1 Introduction

Mable is an instruction-level machine simulator framework which can be applied to a wide range of architectures. The goal of Mable is to provide a workbench for running, debugging, analyzing and tuning applications for a proposed machine. Architectural features such as multiprocessing, speculative execution and superscalar issue make it difficult to verify software correctness and predict performance. To accurately predict performance we need to tune applications to take advantage of these architectural features, and to do so we need not only a simulator but also a set of program development tools. The tuned applications can then be simulated and analyzed to accurately evaluate architectural design trade-offs. Simulators based on Mable support all phases of this process.

Mable simulators are efficient in three ways. First, the simulators emulate the target instruction set efficiently. To be able to run realistic applications the simulation system must not have excessive overhead. With Mable we achieve slowdown factors of 20-200X, depending on the level of detail of the simulation. Second, the simulation system makes efficient use of the simulator user's time by supporting the standard program development loop, including compilation, program execution, debugging and profiling. Mable supports tools for all of these phases of application development. Third, Mable simulators make efficient use of existing software tools and operating system services on the host machine and thereby save in development time of the simulation environment. For instance, we have built Mable simulators which allow the application programmer to use an unmodified serial debugger to debug emulated programs for a multiprocessor or for a uniprocessor with substantial architectural differences from the machine for which the debugger was intended.

Other simulation systems have achieved similar performance but have not allowed the reuse of existing software tools. The development time for a Mable simulator is far shorter than for other comparable systems because we are able to leverage off of existing software.

Mable is not a simulator or a toolkit. It is an idea which can be used to rapidly build simulation environments for a wide range of target architectures, allowing flexible experimentation with new architectural features. We present simulators built with the Mable framework for three different architectures: a multiprocessor with essentially the same instruction set architecture as the host machine, a superscalar processor which uses the same instruction set as the host machine but includes additional support for speculative execution, and a dual-issue processor with a significantly different instruction set architecture than the host. These three examples form a progression from a processor which is very similar to the host machine to one which is very different. Despite these differences we are still able to provide a program development environment using many existing tools.

The next section describes the basic simulation framework and explains how simulators based on the Mable technique can support efficient emulation while allowing the use of existing software development tools. Given this framework the following three sections describe different applications of the technique: Section 3 describes the MP Mable multiprocessor simulator, Section 4 describes the Tsim simulator for the superscalar TORCH processor, and Section 5 describes PPsim, a simulator for a dual-issue embedded processor. Finally we compare Mable to other simulation approaches.

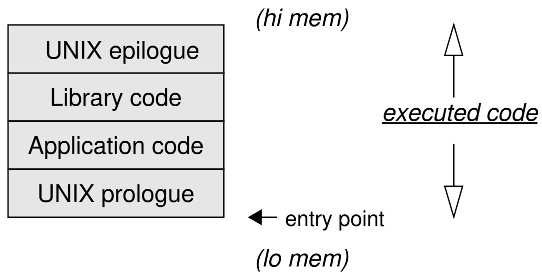

Figure 1: UNIX executable image

2 Theory of Operation

Mable is a framework for building architectural simulators. To create a new Mable-based simulator one must write an instruction emulation engine and optionally a memory simulator for the particular architecture to be simulated. A compiler for the target architecture is also required in order to build the application program, although as we will discuss in Sections 3-5, parts of the host machine's compiler system can often be used. The simulator modules are then linked with the application program in a single address space. The next subsection describes how the instruction emulator and the target application are linked together. Then we describe the structure of the instruction emulator and how it interacts with the operating system and programming tools.

2.1 Simulator Structure

Since one of the goals of Mable is to use existing programming tools it is important for the structure of the emulated application running under the simulator to be similar to a normal host machine application. We therefore assume that the emulated program has the same format as executables for the host operating system, which in the case of our examples is UNIX™.

The text (executable) portion of a UNIX binary consists of a standard prologue followed by the application's object files and library files (see Figure 1). The prologue file (*crt1.o*) provides a facility for passing command-line arguments to the main routine and serves as a standard starting point for the application. When executed, the code in *crt1.o* sets up some execution state and then jumps to the main routine of the application. However, in the case of a target machine application, the application code may contain instructions which the host machine cannot execute.

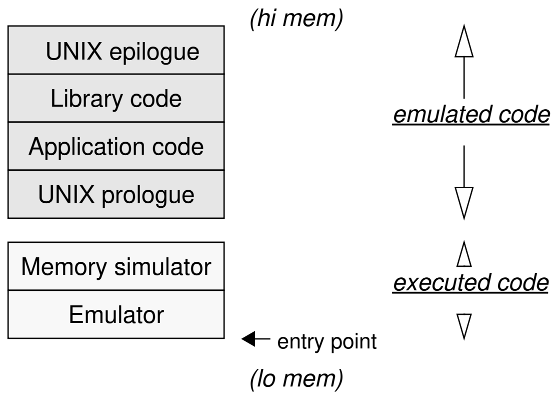

Figure 2: Executable image modifications for Mable

We construct a Mable simulator by inserting a new prologue before the original one. The new prologue actually contains the target-machine instruction emulator (see Figure 2). Thus when the OS jumps to the prologue to start the program it actually starts the simulation engine.

The steps to build an application to be simulated in the Mable framework are straightforward. The target application source code is first compiled in the normal manner using a compiler system which generates the target instruction set. Then in a new linking step a pre-compiled instruction emulator and a memory system simulator are added before the standard UNIX prologue routine and the application program. The symbol tables in the simulator object files are stripped to avoid name conflicts with the application, so the final executable only includes symbols associated with the target program. Because the instruction emulator is the first object file in the executable the simulation routine is executed when the application is invoked. The simulator then has control and starts fetching and simulating code from the original entry point in the application program. Even though the application code is in the text section of the executable it is never directly executed by the host processor.

2.2 The Simulation Engine

The core simulation engine is a tight fetch and execute loop which emulates the instructions in the user application. This loop reads application program instructions as data, decodes them and fetches their arguments. A jump table is then used to rapidly vector to the simulation sequence for the particular target instruction. In these simulation sequences, Mable exploits the fact that the target machine's instruction set is similar to that of the host machine to make instruction emulation more efficient. In many cases, a register/memory or register/register instruction can be simulated with a single host machine instruction. This means that the overhead for each instruction simulated is usually dominated by the time spent in the fetch-and-execute loop to fetch the instruction, fetch its operands, and jump to the correct simulation sequence. Similarities between the host and target architectures also make the emulation loop much easier to implement.

The basic simulation loop can be described by the following steps, typical for a register/register instruction:

1. load an instruction from the code to be emulated.
2. load two source register values from the machine context.

3. decode the destination register address.
4. jump to the proper simulation sequence.
5. execute the simulation sequence.
6. store the result to the machine context.
7. update the program counter.
8. loop back to fetch the next instruction.

The Mable framework allows for easy experimentation with different architectural choices. The state of the virtual machine, including the register file, is kept in memory. This allows the target machine to have different state than the host (e.g. a different register set size). The simulator can also include emulation code for target machine instructions that do not exist on the host machine. For example, in the MP Mable simulator (see Section 3) the target machine is based on a MIPS-III architecture whereas the host machine is a MIPS-I based architecture. The added MIPS-III instructions, load double for instance, are simulated with a sequence of host MIPS-I instructions.

The Mable emulator shares the address space with the target application. This has the advantage of allowing references to the application's address space without the need to do address translation on every instruction fetch and data load or store. However, the presence of the emulator perturbs the memory access patterns by preventing the application from occupying its original location in the address space. This perturbation is minimal and has not caused a problem in our experience.

2.3 System Calls

Unlike many other instruction emulation methods, Mable provides a simple mechanism to pass system calls through to the underlying operating system. When the simulator detects a SYSCALL instruction in the application code the emulation routine simply copies the system call arguments from the target machine's registers into the appropriate hardware registers of the host system. Once this is done, the emulation routine executes an actual SYSCALL instruction. This makes the system call appear to pass directly through to the host machine's OS, and in this way the OS treats the request as it would for any normally executing program running on the host machine. Using this technique facilitates easy access to the underlying services of the host machine. The host machine I/O capability is accessed in this manner and requires no special coding on the part of the application writer.

The simulator also provides support for system calls that are not present on the host machine. If such a system call is requested by the application, the simulator merely branches to a routine which implements the new call. This feature is used in the implementation of MP Mable (see Section 3.2).

2.4 Debugging capability

One of the key advantages of the Mable framework is the ability to reuse existing software development tools. One particularly important aspect is providing support for debugging. The Mable framework allows both the simulator and the emulated application to be run under dbx, the standard UNIX debugger. Furthermore, when debugging the simulated application, the state of the target machine is loaded in at the appropriate times so that from the user's perspective it seems as if the debugging is taking place on the target machine. In order to hide some of the complexity of interacting with the simulator, a special `.dbxinit` file is used to map some complex sequences of dbx commands into simple macro commands.

Almost all of the debugger commands can be used on an emulated program without making any modifications to dbx. Consider for example setting a breakpoint at a particular source line in the emulated program. The programmer uses the usual command to select a source line. The debugger then follows the normal procedure to set the breakpoint: it uses the emulated program's symbol table to determine the instruction to stop at and temporarily replaces the instruction at that location with a *break* instruction. When the breakpoint location is reached by the emulator and the *break* instruction is simulated, the emulator executes a real *break* instruction. The debugger now regains control and replaces the removed instruction before execution is continued. Dbx is capable of placing breakpoints either in the simulator or in the simulated code in this fashion.

In order to make the target machine's state visible to the user, the Mable emulation code for *break* moves the simulator environment out of the host machine's registers and into a save area. The emulated program's state is then loaded into the host registers, in effect mapping the emulated registers into their hardware analogues. If the host machine and the emulated machine do not have identical register sets then a mapping must be defined. For instance, if the host machine is a 32-bit machine and the emulated machine is a 64-bit machine then each emulated register may be mapped to a pair of host registers. If the emulated machine has more registers than the host then dbx macros may be defined to map in one part of the emulated register set at a time. This context switching can only be partially completed by the simulator[1], so the remainder of the task is done by a dbx command macro (also defined in `.dbxinit`). Once this is complete, dbx is capable of examining the simulated program as usual (e.g., stack backtraces, displaying variables, etc.). To continue from a break point, a specially defined macro command, *resume*, is used in place of the normal dbx command, *continue*. The *resume* command causes the emulated program's state to be saved, the simulator state to be restored, and emulation to continue.

Dbx can be used even if the emulated program contains instructions which do not exist on the host processor. While the debugger may not be able to properly decode and print one of the new instructions, it can set a breakpoint, single-step, etc. The real constraint on using the debugger is that the call stack conventions for the host and target machines should be identical in order for dbx to be able to print the call stack and the values of local variables.

2.5 Summary

The techniques described above constitute the blueprint for constructing a virtual machine simulator. We have shown how host operating system services and existing development tools such as debuggers can be reused. The result is a simulation system which can be developed rapidly, provides good performance, and allows code tuning and debugging with familiar tools. We now describe how this simulation framework can be applied to three practical situations. The examples we provide illustrate some of the implementation details of the concepts we have just described, and also show how this simulation concept can be applied to a wide variety of target architectures.

[1] The program counter, for instance, cannot be modified by Mable.

3 Multiprocessor Mable

Multiprocessor Mable, or MP Mable, is a Mable-based simulator for a proposed shared memory multiprocessor. Each of the processors uses the MIPS-III instruction set which is a superset of the MIPS-I instruction set of the host machine on which the simulator runs. [Kane87] One of the goals of MP Mable is to simulate a realistic ordering of load and store instructions between processors. Thus it is important to model each processor's timing accurately including both primary and secondary cache behavior. A second goal is to allow the use of as many of the existing MIPS programming tools as possible for modifying and analyzing the emulated program, including the compilers, debugger and profiling programs. In fact, since existing compilers on the host machine can produce code for the emulated machine's instruction set no changes to the compiler system are necessary. Users of the simulator can therefore use familiar tools, yet the programming environment does not have to be recreated.

3.1 Simulator Implementation

The core of the MP Mable simulator is the instruction emulation loop as described in Section 2. The major additions to the core instruction-emulation loop are a process scheduler to manage context switching and a memory model to accurately simulate delays in the memory system.

Each simulated processor's context consists of the contents of its register files and its cache tags (but not the cache data), and some status information. This state is stored in the host machine's memory. Each processor's status information includes a cycle count or local time that is advanced as instructions are simulated. The simulator emulates one processor at a time, and the instructions from the active processor are simulated until a load or store instruction is fetched. Prior to actually simulating a load or store instruction, i.e. accessing state that is visible to other processor contexts, MP Mable checks to see if the active processor's local time has now become more advanced than the oldest inactive processor. If indeed it has, then a context switch is performed.

The scheduling algorithm described so far causes a context switch only at load or store instructions which is adequate for guaranteeing correct execution of the program. However, to correctly model delays caused by contention for shared memory resources we need to include a detailed model of the memory system and force context switches at appropriate points in the cache miss-handling sequence. If a cache miss is detected during the simulation of a load or store instruction then the simulator calls detailed miss routines coded in C. These routines accurately model traffic on the shared bus and associated interface and can trigger context switches at all critical points within the cache miss sequence.

3.2 Parallel Programming Environment

MP Mable provides a library of support functions to facilitate the coding of parallel applications. These routines include newly-added system calls to create a new process, return a processor's local time, enable or disable statistics gathering, declare a variety of coherency attributes over a range of user memory[2], and so on. Routines are also provided

[2]The MIPS R4000 architecture supports different coherency attributes on a per-page basis.

for parallel programming constructs such as barriers, atomic counters and locks, etc. These routines make it easier for application writers to port programs to the MP Mable system.

Debugging support is provided as described in Section 2. The standard dbx debugger can be used to look at the state of the currently-executing processor or any other processor's context through a set of macros which load a particular context into the host machine's registers. Full source-level debugging is available.

3.3 Profiling and Statistics gathering

MP Mable provides two independent statistics gathering tools. The first mechanism is entirely analogous to the uniprocessor basic-block profiling tools *pixie* and *pixstats* as provided by MIPS. [Smith91] *Pixie* works by instrumenting the application code in order to count basic blocks at execution time. The uniprocessor version of *pixie* cannot be used because it cannot independently instrument each process of a multiprocessor application, so instead MP Mable's simulation loop keeps a tally of individual instruction execution counts for each context. At the completion of the simulation the instruction counts are consolidated on a basic block basis and processed to produce a basic-block count file in the same format that *pixie* generates. A count file is generated for each processor in the MP configuration. The resulting files can then be analyzed with the standard analysis programs, *pixstats* and *prof*, and reports can be produced that provide insight into the behavior of each processor during the execution of the application. The gathering of basic-block count information is an option which slows the simulation rate.

The second technique collects statistics associated with the behavior of the emulated hardware, i.e. the caches and underlying processor interconnections. Counts of cache misses and various types of bus transactions are kept. User-accessible calls are provided in the MP library to allow a user to enable and disable statistics gathering at specific places within the application and to print out the statistics.

3.4 Performance

MP Mable achieves very high simulation speeds because the instruction decoding and dispatching loop is very tightly coded and many of the target-machine instructions can be emulated with a single host-machine instruction. Emulation of simple register-register instructions requires 23 host instructions/target instruction. Emulation of non-native instructions and context switching between virtual processors add additional overhead. However, the time required to emulate a complete application is dominated by the memory simulation time. With a complete memory model simulating cache and bus behavior, the average emulation rate is 167 host instructions/target instruction. This results in a simulation rate of greater than 120,000 target instructions/second on a 25Mhz MIPS R3000-based host system.

4 Tsim

Tsim is an instruction-level simulator for TORCH [Smith90, Smith92], a statically-scheduled superscalar processor that supports speculative execution. Tsim is based on the Mable framework, but in contrast to the MP Mable simulator described in the previous section the instruction-set architecture which Tsim emulates is significantly different from the instruction-set architecture of the host processor (a MIPS R3000). This section illustrates some of the types of architectural features which the Mable simulation framework can handle. The goals of the simulator were to provide a platform to debug the TORCH compiler and to measure the performance of the compiler-hardware system. Tsim not only filled these requirements but also allowed the emulated program to work with the underlying operating-system services and the standard debugger, minimizing development time of the simulator and programming environment.

4.1 TORCH Architecture

TORCH implements the MIPS-I instruction set but differs in a number of ways from a conventional RISC processor. First of all, TORCH is a dual-issue statically-scheduled machine, so two instructions are fetched and executed every cycle. Second, the TORCH processor includes hardware to handle speculative, or "boosted," instructions. A boosted instruction is an instruction which has been moved prior to a branch on which it depends. The TORCH compiler does all instruction scheduling, including choosing which instructions to boost, and predicts whether or not each branch will be taken. The architecture specifies that a boosted instruction takes effect only if the next branch follows the path predicted by the compiler and must be squashed otherwise. The hardware implements this feature by storing the result of a boosted instruction in a shadow register file until the resolution of the next branch, and boosted memory stores are saved in a shadow store buffer. If a branch is correctly predicted then all boosted results are transferred to the real machine state; otherwise the results are ignored. Subsequent boosted instructions can read the values in the shadow structures before the next branch.

A third major difference between TORCH and the base MIPS-I instruction set is the instruction encoding. Extra bits are needed to indicate which source and destination registers are boosted registers and to encode branch predictions. Rather than recoding the entire MIPS instruction set the TORCH designers chose to use 40-bit instructions. Each instruction is a standard MIPS instruction with an "extension" byte of TORCH-specific information appended to it. While the internal instruction cache stores the complete 40-bit instructions, in the external memory the extension bytes for a group of eight instructions are placed together in two words before the MIPS portions of the instructions.

Finally, TORCH supports densely-coded NOPs. If the compiler cannot find two instructions that can be executed concurrently it would normally need to fill an unused slot with an explicit NOP instruction. To reduce the number of explicit NOPs, TORCH allows NOPs to be encoded using a single bit in the TORCH extension byte. An instruction "packet" consisting of two 40-bit instructions can specify up to two NOPs and two normal instructions.

In order to compile programs for the TORCH architecture we had to write a new instruction scheduler and an assembler. However because the TORCH instruction set is similar to the MIPS-I instruction set we were able to use an existing C compiler to produce assembly language. The instruction scheduler then takes advantage of TORCH-specific features during the scheduling process. We were also able to use the standard link editor. Thus we did not have to recreate the entire compiler system.

4.2 Simulator Changes

The starting point for Tsim, the TORCH simulator, was a uniprocessor version of Mable which emulated a MIPS R3000 processor with a perfect memory system (no cache simulation). The core of the simulator consisted of a tightly-coded assembly-language loop just as for MP Mable. The changes required to build Tsim fell into the four areas outlined above: support for superscalar issue, support for boosting, support for 40-bit instructions, and support for densely-coded NOPs.

The changes to support superscalar issue were trivial because TORCH code is statically scheduled. The only difference from the single-issue simulator is that when two instructions are issued together the operands of both instructions must be read before either instruction executes and writes its results into the register file. A dynamically-scheduled processor could be supported although the emulation might be significantly slower, depending on the complexity of the code to determine instruction dependencies.

To implement boosting we added a small number of new data structures which correspond to the hardware structures in a real TORCH implementation: arrays representing the integer and floating point shadow registers and the boosted store buffer. We then made minor changes to the register read and writeback code to access the shadow register files when appropriate, and changes to the emulation of store instructions to write results of boosted stores into the boosted store buffer. We added a test to the branch emulation code to determine the accuracy of the branch prediction, and a second test in the instruction dispatch code to determine when a commit point is reached (after the delay slot of a branch). If a commit point is reached and boosted instructions have been executed then the emulator either commits or squashes the state stored in the shadow registers and the boosted store buffer.

One more consequence of boosting must be handled: a boosted instruction which excepts must not cause a real exception until the next commit point, and the exception must never occur if the instruction is squashed. This exception model is handled by installing signal handlers in the simulator which trap all possible signals caused by exceptions. During emulation of a target instruction which would cause an exception Tsim eventually executes an instruction in the simulation sequence which causes a real exception. The signal handlers catch the exception and determine if the offending emulated instruction is boosted. If so, the exception is noted and the simulator resumes execution normally. At the next commit point, depending on whether the last branch was correctly predicted or not, the exception is either discarded or a real exception is generated.

The code to fetch instructions is complicated by the fact that TORCH uses 40-bit instructions. The TORCH PC is actually an instruction number, not a virtual address, so the emulator must translate the PC into the corresponding virtual address in order to fetch an instruction.

Finally, densely-coded NOPs introduce a substantial amount of complexity into the emulator instruction dispatch code which decides which instructions to execute and in what order. There are many special cases depending, for instance, on whether or not the instruction is in a delay slot. However, the code to implement these cases is localized to one small part of the program.

The TORCH architecture is substantially more complicated than an R3000, yet it was possible to add the features described above to the base simulator with fairly minimal effort. The new simulator was up and running after about two person-weeks. Furthermore, we were still able to use the standard MIPS debugger as described in the next subsection.

4.3 Debugging Support Changes

Very few additional modifications were necessary to get the standard debugger to work with a TORCH program being emulated by Tsim. The TORCH compiler system generates a normal MIPS object file with a full symbol table. Only the text segment is special since it contains 40-bit TORCH instructions. However, since all of the extensions in the TORCH instruction set are encoded in one byte and the remaining four bytes are identical to a MIPS instruction one can still use dbx if the PC is adjusted to point to the MIPS part of the instruction. The only adjustments required were some simple changes to the macros in the .dbxinit script which roll in the emulated context. The TORCH assembler produces a special symbol table in which the addresses associated with text labels are virtual addresses pointing to the MIPS part of the appropriate instruction, so no changes were necessary to dbx's breakpoint mechanism. Almost all of the normal dbx commands worked as usual, including stack traces, source-line and instruction-level breakpoints, and printing of local and global variables.

We also added macros to swap the main and shadow register sets to allow the programmer to look at the shadow register contents using the normal dbx commands. We did have to introduce some special-case code in the Tsim emulator to support single-stepping at the instruction level since dbx could not properly determine the location of the next 40-bit instruction.

The debugger was invaluable for finding errors in the code produced by our compiler system. In principle our system can provide full source-level debugging without making any modifications to dbx. However, there are some serious limitations due to the fact that the TORCH instruction scheduler produces heavily-optimized code which makes it impossible to present a source-line-by-source-line execution model to the programmer. Such problems exist even for a conventional single-issue processor when the compiler's optimizer is turned on, but are even more of a problem for TORCH because many of the features of the architecture cannot be tested without using the optimizing scheduler.

4.4 Statistics Gathering

Tsim includes code to count the number of useful instructions executed (boosted instructions which commit and non-boosted instructions) and the number of simulated machine cycles. Since the primary purposes of the simulator were to test the TORCH compiler for correctness and to measure useful-instruction counts we were less concerned with collecting detailed statistics. Additional statistics collection code and cache models could easily be added just as for the MP Mable simulator.

4.5 Performance

Recall from Section 3 that the base Mable emulation time for a single register-register instruction is 23 host instructions/emulated instruction. The simulator changes to support fetching 40-bit instructions, dual-issue, and dynamic NOPs are all on the critical path of the simulator, in addition to some of the checks for boosted registers and commit points. Other additions to the simulator, such as code to manage the boosted store buffer and to copy shadow values at a commit point, are executed far less frequently and have a lower impact on performance. Note that Tsim does not include a cache model and therefore does not have any overhead for memory simulation. Tsim achieves an average emulation rate of 80 host instructions/target instruction, which gives a simulation rate of about 300,000 target instructions/second on a 25Mhz R3000 workstation.

5 PPsim

The third application of the Mable framework is a simulator called PPsim which was built to support the design of FLASH, a scalable shared memory multiprocessor. Each node of the FLASH machine includes a memory and communications controller chip which contains an embedded processor called the Protocol Processor (PP). Software written for the PP implements the protocols needed to maintain cache coherence and provide user-level message passing. The PP instruction set has been optimized for performing the operations common in communication protocol code. To allow the machine designers to easily debug and optimize the protocol code, they needed an efficient simulator for the PP that would support source-level debugging.

5.1 PP Architecture

Since the PP is an embedded processor, its architecture reflects the special-purpose nature of the functions it performs frequently. The PP is a RISC machine which is based on a modified MIPS-I instruction set architecture. Protocol operations do not require all of the features of a general-purpose processor, so the PP hardware has been simplified by removing support for floating point operations, integer multiply and divide, interlocks, hardware interrupts, exceptions, and address translation. On the other hand, since the PP needs to perform protocol state manipulations rapidly the designers added additional support for bit manipulation and expanded the data path to 64 bits. Finally, for increased instruction throughput, the PP is implemented as a dual-issue statically-scheduled superscalar processor. We are building a custom compiler system for the PP architecture.

5.2 PPsim Implementation

Unlike the simulation loops in MP Mable and Tsim, which were coded in assembly language to maximize performance, we chose to implement this version of the simulator in C. Part of our motivation was to explore the ability of a C-language implementation to compete with the performance of the assembly language versions. In addition, we also wanted to reduce development time and make the simulator more portable and maintainable. Since the instruction set of the PP differs more from the host (MIPS) architecture than was the case with MP Mable or TORCH, implementing the simulator in a high level language is more appropriate than in those cases. An additional source of complexity is the fact that PPsim will ultimately form part of a multiprocessor simulator called FlashLite that also simulates the other hardware units in the FLASH memory and communications controller. Since some of the PP instructions cause actions in other hardware units, the PPsim emulation code for these instructions needs to change state variables in other parts of the FlashLite simulator. If PPsim were written in assembly language, interacting with the other simulator units would be complex and error-prone.

The fundamental structure of PPsim is the same as for MP Mable and Tsim. The core of the emulator is a fetch loop which selects an instruction pair and branches to emulation routines for each of the two component instructions. Just as MP Mable and Tsim use jump tables, PPsim utilizes an array of function pointers to select the emulation code for each kind of instruction. Also like Tsim, PPsim emulates a dual-issue machine and therefore the results of coupled instructions are committed only after both have been emulated. Though PPsim is written in C, we still use the same mechanism to allow the debugger to operate on the emulated code. Short assembly language routines were written to save the simulator state when the debugger is entered and restore it when emulation resumes.

Because we are using a high-level language, it is easy for us to port our simulator to different host architectures in the MIPS family. We currently run our simulations on a 32-bit host, and so 64-bit PP operations are simulated by sequences of 32-bit operations. However, we soon plan to port the simulator to a 64-bit MIPS-III-based host, which will allow the PP's data manipulations to be implemented more directly with 64-bit host machine instructions. This port will be simple because we have written the simulator using the 64-bit data type extensions defined by the GCC compiler [Stallman93]. As a result, porting to a new host is likely to just involve recompilation on the new system. Using C also reduced the development time: the core functionality of PPsim was coded, debugged, and optimized by one person in a week.

5.3 PPsim Performance

To optimize PPsim's performance we examined the assembly code produced by the compiler and made some improvements by changing the C source code, but did not perform all the possible optimizations. We noticed in particular that the performance of the simulator depends strongly on the register allocation. This code was compiled with g++ 2.4.5 under O2. The average simulation speed is 177 host instructions/emulated instruction on a MIPS-I based host, not including a cache model. On our 25 Mhz R3000 host machine this translates to about 132000 instructions/sec. We believe that the major reason for the reduced performance as compared to other Mable versions is the fact that all PP instructions operate on 64-bit quantities and must be emulated by sequences of 32-bit instructions on the host machine. We expect the performance of PPsim running on a 64-bit host to be more comparable to the performance of the assembly-language simulators. We feel that the implementing Mable in C does not impose a severe performance penalty, while it makes implementation and maintenance much easier.

6 Related Work

Mable is a framework for creating an instruction level simulator that can be used to both evaluate the performance of a proposed machine and to debug and tune the software running on the machine. Many tools have been proposed that implement some or all of these functions. These tools fall into three main categories: instruction emulation, trace based simulation, and code annotation and translation. This section compares Mable to these alternative methods.

6.1 Instruction Emulation

Direct instruction emulation is the most straightforward of the simulation options, and the most flexible. It can provide the most comprehensive model of the machine and becomes the natural choice when the goals include development and study of operating systems, low-level device drivers or hardware interfaces. The technique employs a fetch-and-execute simulation loop with a jump table to access sequences of host machine code that emulate target machine instructions. Instruction level simulators generally model all major components of the target machine. If the simulator goal is to boot a "real" operating system then a complete target machine model must be available, including virtual address translation mechanisms and a memory subsystem. Each detail of the target system is modelled at a cost to the overall simulator performance.

An example of this type of system is Sable, an instruction emulator for the MIPS processor family. [MIPS] The uniprocessor Sable simulator has a performance of 1400 host instructions/emulated instruction. At the other end of the performance spectrum for instruction emulators is a simulator for the Motorola 88000 based on threaded code [Bedichek90, Bell73]. This simulator has no cache model and uses a very efficient instruction-dispatch mechanism. The target program is first translated into a decoded form in which each instruction contains a pointer to the simulator code which executes that particular instruction, as well as pointers to the sources and destination of the instruction. When an instruction is fetched the simulator jumps to the address indicated in the predecoded instruction. This scheme reduces the overhead of decoding each emulated instruction at run time. The reported simulation speed is 20 host instructions/emulated instruction for a uniprocessor simulation without a cache timing model but with detailed behavioral models for hardware devices and a virtual memory translation simulator.

Mable is a member of this family of simulators but with significant trade-offs made in the interests of simulation speed and development time. Mable does not model a virtual memory translation scheme but instead relies on the host machine to provide this function. This is accomplished by having the Mable emulator share the address space with the target application. This has the advantage of allowing references to the application's address space without the need to do address translation on every instruction fetch and load/store of data, resulting in a faster model than most other instruction emulators. However this technique has the disadvantage that the presence of the emulator perturbs the memory image preventing the application from occupying its original space. In addition, Mable-based simulators use the underlying OS to provide system services and are compatible with existing program development tools such as the debugger, shortening simulator development time. In contrast, Bedichek's simulator includes a 50,000 line C-language module to implement a debugger, which took six person-months to develop. Unlike Bedichek's simulator, Mable cannot be used to develop device drivers and other low-level operating system code.

6.2 Trace Based Analysis

A popular method for estimating the performance of a proposed architecture is to use a trace-based simulator. Such simulators use instruction and data traces from an existing machine as input to a timing simulator which determines how long the same program would take on the target machine. The traces can be efficiently created using code annotation techniques described in the next section.

Trace-based simulators are used frequently for memory-system performance analysis [Smith87, Eggers89, Chen92]. Since the data reference stream is not strongly dependent on the instruction-set architecture it is reasonable to use a trace from an existing processor. A second common application is the analysis of a superscalar processor with the same instruction-set architecture as the machine on which the trace was generated [Smith89, Butler92]. This approach is relatively simple to implement but has the drawback that the traces do not depend on the target machine. As a result performance estimates can be inaccurate, for instance when different orderings of the memory references change the behavior of the caches. Furthermore, it is difficult to tune the applications since they never run on the target machine.

The speed of this approach depends on the level of detail that is being simulated. In theory trace-driven simulation should be faster than a Mable-based simulator because Mable simulates the timing and the results of every instruction, while a trace-based simulator typically only simulates the timing of memory references. However, if the overhead of memory simulation dominates then the difference is negligible. For example in MP Mable, the simulation of the memory system required on average more than four times the amount of time of the base simulation loop.

6.3 Code Annotation and Translation

The third class of machine simulators is based on transforming the target application into an instrumented program which runs directly on the host processor. There are two techniques in this class, *code annotation* and *code translation*. Code annotators assume that the target application can run directly on the host machine, i.e. the target instruction set architecture is identical to the host instruction set architecture. The application code is augmented by additional instructions which gather execution statistics or call a memory system simulator. Examples of this technique include *pixie* for MIPS systems [Smith 90], developed by Earl Killian, and the *goblin* system for the IBM RISC System/6000, developed at Carnegie Mellon University [SCH+91]. These tools insert additional instructions in each basic block to collect basic-block counts and memory reference counts, or to produce instruction or data traces. Code annotation is used primarily to experiment with changes to the memory hierarchy, to measure performance statistics, or to generate traces, but it cannot be used to experiment with extensive changes to the instruction set architecture. This technique provides very high performance with overhead as low as 2-3X, although the overhead is largely dependent on the detail of the gathered information.

Tango [DGH91], Proteus [Brewer92], and Tango Lite [Golds92] are code annotation systems for efficiently simulating the behavior of a multiprocessor memory system. These systems augment the memory references in a program with calls to a user-supplied memory system simulator. All instructions other than memory references are executed directly on the host processor, yielding efficient program execution. However, as with trace-based simulators, the cost of memory-system simulation often dominates overall run-time of the simulator and thus the performance of Mable-based simulators can still be competitive. Tango provides support for the simulation of parallel programs by using multiple UNIX processes while Proteus and Tango Lite use light-weight processes running in a single address space, but none of these systems allow for changes to the instruction set architecture. It is possible to use an existing uniprocessor debugger with the annotated program, but it can be confusing since the target program has been modified by the code annotation system.

The second direct-execution technique, *code translation*, is more general. In this technique all of the instructions of the target application are translated into the host instruction set, either before or during execution. Thus, target instructions which are not directly supported by the host machine are translated into an equivalent sequence of host instructions. One example of this technique is the binary translation system designed by Digital Equipment Corporation in order to execute VAX and MIPS binaries on their Alpha processor [S+93]. The first component of this tool translates the original code to the Alpha instruction set using a one-to-many translation. The translated code is executed directly in most cases, but runtime emulation is used if the translator cannot statically determine a translation for a particular section of code. The translated code actually runs faster on Alpha than on the original (slower) processors. However, unlike Mable, the translator is used only to simulate the code's effects.

An application of code translation to architectural simulation is the Shade system, developed at Sun Labs and the University of Washington [CK93]. This tool utilizes dynamic code translation to convert the target code to instructions on the host machine. These translations are cached so that compilation overhead is amortized over the execution of the program, and recompilation is only performed when necessary. In addition to Shade's ability to provide efficient execution of a non-native instruction set, it provides the ability to trace and analyze program behavior. Shade allows the experimenter to customize the level of detail collected, even dynamically during program execution, so that only the desired statistics are gathered. As a result, the Shade system is able to achieve performance commensurate to the required level of detail by only invoking simulation code when necessary. Shade shares with Mable the goals of efficient architectural simulation and experimentation, but does not provide the same support for program development. It is difficult for Shade to take advantage of existing program debugging and profiling tools, though their tool includes the ability to do custom profiling.

7 Conclusions

Mable is an efficient framework for simulation of proposed machine architectures. It takes advantage of similarities between the software conventions of the host machine and the proposed machine to allow the simulator developer to create a software development environment for the new machine by leveraging off of the host's software development tools. Thus Mable is a simulation system that provides support for software debugging and tuning, as well as architectural performance tuning of the base machine. The key idea is to present the operating system and the software tools with a program which appears to behave like a normal host application while convincing the application that it is running on the simulated machine. This illusion can be accomplished by replacing the standard UNIX prologue with an efficient instruction emulator and by mapping the application's machine state to the host machine before system services are called.

The Mable-based simulators provide support for application and compiler development with faster emulation and shorter simulator development time than traditional instruction emulation systems. While Mable is not a simulator toolkit, it is relatively easy to build simulators for new architectures because of the extensive reuse of underlying host operating system services. Furthermore, the Mable concept is flexible enough to support a variety of architectural simulation problems. We have shown how the Mable framework has been used to implement simulators for three very different architectures on the same host machine.

8 Acknowledgments

We would like to acknowledge Earl Killian of Quantum Effect Design for suggesting the ideas behind Mable. We also thank Helen Davis, Margaret Martonosi, Dave Ofelt and Rich Simoni for reviewing drafts of this paper, and the

TORCH and FLASH teams. This research was supported by ARPA under contract N00039-91-C-0138. John Heinlein was supported by an Air Force Laboratory Graduate Fellowship (AFOSR).

9 Bibliography

[Bedichek90] Robert Bedichek. Some Efficient Architecture Simulation Techniques. In *Proceedings of the Winter 1990 USENIX Conference*, Jan. 1990, p. 53-63.

[Bell73] James Bell. Threaded Code. *Communications of the Association for Computing Machinery.* June 1973.

[Brewer92] Eric A. Brewer, et al. PROTEUS: A High-Performance Parallel-Architecture Simulator. In *Proceedings of ACM Sigmetrics and Performance '92*, Newport, RI, June 1992.

[Butler92] Butler, M., et al. An Investigation of the Performance of Various Dynamic Scheduling Techniques. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, Portland, OR, December 1992.

[Chen92] Chen, J.B., et al. A Simulation Based Study of TLB Performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture,* Gold Coast, Qld., Australia, May 1992, pp. 114-23.

[CK93] Robert F. Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. University of Washington Technical Report, UWCSE 93-06-06, June 1993.

[DGH91] Helen Davis, Steve R. Goldschmidt, and John Hennessy. Multiprocessor Simulation and Tracing Using Tango. In *Proceedings of the 1991 International Conference on Parallel Processing (ICPP '91)*, August 1991, pp. II 99-107.

[Eggers89] Susan J. Eggers, et al. Techniques for the Trace-Driven Simulation of Cache Performance. In *1989 Winter Simulation Conference Proceedings*, Washington, DC, Dec. 1989, pp. 1042-6.

[Golds92] Stephen Goldschmidt. The Accuracy of Trace-Driven Simulations of Multiprocessors. Ph.D. Thesis, CSL-TR-92-546, Stanford University, September 1992.

[Kane87] Gerry Kane. *MIPS RISC Architecture.* Prentice-Hall: Englewood Cliffs, NJ, 1987.

[MIPS] Mips Technologies Inc. System Programmer's Package.

[S+93] Richard L. Sites, et al. Binary Translation. *Communications of the Association for Computing Machinery*, 36(2), February 1993, pp. 69-81.

[Smith87] Smith, A.J. Line (Block) Size Choice for CPU Cache Memories. *IEEE Transactions on Computers,* C-36(9), September 1987, pp. 1063-75.

[Smith89] Michael D. Smith, Mike Johnson, and Mark Horowitz. Limits on multiple instruction issue. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, 1989, pp. 290-302.

[Smith90] Michael D. Smith. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, WA, May 1990, pp 344-54.

[Smith91] Michael D. Smith. Tracing with Pixie. Stanford University Technical Report, CSL-TR-91-497, November 1991.

[Smith92] Michael D. Smith. Efficient superscalar performance through boosting. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October, 1992, p. 248-59.

[Stallman93] Richard Stallman. *Using and porting GNU CC*. Free Software Foundation, Cambridge, MA, June 1993.

[SCH+91] Chriss Stephens, Bryce Cogswell, John Heinlein, Gregory Palmer and John P. Shen. Instruction Level Profiling and Evaluation of the IBM RS/6000. In *Proceedings of the 18th International Symposium on Computer Architecture*, Toronto, Canada, May, 1991, pp 180-9.