Software Infrastructure and Tools for the TRIPS Prototype

Bill Yoder, Jim Burrill, Robert McDonald, Kevin Bush
Katherine Coons, Mark Gebhart, Sibi Govindan, Bertrand Maher
Ramadas Nagarajan, Behnam Robatmili, Karthikeyan Sankaralingam, Sadia Sharif
Aaron Smith, Doug Burger, Stephen W. Keckler, Kathryn S. McKinley

Computer Architecture and Technology Laboratory
Department of Computer Sciences
The University of Texas at Austin
`cart@cs.utexas.edu - www.cs.utexas.edu/users/cart/trips`

ABSTRACT

The TRIPS hardware prototype is the first instantiation of an Explicit Data Graph Execution (EDGE) architecture. Building the compiler, toolset, and system software for the prototype required supporting the system's unique dataflow construction, its banked register and memory configurations, and its novel Instruction Set Architecture. In particular, the TRIPS ISA includes (i) a block atomic execution model, (ii) explicit mappings of instructions to execution units, and (iii) predicated instructions which may or may not fire, depending on the outcome of preceding instructions. Our primary goal has been to construct tools to consume standard C and Fortran source code and generate binaries both for the TRIPS software simulators and hardware prototype. A secondary goal has been to build the software infrastructure on standard platforms using standard tools. These goals have been met through a combination of off-the-shelf and custom tools. We present a number of design issues and their resolution in enabling end users to exercise the prototype ISA using familiar tools and programming interfaces. Finally, we offer download instructions for those who wish to test-drive the TRIPS tools.

1 INTRODUCTION

The development of the TRIPS processor has required building a large software infrastructure and set of tools (or *ttools*), including functional and timing simulators; an assembler, linker, and binary utilities; a high-level language compiler and instruction scheduler; a set of optimized and unoptimized runtime libraries; a resource manager to coordinate and control system resources; and a variety of build and test utilities.

To manage the complexity of compiler development, we implemented major components using well-defined interfaces, a variety of Open Source products, and modular construction.

To create a runtime system in a timely fashion with modest resources, we limited the feature set of the operating environment, put the bulk of command and control on a stock desktop *Host PC* running x86/Linux, used off-the-shelf software and development tools for the motherboard, and defined simple but powerful protocols between the Host PC and motherboard and between the motherboard and chips.

A key theme of software development has been to adapt off-the-shelf solutions when possible and to create homegrown solutions when necessary.

The remainder of this paper is organized as follows. Section 2 discusses the TRIPS language toolchain and the approach we used to implement a full-featured software development kit. Section 3 discusses several of the TRIPS software simulators, enabling us to prove out a number of design ideas during every stage of development. Section 4 discusses the TRIPS operating environment, with its use of a host-based resource manager to download, execute, and control TRIPS binaries on the target processors. Section 5 covers the TRIPS build, integration, and testing processes. Section 6 describes the availability of the tools and simulators for the TRIPS prototype.

2 LANGUAGE TOOLCHAIN

The TRIPS toolchain is responsible for consuming source files written in ANSI C, a subset of GNU C, and Fortran 77; applying both classic and novel optimizations to the control flow structures; and generating binary objects that can be linked with standard math and C runtime libraries. Support for C++ and Fortran 90 applications has thus far not been judged critical. Toolchain construction has been guided by a well-designed set of components or *links*, each with clearly specified syntactical and calling conventions. Figure 1 shows the various components of the toolchain.

2.1 Architectural Constraints

The TRIPS architecture has several features that require unusual support in the software toolchain. The ISA defines a *block atomic execution model*, in which program functions are subdivided into variable-size blocks of up to 128 instructions [1]. When fetched, each block is mapped onto a row of architectural registers and a grid of execution units. Each TRIPS block comprises a *header chunk* and one to four *in-*

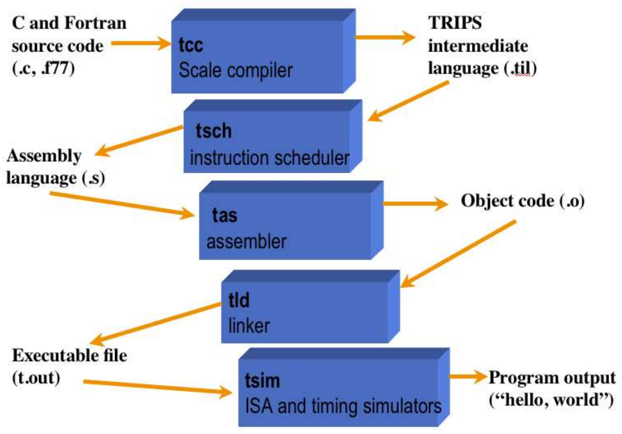
C and Fortran source code (.c, .f77)
tcc Scale compiler
TRIPS intermediate language (.til)
tsch instruction scheduler
Assembly language (.s)
tas assembler
Object code (.o)
tld linker
Executable file (t.out)
tsim ISA and timing simulators
Program output ("hello, world")

Figure 1: Toolchain links.

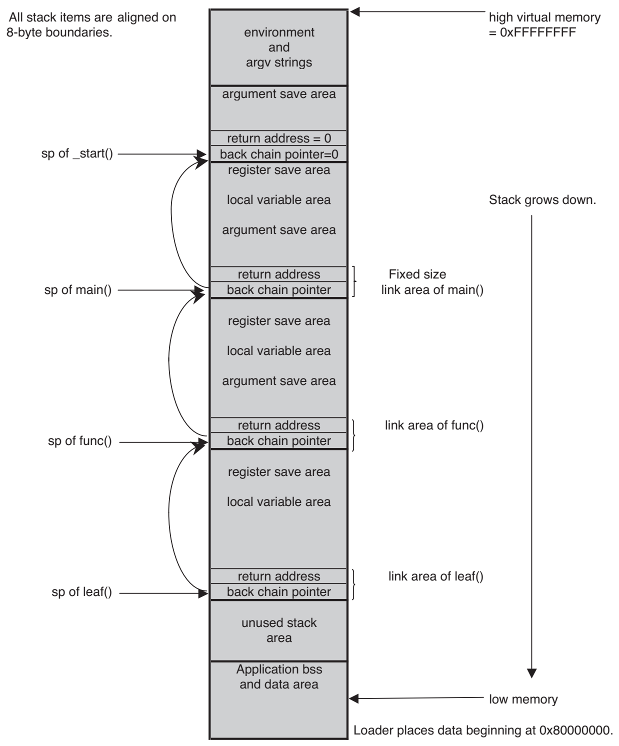

Figure 2: Application memory organization.

struction chunks. Register read and write instructions are embedded in the header chunk, in addition to fields for magic number, block type information, and processor control flags. The ISA places strict limits on the number of instructions in a block, on the number of register reads and writes, and on the combined number of load/store instructions.

Optimally mapping instructions on the grid is critical, due to the TRIPS dataflow execution model, in which each execution node fires when its operands arrive. Cycle counts depend not only on opcode cycles but on operand arrival times, themselves determined by feeds from banked architectural registers, a routing operand network (*OPN*), and a banked L1 data cache.

Despite such contraints, numerous features have made the prototype easier to program, including regular and clearly defined instruction formats; a global 40-bit address space, which enables software to access any chip register or memory location through its unique address; and a uniform processor exception model, in which all processor exceptions occur at block boundaries and present a consistent interface to the software.

Throughout toolchain development, we relied on a robust functional simulator, an accurate timing simulator, and a sophisticated set of performance tools to analyze execution traces, to visualize instruction placement, and to pinpoint critical paths and execution bottlenecks.

2.2 Binary Utilities

The assembler, linker, and other utilities, such as nm, objdump, and ar, are ports of the GNU binary utilities (*binutils*), available from the Free Software Foundation [4].

A number of benefits have accrued from the decision to use the binutils package:

- A leveraged and rich set of functionality based on a mature codebase.
- A large installed user base and an active, experienced development community.
- A common set of application programming interfaces that can be used by other tools, such as the simulators and debugger.

We chose to support a simple version of the Executable Linking Format (*ELF*) to output a small number of string tables and *text*, *data*, and *bss* program sections. Figure 2 shows how the linker lays out application data in virtual memory. The figure also reflects the simple but effective calling convention of the TRIPS Application Binary Interface (*ABI*) [9]. One simplification that made binutils development easier was dropping support for shared libraries–all TRIPS executables are statically linked, which dramatically simplifies installing, versioning, and debugging toolchain components and applications.

The utility front ends are largely platform independent. For example, once we specified the TRIPS data types, such as 64-bit pointers, longs, and doubles and 32-bit integers and floats, the GNU utilities transparently provided support for allocating and accessing TRIPS scalar variables.

However, to support the prototype ISA and high-level compiler. the TRIPS-specific back ends required a significant porting effort. Descriptions of major customizations follow.

2.2.1 Block-oriented rather than line-oriented assembly

The GNU front end assumes that each line of a source file translates into one instruction word. Once the TRIPS Assembly Language (*TASL*) had been defined, encoding the individual 32-bit instructions proved straightforward, simply requiring the tas assembler to parse and translate the source file a line at a time [14].

However, to support the TRIPS block-atomic model, TASL syntax groups instructions and register reads and writes into *blocks*, demarcated with "block begin" (.bbegin) and "block end" (.bend) directives for that block. The tas assembler therefore tracks the individual statements after the opening .bbegin directive and enters them into a memory structure representing the execution grid and architectural registers. When encountering the matching .bend directive, the assembler traverses the data structure, marks instructions that require relocation, uses the GNU Binary File Description (*BFD*) routines to translate x86 Little-Endian formats to TRIPS Big-Endian, and commits the whole block to disk.

Throughout binutils development, a balance needed to be struck between treating instructions as individual 32-bit words and treating "instructions" as variable-size code blocks. In addition, the ISA definition of 32-bit instructions and 64-bit pointers and data types required syntactical support to construct 64-bit data from pieces of the 32-bit instructions. Although the TRIPS prototype restricts global physical addresses to 40 bits and although the *ttools* compiler limits programs to 32-bit virtual addresses, nothing in the ISA or assembly language definition precludes full 64-bit addressing.

2.2.2 Block integrity checking

It is easy to write incorrect TASL code. The tas assembler provides simple error checking during the first pass while parsing the individual instructions, to ensure that instruction formats are correctly specified, that immediate values lie within range, that proper target operands are in place, and that no block contains more than 128 instructions.

In addition, due to the target formats of the ISA, a second round of checking is desireable, to ensure that each block is well-formed. Hence, after each block of instructions is assembled, a two-pass checker is invoked to ensure that source instructions properly target destination instructions, that predicate outputs target instructions which in fact expect predicates, that instructions expecting one or two operands receive them, and so on.

An automated TASL generator tg, which produces thousands of lines of randomized but legal TASL code, proved invaluable in shaking out assembler/linker bugs before compiler readiness. The same tool was used during hardware bringup to test processor functionality.

2.2.3 Limited reach of call-by-offset instructions

The prototype ISA defines a number of *constant instructions* which embed immediate values in 16 bits of the 32-bit word. Using a combination of *generate* and *append* constant instructions (GEN and APP), the compiler can construct efficient instruction sequences to generate addresses and literals.

Additionally, the ISA provides both absolute and relative branching instructions. The CALL and BR instructions support 64-bit absolute addressing, just as the CALLO and BRO instructions contain *offset fields* to specify target addresses.

To save instruction count (4 *vs.* 2 constant instructions), the compiler emits CALLO and BRO instructions for all jump targets. Because the offset field specifies 128-byte chunks and supports a 20-bit offset, any target within a 2^{27} byte-spread is reachable.

However, for very large applications the 20-bit offset field can be insufficient. In such cases, the linker must detect the shortfall, "relax" the current code section, and insert a *trampoline* above the current block which fully specifies the absolute target address with a fabricated BR instruction. The original CALLO or BRO instruction in the block is retargetted to the adjacent trampoline block, which bounces execution to the far-flung address. Code trampolines introduce an extra level of indirection. Fortunately, the need for them rarely occurs.

2.2.4 Disassembling TRIPS binaries

The binary utilities and debugger provide several disassembly routines. Again, the GNU model of one instruction per code word had to be extended to support consuming a header chunk of 128 bytes at once, parsing and rearranging the header into register reads and writes, reading the following one to four instruction chunks, and finally disassembling the individual code words, while maintaining integrity of the complete code block.

2.3 Compiler and Instruction Scheduler

The TRIPS compiler is *Scale* [8], a Java-based set of 700+ classes offering classic optimizations as well as TRIPS-specific optimizations. Scale has proven an ideal research vehicle, supporting several code generators (Alpha, Sparc, PowerPC, TRIPS), configurable optimizations, and compilation phases that can be arranged in arbitrary order.

One early design decision was to partition TRIPS-specific compilation into two major components:

- The Scale compiler, which produces an intermediate form of assembly code.
- An instruction scheduler (tsch), which translates the intermediate output from Scale into low-level dataflow instructions and maps each explicitly onto the execution grid.

As described below, this organization has enabled the generation of a RISC-like, high-level assembly language from the Scale compiler and the generation of a dataflow syntax from the instruction scheduler, with little loss in tool efficiency.

2.3.1 Hyperblock generator

The TRIPS prototype accepts up to 128 instructions per block and executes them in dataflow order and in parallel. Because basic blocks of control-dependent applications typically consist of 5-6 instructions between branches, and because the prototype ISA disallows branches to targets within a block, one of our key challenges has been to aggregate basic blocks into much larger predicated hyperblocks [11].

The original goal of the hyperblock generator was simple: join basic blocks into large hyperblocks, the bigger the better [7]. The early design of the hyperblock generator placed hyperblocking in one of the early phases of compilation, controlling the degree of while and for loop unrolling, if conversions, and function inlining. The compiler broke up and then re-constituted any of the resulting blocks that exceeded any of the prototype constraints, such as those containing more than 32 loads and stores.

What we discovered:

- The prototype instruction contraints are *hard* limits. Even if one block in a million exceeds a hardware constraint, the resulting binary cannot execute properly.
- It is difficult for early analysis to form accurate instruction estimates. In some cases, instruction counts will be overly pessimistic due to later optimizations that remove instructions. In other cases, instruction counts will be overly optimistic, ignoring potential store instructions for spilling by the register allocator, which is invoked far downstream from the hyperblocker.
- Block-splitting is difficult, in itself requiring not only *reverse if-conversion* but analysis to build new hyperblocks from resulting fragments. Such regenerated code is typically inefficient and requires redundant hyperblocking algorithms.

Consequently, a significant structural revision was forced, in which the hyperblock generator migrated to the compiler back end, to good effect. Unrolling while loops has subsequently migrated; for loop migration is soon to follow. Other compiler optimizations are anticipated:

- Hyperblocking based on execution profiles, to pull "hot" basic blocks into the same hyperblock.
- Reducing the number of block exits, to train the prototype's branch predictor in execution patterns.
- Instruction merging, to minimize the number of redundant instructions.

2.3.2 TRIPS compiler driver

The dynamic nature of the toolchain has required frequent and sometimes intricate modifications of command line options to various components. For example, a variety of options for hyperblocking, both in the compiler front end and back end, coexisted for a number of months. To hide these changes from the end user, while providing full access to individual options, the tcc compiler driver was developed.

Originally implemented as a simple bash script, tcc has grown to 2,500+ lines of Perl to support dozens of command-line options, both TRIPS-specific and gcc-compatible, such as canned optimizations (-O3 and -O4), debugger options (-g), and component options, including -D to the preprocessor, -Wc,<opt> to the compiler, -Wa,<opt> to the assembler, -Wl,<opt> to the linker, and –verbose to all. This gcc-friendly support has aided importing large quanities of existing applications with only minor Makefile edits, as in changing CC=gcc to CC=tcc.

2.3.3 High-level vs. low-level assembly languages

Programmers are familiar with RISC-like, algebraic-formulated languages which execute in program order. We therefore defined the TRIPS Intermediate Language (*TIL*) [10] as a user-friendly target language, employing a simple and consistent syntax:

opcode result, operand1 [, operand2]

Example: The following code for block main$4 shows how architectural registers G3 and G12 are read into temporary registers T0 and T1 of the execution grid, how some amount of decision-making occurs to determine the outcome of a branch, and how grid outputs are written back to registers G12 and G13 before the block terminates.

```
.bbegin main$4
        read    $t0, $g3
        read    $t1, $g12
        mov     $t2, $t0
        addi    $t3, $t1, 1
        extsw   $t4, $t3
        tlti    $t5, $t4, 10
        tnei    $t6, $t5, 0
        bro_t<$t6>      main$3
        bro_f<$t6>      main$5
        write   $g12, $t4
        write   $g13, $t2
.bend
```

The instruction scheduler, charged with mapping instruction blocks onto the execution grid, consumes TIL files and produces target-form output.

Example: In this translated form of main$4, the TASL specifies exactly how inputs are fed into the grid, which execution nodes ((N[0]-N[127]) are brought into play, and where the execution nodes direct their results.

```
.bbegin main$4
;;;;;;;;;;; Begin read preamble
R[3] read G[3] N[3,0]
R[0] read G[12] N[0,0]
;;;;;;;;;;; End read preamble
N[3] <0> mov W[1]
N[0] <1> addi 1 N[4,0]
N[4] <2> extsw N[8,0] W[0]
N[8] <3> tlti 10 N[12,0]
N[12] <4> tnei 0 N[16,0]
N[16] <5> mov N[20,p] N[24,p]
N[20] <6> bro_t B[0] main$3
N[24] <7> bro_f B[1] main$5
;;;;;;;;;;; Begin write epilogue
W[0] write G[12]
W[1] write G[13]
;;;;;;;;;;; End write epilogue
.bend
```

The TRIPS prototype maps the virtual node numbers above to physical grid coordinates, by using 2 of the 7 bits of the execution node number for row specifier (*y*-direction), 3 for frame specifier (*z*-direction), and 2 for column specifier (*x-direction*).

However, the ISA itself dictates no such coordinate system, and the hardware can choose to map instructions in any way that is consistent.

Note also that the order of TIL instructions dictates execution order. No such assumptions hold for the TASL–targets fire whenever their operands arrive.

By supporting both TIL and TASL, the TRIPS toolchain enables programmers to express and examine their assembly code in a familiar manner, while enabling the instruction scheduler to specify instruction placement precisely and enabling the assembler to translate source files easily into machine code.

2.4 Runtime Libraries

2.4.1 C runtime library

The compiler developers chose the Dietlibc embedded C runtime library authored by Felix von Leitner for its small footprint, base set of functionality, and portability [13]. TRIPS-specific customizations have included:

- Implementing a generic system call interface, so that the compiler treats system calls as function calls, whose definitions are automatically generated by the m4 macro preprocessor with register setups, traps to the SCALL instruction, and return values.
- Increasing the amount of inlining used by the compiler and grouping like functions into multi-compilation units, so that the compiler can inline across modules.
- Defining routines in the C runtime startup module (crt.o) to interface with the program loader and set up the stack, provide software floating point division, determine the base physical address of the chip configuration space, implement setjmp(), and so on.
- Increasing the amount of buffering used by printf() and the memory manager in calls to malloc(), to minimize runtime overhead.

2.4.2 Math libraries

We chose the SunSoft Freely Distributable LIBM (*libfdm*) C math library for its transcendental math functions [12]. The *libfdm* routines are IEEE-754 conformant, portable, and well-tested. The intention is to replace key functions such as sqrt() with those from the IBM IEEE math library (*MathLib*) [5], due to its *accurate table method*, which executes efficiently on the prototype.

Due to area constraints, the prototype includes no floating-point divide unit and no FDIV instruction. Starting from John Hauser's platform-independent implmentation of the IEEE Standard for Binary Floating-Point Arithmetic [6], we extracted the 64-bit software divide routines, hand-tuned them for efficient execution, and "baked" them into the C runtime startup module crt0.o, available to all applications.

2.4.3 Optimized string and memory library

The dietlibc runtime library is designed to be compact and portable across platforms but not necessarily optimal for a given platform. For example, the dietlibc strcpy() routine has at its kernel:

```
    while (*dest++=*t++);
```

This implementation is portable and reasonably efficient, but can be re-written to take advantage of large blocks of predicated instructions. Developers have TIL-coded strcpy() and other key string and memory operations, grouping them into an *optimized library* that the linker consults before falling back to the corresponding dietlibc routine.

A final step involves running the hand-coded routines through the TRIPS *simulated annealer*, which assembles the source, executes the code on the timing simulator or hardware, records the cycle count, re-arranges instructions on the grid according to a variety of heuristics, and repeats the process until a minimal threshold of cycle count is achieved.

The ultimate goal is for the Scale compiler and instruction scheduler to produce equivalent code quality to hand-tuned code, with their automated loop unrolling, inlining, hyperblocking, other transformations, and instruction placement.

2.4.4 Multiple libraries

The default toolchain behavior is to generate optimized TRIPS binaries with array address strength reduction, dead variable elmination, copy progation and useless copy removal, loop invariant code motion, scalar replacement, tree height reduction, and other optimizations, but without hyperblocking (-O3). For fully optimized applications, a set of runtime libraries compiled with hyperblocking is available and presented to the linker when users compile their applications with -O4. Also, for users of the debugger, the -g flag causes the linker to link debuggable libraries compiled neither with the above optimizations nor function inlining so that debugging runtime functions closely resembles those of conventional architectures.

3 SOFTWARE SIMULATORS

We have developed a variety of software simulators for TRIPS, from a simple ISA simulator to a detailed Verilog chip simulator.

Simulators in the *ttools* release include a functional simulator (tsim_arch) and a cycle-accurate simulator (tsim_proc). Despite disparate goals, both share the same C++ code base and key attributes:

- A built-in program loader, using the *BFD* utility routines, to load the code and data into target memory, to create TLB entries for the text and data segments as defined by the linker, to initialize the process registers and stack, and to schedule the executable to run.
- Support for argv, argc, and envp program variables and for stdio, stdout, and stderr console I/O.

- A common execution model of fetching the next block of instructions, mapping the instructions onto the architectural registers and execution grid, injecting block inputs into the grid, executing each instruction as its operands become ready, forwarding results to target execution nodes, and outputting register writes and memory stores.

- A set of proxy services for handling system calls.

- Common tracing and debugging output formats.

- A set of statistical collection routines.

Both simulators model single processor behavior. A system simulator (tsim_sys) has been developed to simulate execution of multiple processors, enabling the user to download and execute one or more applications under the control of the TRIPS Resource Manager, discussed in section 4.

To manage complexity and guarantee deterministic behavior, all simulators execute as single-threaded processes on x86/Linux-based hosts.

3.1 Functional Simulator

The tsim_arch functional simulator is an architecture-level simulator intended to model accurately the TRIPS processor architecture. It is equivalent to a traditional instruction-set simulator (or functional emulator) and does not provide realistic cycle counts.

3.1.1 Execution model

Beginning with the header chunk of the first fetched block, typically from the _start() routine of crt0.o, tsim_arch maps the block reads onto the designated architectural registers and maps the individual instructions onto the execution grid. Each register read is forwarded to its target execution node or nodes.

Beginning at the "first" active execution node, the simulator visits one node after another, executing each instruction if its input operands are available and forwarding the result to the target node or nodes, until it has traversed the grid. Then the simulator begins a second pass through the grid, again firing instructions with ready operands, and a third, and so on, until the *block completion logic* is satisfied, typically when all store instructions have fired, or until a program exception occurs. At block completion time, the simulator commits all block outputs, both register writes and memory stores. The next block is fetched, based on the virtual address specified in an executed BR or CALL instruction.

Through the TLB entries, the simulator translates all memory references to physical addresses. If a TLB miss or other fatal program exception occurs, a block playback capability outputs an execution trace for the programmer.

As a convenience, the simulator maintains symbol table information from the binary in order to print block and variable names during trace operations.

3.1.2 System call support

Both simulators provide limited system call support, driven by the requirements of targeted applications. Currently, the simulators support the following: brk(), close(), creat(), exit(), fstat(), getpid(), gettimeofday(), lseek(), lstat(), open(), read(), stat(), time(), unlink(), and write().

The C runtime library translates system calls into SCALL instructions, which trap into the simulator's system services module. The trap handler in turn proxies the service request on the host, according to the POSIX definition of the call.

3.2 Timing Simulator

In addition to simulating TRIPS processor functionality, (tsim_proc) offers a detailed, cycle-accurate model of the prototype TRIPS processor. It models the internal organization and latencies of the processor and is intended to support processor-level performance analysis.

Whereas tsim_arch models the behavior of architectural registers and execution nodes at the software-visible level, tsim_proc additionally models the microarchitecture structures within each. For example, tsim_proc details the behavior of individual execution nodes when sent commands from the global control unit, when reading packets from and writing packets to the operand network, when filling and consuming internal buffers, and during block flushes and commits–in addition to actually selecting, executing, and retiring instructions.

On a 2.9GHz Pentium 4, tsim_arch can execute 800,000+ simulated instructions per second. Although the increased level of detail causes tsim_proc to execute 300-800 times slower than tsim_arch, we have been pleased to confirm that performance metrics obtained from tsim_proc closely match those of the actual prototype, discounting memory latencies outside the modeling capability of the timing simulator. Statistical output from tsim_proc includes instructions committed and those flushed (both speculatively and non-speculatively), branch predictor hits and misses, icache and dcache hits and misses, speculative load hits and misses, forwarded stores, OPN packet reads and writes, busy and stalled cycles, buffer occupancy rates, and total cycles per block.

4 SYSTEM SOFTWARE

This section focuses on system software as it executes on the prototype hardware. However, much of this same software executes equivalently on our system simulator platform. Those interested primarily in using the software toolchain and simulators can skip to Section 6, "Release Information."

4.1 System software goals

System software goals include enabling efficient hardware bring-up; enabling users to download, execute, observe, manage, and debug applications; demonstrating the performance and capabilities of the prototype on targeted workloads, such as on EEMBC and SPEC CPU2000 benchmarks; and supporting full hardware utilization of processor, chip, memory, and

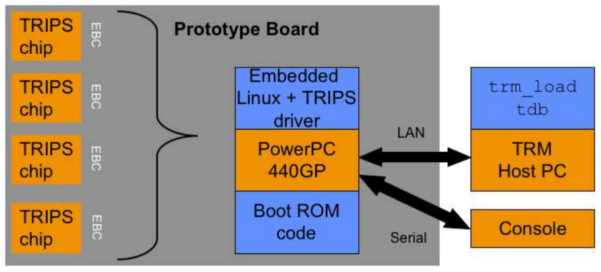

Figure 3: System software components.

board resources.

4.2 System software components

The TRIPS system employs these components:

- An x86/Linux Host PC to manage the motherboards and provide a networked file system.
- A local area network to support communication among the motherboards and Host PC.
- A hardware debugger to provide initial access to the PowerPC 440GP registers, TLB entries, peripheral bus, SDRAM controller; and to program the onboard flash memory.
- A bootloader programmed into flash memory.
- An embedded Linux kernel executing on the motherboard.
- A device driver and daemon process to mediate communication between motherboard and target processors.
- A cross-compilation toolchain for the embedded PowerPC 440GP processor.
- A resource manager running on the Host PC to monitor and control the target processors,

Figure 3 shows components of the system software.

To manage complexity, system software is organized in layers, from device driver on the motherboard to end-user clients on the Host PC. To leverage engineering resources, much of the software derives from Open Source and off-the-shelf products.

4.3 Board Software

Up to four dual-core TRIPS chips on individual daughtercards populate the prototype motherboard. The embedded PowerPC 440GP processor serves two major purposes:

- To read and write addresses in the global memory space shared among processors.
- To service processor interrupts and pass them on to the Host PC.

A custom Linux 2.6 kernel module has been developed to manage PowerPC 440GP bus transactions with the External Bus Control unit on each of the four TRIPS chips. Extensions to the software on the PowerPC 440GP will handle TRIPS system calls locally.

4.3.1 Boot process

Wolfgang Denk's Free Software project, the capable and Open Source u-boot bootloader and Linux 2.6 kernel tree, greatly mitigated the complexities of booting the PowerPC 440GP into Linux [2].

During a motherboard reboot, the PowerPC 440GP is reset and begins executing u-boot from flash memory to initialize processor registers and peripherals, including its UART, Ethernet, and SDRAM controllers. After copying itself to SDRAM and jumping there, the bootloader connects across the LAN connection to the Host PC to download the kernel into processor memory and transfer execution to the kernel entry point.

After re-initializing board components, the kernel connects across the LAN to learn its hostname and network parameters (via DHCP), to set the onboard clock (via NTP), to mount its root file system (via NFS), and to accept logins (via SSHD).

4.3.2 EBI adapter

At the end of boot process, a boot script creates a /dev/trips device file, installs the ebi_driver.ko kernel module, and invokes the ebi_adapter user-level process.

After detecting which TRIPS chips are physically present on the board, the EBI adapter spawns two threads–one to listen on a well-defined socket for Host PC commands from the TRM and another thread to wait on processor interrupts potentially from all four chips.

4.4 Resource Manager

The TRIPS Resource Manager (*TRM*), a user-level process on the Host PC, functions as a light-weight operating system, initializing the chips, allocating memory, downloading applications, servicing system calls, and monitoring performance.

4.4.1 Low-Level communication

We defined a low-level Hardware Abstraction Level (*HAL*) consisting of a protocol and API supported between the TRM executing on the Host PC and the EBI adapter executing on each motherboard. The protocol is simple but powerful, consisting of individual read and write requests, which always originate from the TRM, and a signalling mechanism by which the EBI adapter can notify the TRM of processor exceptions.

4.4.2 System call support

The prototype relies on the Host PC for all system services, including console and file i/o, memory allocation, wallclock access, and process termination.

When the application issues a system call, such as an open(), read(), or write(), a well-defined sequence occurs, as specified in [9]:

- System call parameters are written to registers G3 and following, according to compiler calling conventions.
- The C runtime library places the numeric id of the system call in G0, updates the stack pointer in G1, sets the return address in G2, and issues the SCALL instruction.
- The processor sets an interrupt bit in the External Bus Controller unit and halts.
- The device driver on the PowerPC signals the user-level EBI adapter, which in turn raises an exception for the TRM.
- During its main loop, the TRM pulls the request from its network queue and discovers which processor on which chip on which board has caused the exception.
- After determining that the exception is due to a system call, as opposed to a DTLB translation error, a divide-by-zero error, a breakpoint, or other exception, the TRM reads the numeric ID of the system call from G0, reads the pass parameters, depending on the type of system call, and executes the call on behalf of the TRIPS application.
- Upon proxying the call on the Host PC, the TRM updates processor memory locations as needed and writes the system call return value into G3.
- Finally, the TRM clears the *Processor Status Register* to restart processor execution.

With remote servicing complete. the application resumes execution at the return address specified in G2. According to compiler calling conventions. the TRM leaves the return value of the system call in G3. If error occurs, such as an invalid write(), the return value will be -1 and the TRM will set program variable errno appropriately, according to POSIX convention.

4.4.3 Client-level communication

Running as a background process, the TRM server supports multiple users and multiple applications executing on multiple processors. Communication between TRM server and the user's client application occurs across a TCP/IP socket, connected to the user's terminal session. To simplify control flow, all client requests to the TRM are *blocking*. For example, when the user requests the TRM server to download an executable file from the Host PC into chip memory, the user's session will wait for the request to complete.

4.5 TRM Clients

A number of x86/Linux client utilities have been developed so that users can easily allocate system resources on the Host PC, download applications, and run them. Additional clients enable users to configure clock speeds and *morph* modes, to examine and modify current application status, and to produce statistical information.

4.6 Symbolic Debugger

A special TRM client is the TRIPS symbolic debugger, tdb. Although many applications can be instrumented with printf() statements or debugged on other platforms, the desire has been to support TRIPS-specific data types, breakpointing, block stepping, memory and register accesses, and runtime libraries.

Due to its powerful feature set and familiar interface, we chose to port the GNU debugger gdb to the prototype. The work was highly leveraged, based on:

- The portable x86/Linux gdb front-end, for command-line editing and platform-independent operations.
- The BFD library, used to read ELF executables. which was already part of the TRIPS binary utilities,
- The TRIPS opcode library, already built into the assembler, to disassemble instructions on a per-block basis.

Those portions of tdb requiring further customizations are discussed below.

4.6.1 Host-target protocol

gdb normally offers support for remote debugging in two forms:

- A platform-specific *stub library* is linked into each target executable and handles interrupts and commands from the host.
- A gdbserver runs as a proxy on the target as a separate, heavy-weight process and controls the behavior of the target application.

Both forms use a low-level string-based protocol between host and target. Building on the capabilities of the TRM, we defined a higher-level protocol and an API to communicate between (a) the gdb backend on the Host PC and (b) the EBI on the motherboard. The API includes routines to download and execute programs, to set and clear breakpoints, to continue and block-step execution, to read registers and virtual memory locations, and to map virtual to physical addresses.

From the debugger's perspective, the TRM is both:

- A virtual processor, responding to memory and register requests and breakpointing commands.
- An operating environment, with session management, program loader, file i/o, and system call support.

4.6.2 Symbol table entries

The binary utilities and gdb front end already provide extensive support for symbol table, data type, and line number information in the form of *stabs* entries. However, the Scale compiler required extensive effort to generate symbol and line information. A major challenge has been to match source lines against instruction blocks, as blocks typically contain numerous source lines. If a code block subsumes multiple source lines, which line should the compiler identify as the "right" line to represent the whole block? Although the compiler can emit code with one-line-per-block, the resulting binary is artificially and excessively bloated.

To that end, the GNU Dynamic Data Debugger (ddd) has been ported to the prototype system as a front end to tdb [3]. By providing a friendly user interface and scrollable visual listings, ddd enables users to orient themselves in their source code while stepping through their application.

4.6.3 Breakpoints

The processor offers both *break-before* and *break-after* hardware breakpoints on a per-block basis. Built on top are TRM functions to set and clear both types as well as GNU platform-independent functions. tdb uses a mix of functions for its breakpointing support.

When execution has stopped on a breakpoint and the user issues a *continue* command, tdb uses TRM functions to remove the current *break-before* breakpoint, set a *break-after* breakpoint, and continue execution. When it almost immediately hits the *break-after* breakpoint, tdb uses TRM functions to restore the original *break-before* breakpoint, clear the current *break-after* breakpoint, and resume execution transparently. In this manner, the user interacts with the debugger in familiar fashion.

Note that due to the TRIPS block-atomic execution model, single-stepping in tdb translates to *block-stepping* through the binary; that is, breakpoints are set at block boundaries as the processor executes a whole block at once. Externally, program order is maintained, but within a block instructions fire in non-deterministic order, governed by latencies within the operand network and memory request fulfillments. A future enhancement will enable the programmer to pinpoint processor faults, such as a divide-by-zero instruction; namely, an ISA simulator sewn into the debugger will replay the block in software to identify the offending instruction.

4.6.4 Stack unwinding

As with any gdb port, we needed to implement *stack unwinding* routines to provide backtracing, pass parameter, and local variable information. Our debugger required two key enhancements:

- In order to examine the current stack frame at function invocation, the compiler changed its stabs generation to ensure that breakpoints are set always *after* the function prologue.

- In order to determine which values are pushed on the stack, the debugger uses extensive disassembly and analysis of the prologue, to trace the possible source of STORE instructions back to the stack pointer.

4.6.5 Physical addressing

gdb restricts itself to virtual addresses, as determined by the linker at link time and the call stack at runtime. In order to support physical memory lookups, we developed a private interface to the TRM, by which the debugger can query the physical address of a given virtual address. The gdb *e*(X)*amine* command has been extended to provide such lookups.

Example: The following interaction is based on the *dhrystone* benchmark, when a breakpoint is set at function Proc_7() and program variable Int_Loc is examined.

```
(tdb) c
Continuing.

Breakpoint 1, Proc_7 (Int_1_Par_Val=2,
Int_2_Par_Val=3, Int_Par_Ref=0xffffeec0)
    at dhry.c:473
*Int_Par_Ref = Int_2_Par_Val + Int_Loc;
(tdb) ptype Int_Loc
type = int
(tdb) p Int_Loc
$1 = 1
(tdb) p &Int_Loc
$2 = (One_Fifty *) 0xffffee50
(tdb) x/p &Int_Loc
0xffffee50:  paddr=0x0147ffee50
```

We learn from this interaction that Int_Loc is an integer variable, is defined as an application-specific One_Fifty type, holds a value of "1", resides on the stack at virtual address 0xffffee50, and occupies physical memory location 0x0147ffee50.

5 DEVELOPMENT PROCESSES

5.1 Source Code Management

We use the Concurrent Version System (*CVS*) to automate nightly builds from several local and remote source repositories. Nightly *cron* jobs pull from the repositories, build toolchain and supporting components, run some number of "sanity checks" against the newly built components, and if successful, create a 130MB root file system with those components. Unless one of the key components breaks, developers enjoy fresh bits each morning.

5.2 Regression Testing

To verify operation of constantly evolving toolchain components, thousands of nightly regression tests are run after the root file system is installed. Depending on a weekly rotation schedule, tests last from 2 to 16 hours.

5.2.1 SPEC CPU2000

Because the prototype supports both streaming and irregular control flow applications, the SPEC CPU2000 benchmarks represent important workloads. To maintain the integrity of the SPEC CPU2000 framework, care has been taken to customize only platform-specific information, such as pointer type definitions and Fortran formatted-output specifiers, and to leave in place the original runspec tool and directory structure from the SPEC distribution media.

With a few simple commands, users can specify the toolchain version and compiler flags to use, build the benchmarks in parallel, package the binaries and download them to the Host PC, run them on the hardware, and generate email reports, Webpages, and graphs showing block and cycle counts.

6 RELEASE INFORMATION

The current release package of *ttools* is *TRIPS Tools, Version 1.0*. It is freely available by means of a Web distribution page. A description of release contents follows.

6.1 Release Components

A variety of x86/Linux-based tools, TRIPS binaries, and sample sources are included in the release:

- C and Fortran cross-compilers, binary utilities, ISA and cycle-accurate simulators, and performance analysis tools.
- Application and system header files.
- Optimized and unoptimized runtime libraries.
- C and Fortran test files.
- Source code for hand-optimized routines.

Note that the package does not include system software, such as the resource manager, system simulator, debugger, and hardware-specific components.

Note also that the cross-tools are supported only on x86/Linux systems.

6.2 Manuals and Reference Information

The following documents are included in the release:

- *Tools Quickstart Guide*
- *Processor Reference Manual 1.2*
- *Chip Reference Manual 1.0*
- *Performance Manual*
- *Intermediate Language (TIL) Manual*
- *Application Binary Interface (ABI) Manual*
- *Assembler and Assembly Language (TASL) Specification*
- *Object File Format (TOFF) Specification*

A variety of published papers are also included, as are slides from the HPCA-12 full-day tutorial, "Design and Implmentation of the TRIPS EDGE Architecture."

6.3 Download Instructions

A request page for the *ttools* software is simple to fill out:

http://www.cs.utexas.edu/users/cart/trips/ttools-req/

The form asks you to enter name, institution, email address, and technical interest. Within 2-4 days, you will receive a download invitation and details concerning how to install and configure the 27MB gzipped package.

REFERENCES

[1] D. Burger, S. Keckler, K. McKinley, and et al. Scaling to the end of silicon with edge architectures. In *IEEE Computer, 37 (7)*, pages 44–55, July 2004.

[2] W. Denk. U-boot - open source firmware for embedded powerpc, 2007. http://www.denx.de.

[3] F. S. Foundation. Ddd - data display debugger, 2005. http://-www.gnu.org/software/ddd.

[4] F. S. Foundation. Gnu binutils, 2006. http://sources.redhat.com/-binutils.

[5] S. Gal and B. Bachelis. An accurate elementary mathematical library for the ieee floating point standard. In *ACM Transactions on Mathematical Sofware,, Vol 17, No. 1*, pages 26–45, March 1991.

[6] J. Hauser. Softfloat, 2002. http://www.jhauser.us/arithmetic/-SoftFloat.html.

[7] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, June 1992.

[8] K. S. McKinley, J. Burrill, B. Cahoon, J. E. B. Moss, Z. Wang, and C. Weems. The scale compiler. Technical report, University of Massachusetts, 2001. http://ali-www.cs.umass.edu/∼scale/.

[9] A. Smith, J. Burrill, R. McDonald, and et al. Trips intermediate language manual - tr-05-22. Technical report, University of Texas, March 2007.

[10] A. Smith, J. Gibson, J. Burrill, and et al. Trips intermediate language manual - tr-05-20. Technical report, University of Texas, March 2007.

[11] A. Smith, R. McDonald, R. Nargarajan, K. Sankaralingam, D. Burger, K. McKinley, and S. Keckler. Dataflow predication. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*, pages 236–245, 2006.

[12] SunSoft. Freely distributable libm, 2004. http://www.netlib.-org/fdlibm/readme.

[13] F. von Leitner. Diet libc - a libc optimized for small size, 2006. http://www.fefe.de/dietlibc.

[14] B. Yoder, R. McDonald, and et al. Trips assembler and assembly language manual - tr-05-21. Technical report, University of Texas, September 2003.