shows a single-processor program that computes the inner product of two vectors at the processor's peak speed of two floating-point operations per cycle (25 double-precision MFLOPS at the current clock rate). The two vectors are stored in memory, one in the left memory and one in the right memory. The routine is pipelined to use a two-cycle loop that does two multiplications and two additions. The code of figure 7 uses both the arithmetic pipeline discussed at section 2.1 and figure 4 and the sequencer pipeline (delay slot) discussed at sections 2.5 and figures 5 and 6.

Another Toolkit demonstration program (code not shown here) solves $n \times n$ systems of linear equations $Ax = b$ by means of Gauss-Jordan elimination with full pivoting, using the algorithm given in [19]. The running time of the algorithm is dominated by the row reduction performed each time a pivot is selected. This involves subtracting a multiple of the pivot row from each other row of the matrix, as expressed in the following Fortran program:

```
DO 21 LL=1,N
  IF (LL.NE.ICOL) THEN
      DUM=A(LL,ICOL)
      A(LL,ICOL)=0.0
      DO 18 L=1,N
        A(LL,L)=A(LL,L)-A(ICOL,L)*DUM
        18 CONTINUE
      B(LL)=B(LL)-B(ICOL)*DUM
  ENDIF
21 CONTINUE
```

The Toolkit implementation runs the inner loop (`DO` loop 18) at a rate of one floating-point operation per cycle (12.5 MFLOPS). This is accomplished by holding the matrix `A` in left memory, but copying the pivot row `A[ICOL,*]` into right memory when the pivot is chosen. Using both memories provides enough bandwidth to schedule a floating-point operation on every cycle of the inner loop.

The bottleneck in this computation is the shared floating-point result bus which is shared by the Multiplier and ALU. Thus, even though subtractions and multiplications could be done simultaneously, only a single result

```
Cycle
   (label inner-product)
1  ;;setup loop count, data address pointers, clear data path:
   ((sequencer push true n-1)              ;L:=n-1, TOS:=PC+2
    (flop (x r10) (y r10) (* dmult x y))   ;[r10]=0
    (lag (add dz ramf high) (d lbase-1) (b a1))  ;A_lag := 1 + lbase-1
    (rag (add dz ramf high) (d rbase-1) (b a1))) ;A_rag := 1 + rbase-1

2  ;;load first data items into registers, increment data pointers:
   ((flop (x r10) (y r10) (* dmult x y &latch) (+ dadd x y))
    (lmio m->r r1)                         ;[r1]:= [1 + lbase-1],
    (rmio m->r r2)                         ;[r2]:= [1 + rbase-1],
    (lag (add zb ramf high) (b a1))        ;A_lag := 1+[a1_left]
    (rag (add zb ramf high) (b a1)))       ;A_rag := 1+[a1_right]

   ;; The next two cycles are the inner loop.
3  ((sequencer rfct)                       ;PC:= TOS if L==0, L:=L-1
    (flop (x r1) (y r2) (t *)              ;T drives mul result to ALU
          (* dmult x y &latch)            ;compute r1 * r2
          (+ dadd  t z &latch))            ;compute mul + accumulator
    (lmio m->r r3)                         ;[r3]:= 1+[a1_left]
    (rmio m->r r4)                         ;[r4]:= 1+[a1_right]
    (lag (add zb ramf high) (b a1))        ;A_lag := 1+[a1_left]
    (rag (add zb ramf high) (b a1)))       ;A_rag := 1+[a1_right]

4  ((flop (x r3) (y r4) (t *)              ;T drives mul result to ALU
          (* dmult x y &latch)            ;compute r3 * r4
          (+ dadd t z &latch))             ;compute mul + accumulator
    (lmio m->r r1)                         ;[r1]:= 1+[a1_left]
    (rmio m->r r2)                         ;[r2]:= 1+[a1_right]
    (lag (add zb ramf high) (b a1))        ;A_lag := 1+[a1_left]
    (rag (add zb ramf high) (b a1)))       ;A_rag := 1+[a1_right]

5  ((flop (t *) (* &latch) (+ dadd t z &latch)))  ;drain pipeline
6  ((flop (t *) (+ dadd t z &latch)))
7  ((flop (+ &latch)) (sequencer cjp true done))
8  ((flop (t + r5)))                       ;store final result in r5
   (label done)
```

Figure 7: Inner Product. This routine computes the inner product of two vectors of length $2n$ at the processor's peak speed of 2 floating-point operations per cycle. The vectors are stored in left and right memory beginning at locations 1+**lbase-1** and 1+**rbase-1**. The inner loop, which does two multiplications and two additions, is executed n times. The ALU result register (z) acts as the accumulator. In the annotations L, TOS, and PC are the loop counter, the top of stack, and the program counter, respectively. A_{lag} is the address from the left address generator, etc. Register contents are denoted as [*name*].

can be written to the registers on each cycle. In contrast, the inner-product program can sustain two floating-point operations per cycle because it uses the ALU result register to hold the partially computed sum, and writing to the registers is not required.

3 High-level Programming Model

The Toolkit software environment includes a compiler that converts numerical routines written in a high-level language (the Scheme dialect of Lisp [14]) into extremely efficient, highly pipelined Toolkit programs. These compiled routines can be combined with hand-written assembly language programs by means of a programmable linker.(See [24] for another system where Lisp source code is translated to VLIW code.)

Our present compiler requires the programmer to divide programs into data-independent segments, that is, segments in which the sequence of operations is not dependent on the particular numerical values of the data being manipulated. For instance, the sequence of multiplications and additions performed in a Fast Fourier Transform is independent of the numerical values of the data being transformed. The compiler generates Toolkit instructions for a program's data-independent computations, leaving the programmer to implement the data-dependent branches in assembly language.

3.1 Compiling Data-independent Computations

The first stage of the compilation process uses the partial-evaluation technique described in [7] and [9] to extract the underlying numerical computation from the high-level program. The first stage accepts a data-independent program written in the Scheme, a dialect of Lisp, together with information that specifies the format of the program's inputs and outputs (see [7]). The result of the first stage is a linear block of numerical operations. Unlike the output of traditional compilers for high-level languages, which may include procedure calls and complex data structure manipulations, the compiled output produced by partial evaluation consists entirely of numerical operations—all data-structure operations have been optimized away by partial evaluation. This purely numerical program is further optimized using traditional compilation techniques such as constant-folding, dead-code elimination, and common-expression elimination. The resulting program has a data-flow graph that is acyclic; this becomes the input to the second stage.

The second stage of the compiler uses the graph to produce actual Toolkit instructions for one processor, choosing an order for the numeri-

cal operations that optimizes performance. This is accomplished in two phases. The first phase chooses a preliminary order of operations that attempts to minimize the number of loads and stores and assigns the spilled intermediate results to individual memory banks, with no regard for latency of operations or pipeline delays. This phase is similar to the instruction ordering and register allocation algorithms as described in chapter 9 of [6].

The order of operations produced by the first phase serves as advice to the second phase, which proceeds (machine) cycle-by-cycle attempting to fill all available slots in the pipeline and managing allocation of resources such as memories, buses and registers.

This two-phase strategy seems to give very good performance. It combines the traditional "results-up scheduling," in which the data-flow graph of the computation is analyzed to determine the ordering of the instructions that maximizes immediate reuse of intermediate results and reduce memory traffic, with "cycle-based scheduling," which works forward through the program from the operands towards the results, choosing the order of the instructions incrementally in an attempt to keep the processor pipeline full. In point of fact, however, the basic block produced by the partial evaluator is so large that any non-trivial register allocation technique could be expected to do well.

Compiled programs bear little resemblance to hand-coded programs. In hand-written programs, pipeline stages relating to a particular operation, such as a multiply, tend to be located close to each other. In contrast, the compiled code spreads computations out over time in order to fill in pipeline slots. The compiler schedules loads and stores retroactively, intentionally placing them as far back in time as possible, to leave memory access opportunities available for later instructions. Indeed, loading a register from memory may occur dozens of cycles before the operand is used. Partial evaluation makes this extreme lookahead possible by providing huge basic blocks of straight-line numerical code.

The compiled code tends to be extremely efficient, even when generated from very high-level source code. As a simple example, figure 8 shows a highly abstract definition of a vector addition procedure `add-vectors`, implemented by applying to the addition operator a general transformation `vector-elementwise`, which converts an n-ary scalar function f into an n-ary vector function that applies f to the corresponding elements of n-vectors, $v^1 = (v_1^1, v_2^1, \ldots), v^2 = (v_1^2, v_2^2, \ldots), \ldots, v^n = (v_1^n, v_2^n, \ldots)$, and pro-

duces the vector of results $(f(v_1^1, \ldots, v_1^n), f(v_2^1, \ldots, v_2^n), \ldots)$.

This implementation of `add-vectors` is inefficient in most Scheme implementations. Since the procedures operate on vectors of any length, there must be a run-time loop that counts through the vector index (here implemented in `generate-vector`). An even greater source of inefficiency is that the arity of the procedure argument `f` is not known to `vector-elementwise`, which therefore must explicitly construct lists for `f` at run time.

The compilation is specified by applying `add-vectors` to two vectors, each consisting of four structures called placeholders (see [7]). The placeholders represent the numeric inputs to the compiled program. By specifying the length and the elements of `vector-1` and `vector-2` the placeholders make the program data-independent. Thus, specifying the number of vectors and vector length permits the compiler to generate code after all tests and operations on data structures are performed at compile time. This leaves only the actual component additions as the only "real work" to be performed at run time.

Figure 9 shows the compiled output. The entire computation, including moving data to and from memory, is accomplished in nine cycles. This density of "useful" operations—4 additions in 9 cycles—is untypically low for compiler output, because the program is so short. A more typical result is found in our Solar System integration, where the Scheme source program, written in similarly abstract style, computes and integrates the General-Relativistic gravitational force between pairs of planets. For this program, the compiler generates code in which 98% of the cycles contain floating-point operations.

We have also recently begun to extend the partial-evaluation approach to deal with parallel computations, scheduling the compiler output to multiple Toolkit boards. As with register allocation, it appears that the basic blocks produced by partial evaluation are so large that many techniques can be expected to work well. For example, a prototype compiler by Surati /citeBerlinSurati [21] automatically extracts fine-grained parallelism from conventionally written data independent programs. As a benchmark, the compiler was able to generate automatically-parallelized code for the n-body problem, targeted to an eight-processor Toolkit configuration. This code achieves a speedup factor of 6.2 over the almost optimal uniprocessor code produced by our current Toolkit compiler. This speedup is notable, given the relatively high latency of transmitting data between Toolkit

```
(define (vector-elementwise f)
  (lambda (vectors)
    (generate-vector
     (vector-length (car vectors))
     (lambda (i)
       (apply f (map (lambda (v) (vector-ref  v i))
                     vectors))))))

(define (generate-vector size proc)
  (let ((ans (make-vector size)))
    (let loop ((i 0))
      (if (= i size)
          ans
          (begin (vector-set! ans i (proc i))
                 (loop (+ i 1)))))))

(define add-vectors (elementwise +))

(define vector-1
  (vector (make-placeholder 'vector-1-element-1)
          (make-placeholder 'vector-1-element-2)
          (make-placeholder 'vector-1-element-3)
          (make-placeholder 'vector-1-element-4)))

(define vector-2
  (vector (make-placeholder 'vector-2-element-1)
          (make-placeholder 'vector-2-element-2)
          (make-placeholder 'vector-2-element-3)
          (make-placeholder 'vector-2-element-4)))

(add-vectors vector-1 vector-2)
```

Figure 8: This highly abstract implementation of **add-vectors** works for an arbitrary number of vectors of arbitrary length. Using placeholders, the definition is automatically specialized to add two vectors of length four.

```
((lag (add dz nop low) (d 5)) (rag (add dz nop low) (d 1)))
((lag (add dz nop low) (d 7))
 (rag (add dz nop low) (d 3))
 (lmio m->r r20)
 (rmio m->r r19))
((flop (+ dadd x y) (x r20) (y r19))
 (lag (add dz nop low) (d 4))
 (rag (add dz nop low) (d 0))
 (lmio m->r r3)
 (rmio m->r r26))
((flop (+ &latch dadd x y) (x r3) (y r26))
 (lag (add dz nop low) (d 6))
 (rag (add dz nop low) (d 2))
 (lmio m->r r9)
 (rmio m->r r8))
((flop (+ &latch dadd x y) (x r9) (y r8) (t + r20))
 (lag (add dz nop low) (d 8))
 (rag (add dz nop low) (d 8))
 (lmio m->r r19)
 (rmio m->r r17))
((flop (+ &latch dadd x y) (x r19) (y r17) (t + r3))
 (lag (add dz nop low) (d 9))
 (rag (add dz nop low) (d 9))
 (lmio r->m r20)
 (rmio r->m r20))
((flop (+ &latch) (t + r9))
 (lag (add dz nop low) (d 10))
 (rag (add dz nop low) (d 10))
 (lmio r->m r3)
 (rmio r->m r3))
((flop (t + r19))
 (lag (add dz nop low) (d 11))
 (rag (add dz nop low) (d 11))
 (lmio r->m r9)
 (rmio r->m r9))
((lmio r->m r19) (rmio r->m r19))
```

Figure 9: Compiled output for the program shown in figure 8.

boards (3 cycles per 64-bit word). The speedup for this automatically-extracted parallelism compares favorably with speedup ratios obtained using manual restructuring of code for parallel supercomputing.[5]

3.2 The Programmable Linker

Currently, the compiler handles only straight-line code [6] and permits calls to a library of hand-coded numerical subroutines. This restriction is severe but tolerable for many important applications—many numerical programs consist of long segments of straight-line code separated by just a few run-time tests. For example, integrating a system of ordinary differential equations over an interval with an explicit-formula integrator such as Runge-Kutta requires run-time tests only to determine if the end of the interval has been reached, and, for an adaptive integrator, whether to adjust the step-size. In contrast, an implicit-formula integrator such as Backward Euler requires the solution of systems of linear equations to choose good pivots, thereby generating data-dependent operations. This is currently beyond the capability of the compiler, and the Toolkit numerical subroutine library includes a hand-coded linear-equation solver that can be used in conjunction with compiled code.

Combining compiled and hand-written assembly code is accomplished using a programmable linker. For instance, to produce a program that accomplishes Runge-Kutta integration, one writes a simple assembly-language loop that counts to the end of the time interval. Within each iteration, the loop calls a compiler-generated subroutine that computes the integrand (compiled from source code written in Scheme).[7]

It is amusing to observe that this style of pasting together hand-written and compiled code inverts the traditional role of high-level and low-level programming. Ordinarily, one writes code in a high-level language that may call hand-written assembly-code subroutines for speed-critical applications.

[5] For example, experiments performed at University of Illinois, using manual restructuring of code on a suite of benchmarks for the Cray YMP, report an average speedup factor of 4.0 for 8 processors [12].

[6] Although, as illustrated by the example in figure 8, many programs that are apparently data-dependent become data-independent when the data structures are specified using placeholders.

[7] See [2] for a detailed example.

Here, we write the inner-loops in a high-level language, and compile them for inclusion as subroutines, called from hand-written assembly code.

4 Example: A Breakthrough in Solar-System Integrations

One of our motivating examples in embarking on the Toolkit project was our experience with the Digital Orrery [5]. The Orrery, constructed in 1983-1984, is a special-purpose numerical engine optimized for high-precision numerical integrations of the equations of motion of small numbers of gravitationally interacting bodies. In 1988, Sussman and Wisdom [22] used the Orrery to demonstrate that the long-term motion of the planet Pluto is chaotic. The positions of the outer planets were integrated for a simulated time of 845 million years, a computation in which the Orrery ran continuously for more than three months.

It was natural, then, that our first task for the new Toolkit was to duplicate some of the Orrery's outer planet integrations.[8] This allowed us to check out and debug the Toolkit on a real problem. We quickly implemented a high-order multistep integrator of the Stormer type—written in Scheme and compiled using the Toolkit compiler—and discovered that each board of the Toolkit was about three times faster than the entire Orrery on this program. Some of this speedup was because the Orrery did not have a single instruction to compute square root or divide. In the Orrery, such operations were performed by Newton's method, using a table to get initial approximations.

Wisdom and Holman [26] then developed a new kind of symplectic integrator for use in Solar-System integrations. This integrator is about a factor of ten faster than the traditional Stormer's method, but it appears not to accrue too much error over long time scales.

With the speedups available from the Toolkit and the new integrator, we performed a 100-million-year integration of the entire Solar System (not just the outer planets, as with the Orrery), incorporating a post-Newtonian approximation to General Relativity and corrections for the quadrupole moment of the Earth-Moon system. The longest previous such integration [20] was for about 3 million years. The results of the integration verify the principal results of the Orrery integrations. This confirms that the evolution of the Solar system as a whole is chaotic with a remarkably short time scale

[8] With the completed construction of the Supercomputer Toolkit, the Orrery was officially retired, and was transferred to the Smithsonian Institution in August, 1991.

of exponential divergence of about 4 million years. A complete analysis of the integration results appears in [23].

Our integration used eight Toolkit boards running in parallel. Each board simulated a Solar System with slightly different initial conditions. The evolving differences were compared to estimate the largest Lyapunov exponent[9]. The differences in initial conditions were chosen to be 1 part in 10^{16} in one coordinate of a planet. We found that the chaotic amplification of this difference is such that a 1 cm difference in the position of Pluto at the beginning of the integration produces a 1 Astronomical Unit difference in the position of the Earth at the end of the integration.

Programming the integration proceeded essentially along the lines indicated in section 3.2. The integrator and the force law were written as high-level Scheme programs. The accumulation of position was implemented in quad precision (128 bits), and the required quad precision operators were written in Scheme.[10] The Scheme source was compiled with the Toolkit compiler, and the resulting routines were combined with a small amount of Toolkit assembly code. The compiled code contains almost 10,000 Toolkit cycles for each integration step, and more than 98% percent of the cycles perform floating-point operations.

The host-side control program was also written in Scheme. The Toolkit was downloaded with initial conditions. It was then repeatedly run for 10^6 7.2-day integration steps, with the state uploaded to the host at the end of each 10^6 step segment. Each segment and upload took about 12 minutes of real time. The 100 million year integration took about 5000 such segments, for a total time of about 1000 hours of run time.

[9]The Lyapunov exponent is a measure of the tendency of nearby solutions to diverge.

[10]In hindsight, the use of quad precision appears to have been overly conservative for this problem, and we plan to rerun the computation at ordinary double precision to confirm this.

5 Summary and Conclusions

We consider the Solar System simulation to be a demonstration of the success of the Toolkit project. A breakthrough computation was performed with a reasonable amount of cost and effort, and most importantly, the computation led to important scientific insights. We are currently pursuing applications to circuit simulation and to optimization and control.

In hindsight, there are a few places where the Toolkit architecture could have been improved. There are no datapaths that allow results to be stored in code memory, so that, for example, all repeat counts in the sequencer must be fixed at compile time. The sequencer chip provides no method to view the addresses stored in its internal registers without executing the instructions at those addresses. This makes it difficult to catch and process interrupts, e.g., those caused by parity errors. A better design would make that state more accessible. It would be convenient to set and sense condition codes based upon results of address-generator computations. Also, the ALU and Multiplier share a single result bus which limits the opportunities to use these units simultaneously. These are details arising from our choice of 1989-standard parts. A second-generation Toolkit design would be almost certainly be based upon parts with higher levels of integration, such as high-speed microprocessors. The use of standard microprocessors brings with it standard software, some of which can be used with little modification.

A more serious limitation of our prototype Toolkit is that we have provided only slow communication with the host. This limits applications to those that require very little communication, such as the long-term integration of systems of ordinary differential equations. It is relatively easy to improve this by fabricating a special board with connections to the fast interprocessor communication channels. Such a communications adapter could buffer communications to and from the host, at host-memory speed. Improving this communication is a top priority for future hardware development.

On the software side, we believe that the Toolkit's scientific package and the Toolkit compiler are major steps forward. Nevertheless, our software-support system has a long way to go. While the compiler makes it easy to compile straight-line code, thus automating the translation of large algebraically-specified systems such as force laws, the compiler has no concept of data-structure or of data-dependent conditional jump. Thus, for

example, all of the complex control structures that support implicit integrators with variable stepsize and variable order (such as Gear's method for stiff systems) must currently be tediously constructed in assembly language. Extending the range of applications that can be handled by the compiler is a clear direction for future work.

In summary, despite its prototype status, the Toolkit demonstrates a means practical within the limits of current technology, to provide relatively inexpensive supercomputer performance for a limited, but important class of problems in science and engineering. The key is to avoid the generality—both in architecture and in compilation technology—that cause computers of comparable speed to be expensive to design, build, and program. The result is machines that are not as fast as the fastest supercomputers, but whose price advantage permits them to be used for applications that would be otherwise infeasible.

Acknowledgments

The Toolkit project would not have been possible without continual support and encouragement from Joel Birnbaum. Henry Wu devised the board interconnection strategy. John McGrory built a prototype host interface and debugging software, and Carl Heinzl wrote an initial set of functional diagnostics. Dan Zuras spent his vacation coding high-precision scientific subroutines. Rajeev Surati wrote a simulator that has become a regular part of our software development system. Sarah Ferguson implemented a Toolkit package for adaptive-stepsize Runge-Kutta integration. Karl Hassur and Dick Vlach did the clever mechanical design of the board and cables. Albert Chun, David Fotland, Marlin Jones, and John Shelton reviewed the hardware design. Sam Cox, Robert Grimes, and Bruce Weyler designed the PC board layout. Darlene Harrell and Rosemary Kingsley provided cheerful project coordination.

References

[1] H. Abelson, A. Berlin, J. Katzenelson, W. McAllister, G. Rozas, and G.J. Sussman, "The Supercomputer Toolkit and its Applications," *Proc. of the Fifth Jerusalem Conference on Information Technology*, Oct. 1990. Also available as AI Memo 1249.

[2] H. Abelson, A. Berlin, J. Katzenelson, W. McAllister, G. Rozas, G. J. Sussman, and J. Wisdom, "The Supercomputer Toolkit: A General Framework for Special–Purpose Computing," AI Memo 1329, Artificial Intelligence Laboratory, MIT, November 1991.

[3] H. Abelson and G. J. Sussman, "The Structure and Interpretation of Computer Programs," MIT Press and McGraw–Hill, 1985.

[4] B.J. Adler, "Special Purpose Computers," Academic Press, Inc., 1988.

[5] J. Applegate, M. Douglas, Y. Gürsel, P. Hunter, C. Seitz, G.J. Sussman, "A digital orrery," *IEEE Trans. on Computers*, Sept. 1985.

[6] A. V.Aho, R. Sethi and J. D. Ullman, "Compilers Principles, Techniques, and Tools," Addison-Wesley Publishing Company, Reading, Mass. 1986.

[7] A. Berlin, "Partial Evaluation Applied to Numerical Computation," *Proc. 1990 ACM Conference on Lisp and Functional Programming*, Nice France, June 1990.

[8] A. Berlin and R.J. Surati, "Exploiting the Parallelism Exposed by Partial Evaluation", MIT Artificial Intelligence Laboratory Technical Report no 1414, April 1993.

[9] A. Berlin and D. Weise, "Compiling Scientific Code using Partial Evaluation," *IEEE Computer* December 1990.

[10] Bipolar Integrated Technology, "B2110A/B2120A TTL Floating-point chip set," Bipolar Integrated Technology, Inc., Beaverton, Oregon, 1987.

[11] S. Borkar, R. Cohen, G. Cox, S. Gleason, T. Gross, H.T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P.S. Tseng, J. Sutton, J. Urbanski, and J. Webb, "iWarp: An Integrated Solution to High-speed Parallel Computing," *Supercomputing '88*, Kissimmee, Florida, Nov., 1988.

[12] G. Cybenko, J. Bruner, S. Ho, "Parallel Computing and the Perfect Benchmarks." Center for Supercomputing Research and Development Report 1191., November 1991.

[13] P. Hut and G.J. Sussman, "Advanced Computing for Science," *Scientific American*, vol. 255, no. 10, October 1987.

[14] IEEE Std 1178-1990, *IEEE Standard for the Scheme Programming Language,* Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[15] Integated Device Technology, *High-Performance CMOS Data Book Supplement,* Integated Device Technology, Inc., Santa Clara, California, 1989.

[16] R. A. Kerr, "From Mercury to Pluto, Chaos Pervades the Solar System ," *Science*, Volume 257, pp. 33, July 1992.

[17] J. Laskar, "A numerical experiment on the chaotic behaviour of the Solar System", *Nature*, vol. 338, 16 March 1989, pp. 237–238.

[18] Joel Moses, "Toward a general theory of special functions," *CACM*, 25th Anniversary Issue, August, 1972.

[19] W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, *Numerical Recipes: The Art of Scientific Computing,* Cambridge Univ. Press, 1986.

[20] Thomas R. Quinn, Scott Tremaine, and Martin Duncan "A Three Million Year Integration of the Earth's Orbit," *Astron. J.,* vol. 101, no. 6, June 1991, pp. 2287–2305.

[21] R. J. Surati, "A Parallelizing Compiler Based on Partial Evaluation," S. B. Thesis, Department of Electrical Engineering and Computer Science, M.I.T., June 1992. Also, *Proceedings of the 1992 MIT Student Workshop on VLSI and Parallel Systems*, Charles E. Leiserson, ed., July 21, 1992, pp. 48-1–48-2. Available as MIT Laboratory for Computer Science Technical Report.

[22] G. J. Sussman and J. Wisdom, "Numerical evidence that the motion of Pluto is chaotic," *Science*, Volume 241, 22 July 1988.

[23] J. G. Sussman and J. Wisdom, "Chaotic Evaluation of the Solar System," *Science*, Volume 257, pp. 256-262, July 1992.

[24] S. Tomita, K. Shibayama, T. Nakata, S.Yuasa, and H.Hagiwara, "A Computer with Low-level Parallelism QA-2 — Its Applications to 3–D Graphics and Prolog/Lisp Machines," IEEE Conf. Proceeding of the 13th Annual International Symposium on Computer Architecture,pp. 280–289, June 1986.

[25] S. Ward, "Toward LegoFlops: A Scalable, Modular, 3D Interconnect," *Proc. of IEEE Workshop the Packaging, Interconnects, Optoelectronic for the Design of Parallel Computers*, 1992.

[26] J. Wisdom and M. Holman, "Symplectic Maps for the N-body Problem," *Astron. J.*, Volume 102, p. 1528, 1991.