A Dynamic Binary Translation Approach to Architectural Simulation

Harold W. Cain, Kevin M. Lepak, and Mikko H. Lipasti

Department of Computer Sciences
Department of Electrical and Computer Engineering
University of Wisconsin
Madison, WI 53706

(608) 265-2639
(608) 262-1267 (FAX)

{cain}@cs.wisc.edu, {lepak,mikko}@ece.wisc.edu

Abstract

We present the design of a PowerPC-based simulation infrastructure for architectural research. Our infrastructure uses an execution-driven out-of-order processor timing simulator from the SimpleScalar tool set. While porting SimpleScalar to the PowerPC architecture, we would like to remain compatible with other versions of SimpleScalar. We accomplish this by performing dynamic binary translation of the PowerPC instruction set architecture to the SimpleScalar instruction set architecture, and by mapping the PowerPC architectural state onto the SimpleScalar register set. Using this infrastructure, we execute unmodified PowerPC binaries on an out-of-order processor timing simulator which implements the SimpleScalar architecture. We describe and investigate trade-offs in the translation of some complex PowerPC instructions and advocate adoption of *speculative decode* to optimize instruction translations for the common case. We find that simple decode predictors can reach better than 90% accuracy for guiding speculative decode.

1 .0 Introduction

Simulation has gained widespread use in the computer systems research community as a method of studying the behavior of complex computer systems. Simulation can be used not only to study pre-existing computer systems, but also proposed architectures, a welcome alternative to physically building hardware or creating complex analytical models. Recent advances in hardware performance have enabled the use of increasingly detailed architectural simulators, modeling the attributes of computer systems in such detail that they can boot a commercial operating system [6].

We are currently developing a simulation infrastructure for performing computer architecture research, and are basing this infrastructure on the SimpleScalar architectural toolset [1]. The SimpleScalar simulators are execution-driven simulators which run programs compiled for the Portable Instruction Set Architecture (PISA), a MIPS derivative instruction set. We have modified SimpleScalar's most detailed processor simulator, `sim-outorder`, so

that it can execute unmodified PowerPC binary executables. This toolset has become quite popular in the research community as a platform for computer architecture research. Consequently, there are many different versions in existence, modeling alternative processor and memory system organizations. In the future, we would like to leverage this significant body of work by possibly incorporating other versions of SimpleScalar into our own research infrastructure. In order to ensure compatibility between our version of SimpleScalar and other versions, we must port SimpleScalar to the PowerPC architecture while minimally changing its source code.

We have chosen to perform dynamic binary translation from the PowerPC architecture to the SimpleScalar architecture as a means of executing PowerPC applications while minimally changing the SimpleScalar tool set. We are interested in the PowerPC instruction set because we have access to a PowerPC full-system simulator that is capable of running complex commercial workloads, including all of the AIX operating system as well as code generated dynamically by just-in-time compilation frameworks for Java [6]. Performing this translation is analogous to the kind of translation already performed in hardware by many modern microprocessors[2, 9], which transform individual instructions into a series of less complex operations. Traditionally reserved for CISC instruction sets, this kind of instruction cracking also occurs in recent implementations of the somewhat complicated PowerPC architecture[3, 4]. Implementing this cracking in SimpleScalar involves adding a translation pipeline stage between the processor fetch and decode pipeline stages, in which incoming PowerPC instructions are decoded and translated into the SimpleScalar instruction set, which are then fed to the normal SimpleScalar PISA decode stage.

An alternative method of porting the SimpleScalar simulators to the PowerPC architecture is to directly modify the SimpleScalar source code, rather than performing this translation. The SimpleScalar simulators were designed with porting to other architectures in mind; in fact there is already a distributed version of SimpleScalar for the Alpha platform, and we are aware of SPARC and PowerPC versions currently in development elsewhere. While performing the necessary modifications is certainly feasible, there are many complications which prevent us from adapting SimpleScalar to the PowerPC architecture as cleanly as we would like. For instance, the `sim-outorder` implementation contains a machine definition file which describes the semantics of each instruction in a given instruction set. The format of this machine definition allows for instructions which change the contents of at most two registers. However, the PowerPC architecture contains many instructions whose execution updates more than two registers. Rewriting `sim-outorder` to support additional target registers would be non-trivial, and would also contradict our goal to remain compatible with other versions of SimpleScalar. Many small issues such as this make a direct implementation of PowerPC in SimpleScalar undesirable.

This paper focuses on some of the issues that we have dealt with in implementing dynamic binary translation. We begin with a brief overview of the SimpleScalar and PowerPC architectures, for those unfamiliar with the two. This overview is followed by a description of how we maintain all of the necessary PowerPC architectural state in the SimpleScalar architecture. In Section 4, we present a detailed description of our translation mechanism, including examples of a few PowerPC instructions for which performing translation is problematic. Section 5 discusses the efficiency of our translator for a few benchmarks, and we conclude with a summary of future plans for extending this simulation infrastructure.

2 .0 Architecture Overview

2.1 The SimpleScalar Architecture

Although the Portable Instruction Set Architecture (PISA) implemented in SimpleScalar is a derivative of the MIPS instruction set architecture, there are a few noticeable differences:

- Instructions use a 64-bit encoding.
- There are no architected delay slots: the instructions following control transfer instructions are not exe-

cuted prior to the control transfer.

- The results of comparison operations are stored in 32-bit general purpose registers
- Two additional addressing modes are provided for loads and stores: indexed (register+register) and auto-increment/decrement.
- All data and instruction memory accesses must be aligned.
- There is no equivalent of “supervisor” state--SimpleScalar proxies system calls to the host.
- There is no system exception handler.
- The endian mode is inherited from the host running the simulator.
- All register-writing instructions modify at most two registers.

2.2 The PowerPC Architecture

Our current simulator implements a subset of the 32-bit specification of the PowerPC architecture [8]. PowerPC is a relatively complicated load-store RISC architecture (as compared with simpler RISC ISAs such as Alpha or MIPS). Relevant features of the architecture when comparing to SimpleScalar include:

- Instructions use a 32-bit encoding.
- The results of comparisons are stored in one of eight fields of a 32-bit condition register.
- Instruction memory accesses must be 32-bit aligned.
- Data memory accesses need not be naturally aligned.
- Bi-endian support is defined in the architecture, with preferred mode being big-endian.
- Supervisor state is defined, as well as various exception handling mechanisms.
- Some instructions may write up to 32 general purpose registers, many also write the condition register and exception register in addition to a single result operand.

Of these differences, the most pertinent to our translation efforts are the alignment restrictions of the PISA architecture, the PowerPC arithmetic instructions which set condition register fields and the exception register as side effects, and the lack of system level state and exception support. We present our handling of these operations in Section 4.

3 .0 PowerPC State Mapping

There are two major components of the PowerPC architectural state which must be emulated by the SimpleScalar architecture: the PowerPC register state and the PowerPC memory state, both of which are discussed in this section.

We map the register state of the PowerPC architecture onto the general-purpose registers of the SimpleScalar architecture. Although our translator does not currently support supervisor-mode instructions, we intend to add this support in the near future. In order to facilitate this work, we have defined a register-state mapping for not only user-level PowerPC registers but also system-level PowerPC registers. To achieve this mapping, we use 79 general purpose SimpleScalar registers and 64 SimpleScalar floating point registers which correspond to the full PowerPC register set (32 general purpose, 32 floating point, 35 control registers). Although we could have performed this mapping using the default number of SimpleScalar registers (32 general-purpose, 32 floating point), this solution would have necessitated frequent spilling of register values to memory, and would have contradicted our goal of modeling the internals of a PowerPC processor with reasonable accuracy. Increasing the size of SimpleScalar’s integer and floating point register set is trivial for up to 256 registers because the SimpleScalar instruction set encoding uses eight bit register specifiers, allowing us to implement this state mapping with minimal changes to the simulator.

From Table 1 we see that there is a one-to-one correspondence between each of the PowerPC general pur-

PowerPC Registers (32 or 64 bits)	SimpleScalar Registers (32 bits)
32 General Purpose Registers	32 General Purpose Registers
Condition Register	8 General Purpose Registers
Link Register	1 General Purpose Register
Count Register	1 General Purpose Register
Exception Register (XER)	4 General Purpose Registers
System Level Register Set (31 registers)	32 General Purpose Registers
Floating Point Status And Control Register	1 General Purpose Register
Floating Point Registers 0-31	64 Floating Point Registers

Table 1: Register State Mapping

pose registers and SimpleScalar registers. There is also a one-to-one correspondence for the PowerPC link register, count register, and floating point status register. We split each of the four bit condition fields in the 32-bit PowerPC condition register into its own SimpleScalar register. Each of these bit fields is accessed and modified independently by the PowerPC ISA, as if it were a single register. By splitting this 32-bit entity into eight registers, we prevent the creation of artificial dependences in the translated code, allowing an out-of-order processor model to execute in parallel instructions which read or modify separate fields of the condition register. We use a similar technique for the PowerPC exception register. In this case, we split the least significant three bits of the register into three separate registers, because each of these bits is often accessed independently by many different instructions. The remaining 29 bits of the exception register are included in a fourth SimpleScalar register. There is also a one-to-one mapping for each of the system level registers, with the exception of the 64-bit time base register. Because the SimpleScalar simulator uses 32-bit registers, the time base register must be implemented using two registers. PISA is a derivative of the MIPS instruction set, and its floating point registers follow the MIPS convention of having 32 single precision registers or 16 double precision registers, where each double precision floating point register occupies two single precision registers. Each of the 32 PowerPC floating point registers is double precision, so we must double the size of the SimpleScalar floating point register set to perform this mapping.

The mapping of PowerPC memory state to SimpleScalar memory state is relatively simple compared to the above register mapping. In the SimpleScalar simulator, all memory in the simulated system belongs to the PowerPC machine state; there is no SimpleScalar memory state. All PISA load and store instructions reference the PowerPC memory, and instruction fetches in the i-fetch pipeline stage also reference the PowerPC memory, fetching PowerPC instructions. Translated PowerPC instructions are stored in a buffer between the fetch and decode stage of the pipeline, so there are never any PISA instructions resident in the simulated machine's memory.

4 .0 Instruction Translation Mechanism

As mentioned in the introduction, we add a stage to the processor pipeline between the SimpleScalar fetch and decode stages, which performs the dynamic binary translation. An illustration of this pipeline can be seen in Figure 1. PowerPC instructions are fetched in the first pipeline stage, and these instructions are passed to the translate stage. The translate stage decodes the PowerPC instructions and converts them into a series of SimpleScalar instructions. When a PowerPC instruction is translated into more than one SimpleScalar instruction, we sometimes need to store temporary values between each SimpleScalar instruction. We cannot overwrite architectural state with these temporary values, so we use two additional general purpose and floating point registers for storing temporary results.

With the exception of translated branch instructions, all translated code is straight-line, containing no change in control flow. Branch instructions are translated into a series of SimpleScalar instructions, including one or more branches. All branches must jump to a new program counter value; there are no side-entrances to blocks of translated code. We make this requirement in order to simplify the bookkeeping of translated instructions. Translated instructions are placed into a translated instruction buffer, from which the normal SimpleScalar decode pipeline stage reads. The remaining portion of the pipeline behaves as usual, and has not required any modification on our part.

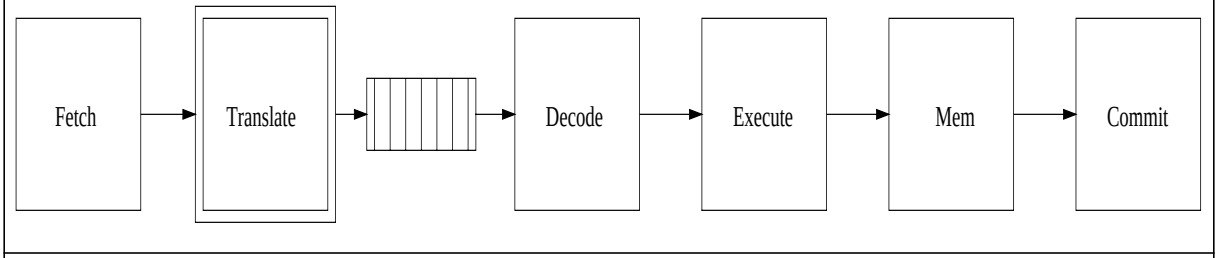

Figure 1. Modified processor pipeline including translation stage and translated instruction buffer

Many of the instructions in the PowerPC architecture have a clear one-to-one mapping to SimpleScalar instructions. In this section, we discuss a few of those instructions for which we do not have an obvious mapping, and the design decisions that we have made for these instructions. One luxury which we have had in the implementation of our dynamic binary translator, which has not been an option in many other binary translation systems (e.g. [10]), is that our target ISA is not necessarily a fixed target. We make a best effort at an efficient translation for all PowerPC instructions. However, if a critical PowerPC instruction is translated into a long series of SimpleScalar instructions, we have the option of changing the Simplescalar architecture to provide support for this PowerPC instruction. Also, there are some PowerPC instructions for which no mapping is possible given the pre-existing SimpleScalar ISA (e.g. the PowerPC *icbi* - instruction cache block invalidate). In cases like these, we have the luxury of a flexible target architecture. We use this approach for critical arithmetic instructions which conditionally set the XER register to indicate exceptional conditions such as overflow.

4.1 Branch Instructions

Control flow instructions are particularly interesting in translating from PowerPC to PISA because of the rich set of branching instructions provided in the PowerPC architecture which do not map directly onto SimpleScalar operations, as contrasted with ALU operations/etc. which tend to be fairly uniform across architectures. In keeping with the goal of needing the fewest possible modifications to the SimpleScalar simulator, we had to consider the translations of control instructions carefully so as to minimize changes to the fetch engine, branch prediction unit, and retirement unit. As an example of the potential complication, consider the PowerPC *bclr* (branch conditional to link register) instruction. In PowerPC, as implied by the mnemonic, a subroutine return jump can be conditional. In SimpleScalar, there are no conditional register direct jumps, only unconditional register direct jumps (e.g. *jr, jalr* instructions). Therefore, in order to translate *bclr*, we must test the condition (with a conditional branch) and also have the ability to perform a register direct jump in the translation of a single PowerPC instruction[1]. This scenario is illustrated in Figure 2.

Since we have chosen to make all PISA instructions corresponding to a given PowerPC instruction reside at

1. It is possible that if we could predicate certain instructions within SimpleScalar, this construct could be avoided. However, the default SimpleScalar PISA does not have this capability so it is not explored.

the same logical program counter (PC) value (to eliminate instruction address space translation issues), this poses a potential problem regarding fetching, branch prediction, and retirement. Specifically, consider branch prediction--now that two control flow instructions can exist at the same logical PC, our prediction hardware must handle the case where multiple branches exist at the same PC, and be able to either predict them correctly independent of one-another, or behave in such a way that the desired behavior for the combination of the two branches is a) correct and b) an accurate model of what might occur in a true out-of-order implementation of a PowerPC architecture (i.e. not suffering a misprediction for one of the two branches every time it is encountered). As shown in column two of Table 2,

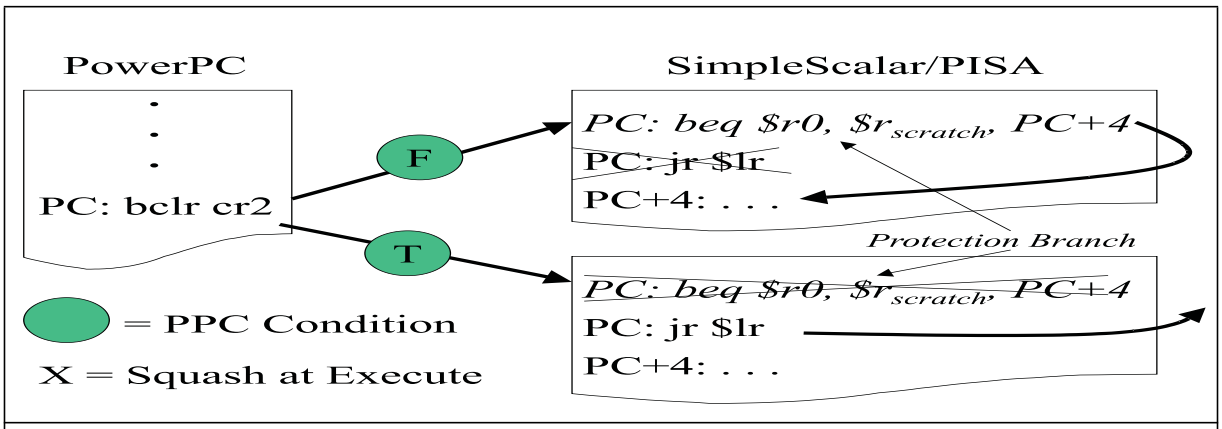

Figure 2. Illustration of PowerPC *bclr* instruction implementation. Depending on the conditional, the register-direct branch or the protection branch is squashed before it can affect control-related logic.

there is considerable variability in the dynamic behavior of such branches. Though such conditional branches are predominantly taken, three benchmarks (`gcc`, `go`, `perl`) have a nontrivial number of not-taken branches of this form. Hence, it is important that we do not compromise the operation of the branch predictor for these types of branches.

We handle this by mandating that this two branch structure shown in Figure 2 (which occurs only in some control instruction translations) has the following form: (i) the target of the protecting conditional branch must always be a new PowerPC instruction, (ii) the protecting conditional branch must always directly precede the requisite unconditional branch, and (iii) both branches must exist in the translation buffer at the time the protecting conditional branch is dispatched. Condition (iii) is easily handled by the design of the decode/translate unit in that it only emits the two branches as a pair subject to resource constraints. Conditions (i) and (ii) are handled by tagging the two branch structure at decode time and handling it as a special case at branch resolution time. The translation used guarantees that if the protection branch is taken, the result of the translated PowerPC instruction is a fall-through, and hence we squash the subsequent unconditional branch (because correct program behavior is fall through) and we indicate the inverted condition to the branch prediction/fetch hardware (i.e that the PowerPC branch was not taken). If the protection branch is not taken, it is guaranteed that the unconditional branch will be taken, and hence we squash the protection branch (by converting it to a nop) and evaluate the result of the unconditional branch with the existing branch prediction/fetch resolution hardware. Notice that in either case, due to our design, we end up with a single control instruction at the given logical PC after the protecting branch is evaluated. Therefore, from the execution back-end's point of view, only one control instruction exists at that PC, and hence we maintain the expected branch predictor, fetch, and retirement behavior.

We must also take special care when converting the PowerPC unconditional branch instruction, because it can produce a larger range of branch displacements than the equivalent PISA branch. In the cases in which the displacement is too large, we must convert this branch to three PISA instructions--two to load the branch address into a temporary register, and a register direct branch. The translator examines the displacement at PPC instruction decode

and emits the proper sequence depending on the displacement.

Finally, it is interesting to note that for all PowerPC control instructions, many different translations are possible for instructions of the same form, because of the flexibility offered in the PowerPC branch opcodes. For example, one of lengthiest branches to translate is the *bdnztlrl* form of the PowerPC *bc* instruction, which decrements the PowerPC counter register, sets the link register, and branches if the condition specified is false and the counter is non-zero. In our current implementation, it takes 13 PISA instructions to implement the semantics of this single PowerPC instruction. Of course, converting all conditional branches to 13 PISA instructions would cause programs to be dominated entirely by the execution of instructions associated with branches, given the relative frequency of branches in general programs [5]. Part of the ongoing refinement of the simulator is to determine which branch types are used frequently and to emit very efficient PISA sequences for those common types and to default to longer, more general sequences for less common encodings. We discuss our initial evaluation of the translation mechanism's efficiency in Section 5. As the simulator progresses, we also plan to examine the execution efficiency of our translated branch sequences, as compared with implementing the complex branches directly. We may also extend SimpleScalar/PISA to exploit the usage of the branch hints present in each PPC branch instruction.

4.2 Memory Operations

Although there is a one-to-one mapping between many of the PowerPC and SimpleScalar memory operations, there are a few complications. The first complication is that PowerPC allows unaligned memory references, while the SimpleScalar architecture does not. For some operations, our translator can statically determine whether or not a memory operation is aligned. However, the majority of memory operations use addressing modes in which some part of the address is stored in a register [5]. In these cases, our translation mechanism cannot statically determine if an operation will be aligned. The translator can derive this information dynamically, but must first allow the processor pipeline to drain, because the register used in a memory operation may be overwritten by an in-flight instruction. We do not want to pay the penalty of draining the pipeline for every register-addressed memory operation, so we have found an alternative solution.

Table 2: Branch and Memory Reference Characteristics

Benchmark	% branch register not-taken	% refs unaligned	% refs predictable (8 entries)	% refs indexed	% length predictable (256 entries)
compress	2.79%	.00%	72%	.06%	98%
gcc	15.98%	.01%	93%	4.43%	99%
go	8.45%	.00%	74%	.03%	98%
ijpeg	3.83%	.02%	86%	8.67%	100%
li	2.07%	.00%	75%	.04%	98%
m88ksim	2.48%	.00%	73%	.03%	98%
perl	7.64%	.00%	12%	2.52%	91%
vortex	2.34%	.00%	58%	13.6%	100%
Java TPC-W	2.61%	.34%	29%	10.67%	100%
DB2 TPC-B	1.17%	.00%	40%	.05%	100%

Because we have an out-of-order processor model with support for precise interrupts, we can translate and issue memory operations under the assumption that they are aligned. If the translated instructions attempt an unaligned access, we throw a soft exception, drain the pipeline, and retranslate the faulting instruction. During this retranslation, we can issue the appropriate sequence of aligned memory operations to emulate a single unaligned access. We call this approach *speculative decode*, since a static instruction is speculatively and optimistically decoded

to a simpler sequence that exploits a runtime attribute (namely, natural alignment), rather than nonspeculatively decoding it to a sequence of instructions that will work correctly in all cases.

The third column of Table 2 shows the frequency of unaligned memory references for the SPECint95 benchmarks as well as our Java TPC-W workload[15] and the TPC-B transaction processing workload [14], when run on the SimOS-PPC functional simulator. In general, there are very few unaligned references in these benchmarks. The fourth column shows the percentage of unaligned references that can be identified at runtime using a simple predictor. In this case, the predictor is a fully-associative memory that holds the PC of up to eight load or store instructions. Such a mechanism predicts 12% to 93% of all unaligned references, and can be used to guide the speculative decode process to eliminate a large fraction of the soft exceptions used for recovery.

An additional complication of the PowerPC architecture are the *stswx* (store string word indexed) instructions and *lswx* (load string word indexed) instructions. These instructions write/read a variable number of bytes to/from memory, specified by a register (up to 128 bytes). In all other cases we are able to statically determine the number of translated instructions necessary for a single PowerPC instruction. In this case we cannot, because in-flight instructions may change the value of the length register between the instruction's translation time and the time at which it should execute. The fifth column of Table 2 shows the percentage of all load and store instructions that are indexed. There is a great deal of variation, with four benchmarks showing a significant fraction of indexed loads (*gcc*, *ijpeg*, *vortex*, *Java TPC-W*).

Because of the relative frequency of this type of instruction in some critical benchmarks, we investigated the possibility of using a length predictor to guide our *speculative decode* mechanism for the indexed load and store instructions. As shown in the rightmost column of Table 2, such a predictor can be quite accurate. In this case, we are using a standard last value predictor [11] indexed by the load or store PC, which remembers the dynamic length of the previous instance of that indexed load or store (we do not pollute the table with values for loads and stores that have fixed lengths). As we can see, even a small direct-mapped predictor with only 256 entries achieves prediction accuracies of greater than 90% in all cases, and nearly 100% for most cases.

4.3 Exception-generating Instructions

Because each of the translated PISA instructions carries the same program counter value as the PowerPC instruction from which it was translated, we can guarantee that precise exception semantics will be maintained for exception-causing translated instructions. A more general problem is that the SimpleScalar simulator only supports user-level code, and does not support exception handling. For this reason, in the future we must make modifications to the SimpleScalar simulator to support PowerPC exception semantics.

5 .0 Instruction Translation Efficiency

Although our simulator is still in the development phase, we are currently able to execute several small benchmarks, including two of the SPECint95 benchmarks, `go` and `li`. We have performed instruction by instruction verification that the execution of these benchmarks is correct, comparing our translating simulator to a reference out-of-order PowerPC processor simulator. In this section, we present some initial results regarding the dynamic translation of these benchmarks.

For the two SPECint95 benchmarks, the dynamic instruction expansion caused by PowerPC to PISA translation varies considerably. As we can see from Table 3, The dynamic instruction count expands by 35% and 86% for `li` and `go` respectively. The primary causes of instruction expansion for both benchmarks are the branch and branch conditional instructions. As described above, these PowerPC instructions require many PISA instructions to implement their powerful semantics. Even for the simplest PowerPC conditional branch, two PISA instructions are required to test the PowerPC condition register field and conditionally take the branch. We are not satisfied with these

instruction expansion numbers, and plan to optimize our instruction translations to reduce this code expansion. This work will be necessary before using this simulation infrastructure for processor core-level architectural research.

Table 3: Dynamic Instruction Count Expansion for all Instructions and Memory Operations

Benchmark	Instruction Count			Memory Operation Count		
	PowerPC	PISA	growth	PowerPC	PISA	growth
`go`	79,457,833	147,418,082	86%	31,855,629	31,849,123	1%
`li`	53,964,399	72,614,311	35%	25,366,976	25,368,880	1%

Despite the increase in overall instruction count, Table 3 also shows that the number of memory operations executed by our translating simulator and our reference simulator are nearly the same. The number of memory operations from our simulator is slightly larger because our simulator performs PowerPC string manipulation instructions (which load or store up to 128 bytes) as a series of smaller memory operations, while the reference simulator counts each multiple-load/store string instruction as a single operation. Given these memory operation expansion results, we believe that the translation process does not materially affect the processor's behavior with respect to the memory system.

6 .0 Simulator Verification

Of course, because we plan to use our simulator for architectural evaluation, we must guarantee that it implements the specified program semantics. In order to enable this verification, we have leveraged existing PowerPC simulators [1, 6]. After the execution of each instruction on our simulator, we compare its machine state to the machine state of a separate verification simulator. Using this technique, we have verified the correctness of our simulator for all benchmarks reported in this paper.

Once we have verified that our simulator is functionally correct for a wider range of benchmarks, we must also verify that the performance reported by the simulator accurately reflects the performance of similar PowerPC processor implementations. We plan to perform this verification by running identical workloads on both the simulator and actual hardware, and comparing the simulator's performance results to performance results collected with hardware performance monitoring counters. If the use of dynamic translation skews the performance results excessively, we will refine our translations, in some cases adding direct support for complex PowerPC instructions in the SimpleScalar simulator, rather than generating long sequences of SimpleScalar instructions.

7 .0 Conclusions and Future Work

As mentioned above, our implementation is still in a preliminary phase. Our work on *speculative decode* appears promising, with simple prediction techniques resulting in up to 93% accuracy for memory reference alignment and virtually 100% accuracy for memory reference length prediction. Once we have worked out all of the bugs for user-level single-threaded programs, we plan on incorporating our translation engine into the SimpleMP simulator, a multiprocessor simulator based on SimpleScalar and developed by Ravi Rajwar at the University of Wisconsin. Once we have a working multiprocessor simulator, we plan on incorporating our detailed timing simulator into the SimOS-PPC full system simulator, and also extending it to support system-level instructions.

We also plan to extend our work on speculative decode to study further optimizations that take advantage of runtime characteristics when decoding instructions from one instruction set to another. These optimizations include varying code generation for loads that are likely to alias to earlier stores in the store queue (similar to the adaptive scheduling mechanism in the Alpha 21264 [7]), varying code generation for loads that are likely to miss in the L1

cache (again, as in the Alpha 21264 [7]), varying code generation for verifying and eliminating stores that are likely to be silent [12], and optimizing pairs or sequences of instructions by compounding them into more complex instructions (as studied in [13], for example).

8 .0 Acknowledgments

This work was supported in part by a grant from the National Science Foundation. We would like to thank the IBM and Intel corporations for their generous equipment donations, and also the anonymous reviewers for their insightful comments on this work. We would also like to thank Karthikeyan Sankar for his PowerPC simulator, which has been used as a reference throughout our development.

References

[1] D.C. Burger and T. M. Austin. “The SimpleScalar Tool Set, Version 2.0," University of Wisconsin Computer Sciences Technical Report #1342, June, 1997.

[2] D. Christie, “Developing the AMD K5 architecture,” *IEEE Micro*, 16(2):16-26, 1996.

[3] K. Diefendorff, “Power4 Focuses on Memory Bandwidth”, *Microprocessor Report*, 13(13):1-8, 1999.

[4] M. Gschwind, E.R. Altman, S. Sathaye, P. Ledak, and D. Appenzeller, “Dynamic and Transparent Binary Translation”, *IEEE Computer*, 33(3):54-59, 2000.

[5] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, California.

[6] T. Keller, A. M. Maynard, R. Simpson, and P. Bohrer. *SimOS-PPC full system simulator*. http://www.cs.utexas.edu/users/cart/SimOS.

[7] R. E. Kessler, “The Alpha 21264 Microprocessor”, *IEEE Micro*, March-April, pp. 24-36, 1999.

[8] C. May, E. Silha, R. Simpson, and H. Warren, *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, 2nd ed. Morgan Kaufmann, San Francisco, California. 1994.

[9] David B. Papworth, “Tuning the Pentium Pro Microarchitecture”, *IEEE Micro*, 16(2):8-15, 1996.

[10] C. Zheng and C. Thompson, “PA-RISC to IA-64: Transparent Execution, No Recompilation”, *IEEE Computer*, 33(3):47-52, 2000.

[11] M. Lipasti and J.P. Shen, “Exceeding the Dataflow Limit via Value Prediction”, Proceedings of MICRO-29, Paris, France, November 1996.

[12] K. M. Lepak and M. H. Lipasti. “On the Value Locality of Store Instructions.” In Proceedings of *ISCA-2000*, Vancouver, B.C., June, 2000.

[13] S. Vassiliadis and B. Blaner and R.J. Eickemeyer, “SCISM: A scalable compound instruction set machine,” *IBM Journal of Research and Development*, 38(1):59-78, 1994.

[14] Transaction Processing Performance Council. *http://www.tpc.org*.

[15] Todd Bezenek, Harold Cain, Ross Dickson, Timothy Heil, Milo Martin, Collin McCurdy, Ravi Rajwar, Eric Weglarz, Craig Zilles and Mikko Lipasti. “Characterizing a Java Implementation of TPC-W,” Third Workshop on Computer Architecture Evaluation Using Commercial Workloads, Tolouse, France, January 2000.