BOOST: Berkeley's Out-of-Order Stack Thingy

Steve Sinha, Satrajit Chatterjee and Kaushik Ravindran
Department of Electrical Engineering and Computer Science
University of California, Berkeley
{ssinha, satrajit, kaushikr}@eecs.berkeley.edu

Abstract

We present a novel scheme based on the Tomasulo algorithm to implement out-of-order execution in stack machines. This scheme not only reduces redundant data movement within the processor core, but also eliminates the stack cache and the associated problem of managing it. We present some preliminary quantitative evaluation of the proposed technique. For a suite of scientific benchmarks we obtain performance that appears to be competitive with out-of-order superscalar GPR processors.

1. Introduction

Stack machines are not popular among computer architects – the presence of a single architectural bottleneck, namely the stack, is viewed as a significant obstacle in the dynamic extraction of instruction-level parallelism. This project is an attempt at designing a scheme to uncover the latent parallelism in the sequential stack code at runtime. Although this project began as a somewhat academic exercise, we believe that our technique is not impractical and can serve as the basis for future high performance computer architectures.

Stack machines have a certain minimalist elegance about them. However, the absence of registers provides more than conceptual economy. By implicitly referring to operands, stack code is often shorter than general-purpose register code. Koopman estimates that stack machine programs are at least 3.75 times smaller than RISC code [1], though it must be noted that any measure of program size strongly depends on the language and compiler used. A less obvious reason for a stack machine having compact code is that they efficiently support code with frequently used subroutines. Since all working parameters are always present on the stack, procedure call overhead is minimal, requiring no memory cycles for parameter passing. Another advantage of stack machines is that on interrupts there is no need to save registers. Thus interrupt response latencies are low, making stack architectures ideal for real-time systems. Stack machines also support modular code that has many short function calls (such as object-oriented programs) better than GPR (General Purpose Register) machines, since registers need not be saved before every function call.

Critics of stack machines argue that these features come at a price. One of the largest of the criticisms is the lack of random access – all data must be computed on the top-of-stack. This excessive traffic to move operands in position to be worked on wastes data memory bandwidth. The stack cache is another issue discussed. Most of the stack machines reported previously in the literature have some sort of stack cache on the processor which serves as the architectural analogue of the register file in GPR machines. Therefore, even though the programmer need not explicitly save registers, the underlying processor still has to ensure that the stack cache "*follows*" the instruction stream. Architecturally, it maybe useful, as the processor can optimize when and how it writes out the stack cache to memory instead of relying on the programmer (or the compiler) to specify it explicitly as in the case of GPR machines[1]. Finally, as mentioned, the stack acts as a serialization point; since each instruction has to work on the top of the stack, it is not semantically possible to have two

[1] Observe that the "sliding window" register file in the SPARC architecture is an attempt to provide a similar behind-the-scenes register management, at the cost of making the architecture more complicated.

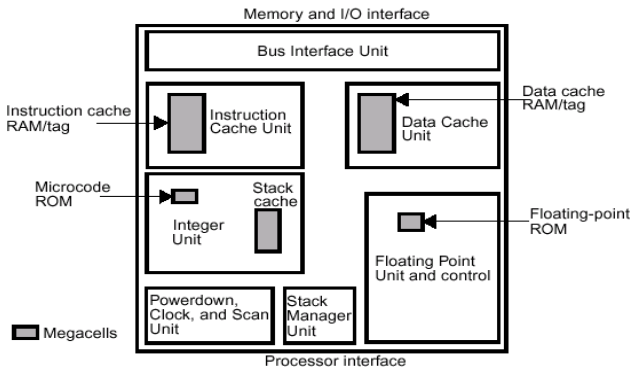


Figure 1 Block diagram of the picoJava-II processor core.

instructions of a single thread to simultaneously run at one moment. This seemingly removes the possibility of exploiting parallelism in the code.

Our goal is to remove the apparently inherent inefficiencies of the stack machine. Taking a page from the industry-standard GPR machine, we have developed a new stack architecture that implements the Tomasulo algorithm to create a micro-data flow engine. With our scheme, we are able to issue multiple instructions out-of-order, remove the stack cache, and remove the need for excessive movement of data objects. We further show that it is possible to extract a large amount of parallelism within stack code, up to a point that it is comparable to GPR machines.

In the next section, we further motivate our work, looking at past stack machines for comparison. In Section 3, we discuss the details of our proposed BOOST architecture. We describe our simulation efforts as well as the results we found, in Section 4. Finally, in the last section, we interpret our findings, discussing the possible consequences of our research.

2. Motivation and Previous Work

There have been many different stack micro-architectures reported in the literature [1, 2, 4]. In this section, we attempt to identify some commonalities of those architectures. Rather than referring to stack architectures in general, we use the SUN Microsystems picoJava-II core as a concrete example of the class of traditional stack micro-architectures [5, 6, 7]. Since this is a recent, yet traditional design, it incorporates most of the advances and salient features of stack machines. The high-level architecture diagram of the picoJava-II core is presented in Figure 1.

The picoJava-II is a typical 0-operand, single issue, single stack processor. Like other traditional stack architectures, it uses a stack cache to cache the topmost entries in the stack. However, the stack cache is not organized as a hardware stack. Instead, it is built much like a traditional GPR machine register file. It has multiple read and write ports and supports random access to the data. The OPTOP register keeps track of the current top of stack. When the stack is modified by an instruction (such as a pop), this register is updated to reflect the new top of stack and no data is actually moved in the stack cache. The processor uses the OPTOP to keep track of the operands for the instructions. The VARS register contains the base of the current operand stack. Data values can be accessed/stored relative to the VARS register and moved to/from the OPTOP. The OPTOP, VARS and PC (program counter) are the only explicit storage locations in the system.

There are certain conceptual and practical problems with this scheme. Conceptually, the stack cache appears redundant, keeping track of data already present in the L1-cache and the store buffers. (Store buffers keep track of stores not yet written back to the L1-cache.) In addition, this mechanism doesn't eliminate all unnecessary data movement. Consider a stack instruction such as *dup* which duplicates the datum at the top of stack. As far as the stack program is concerned, the *dup* instruction is only a means of telling the processor that the element at the top of the stack will be used by some other instruction. No actual movement of data is necessary, but a stack cache-like implementation requires that the data be copied from one register in the stack cache to another.

On a practical level, there is the problem of keeping the stack cache current with the top-of-stack. For example, when the stack grows beyond the stack cache, some of the bottom

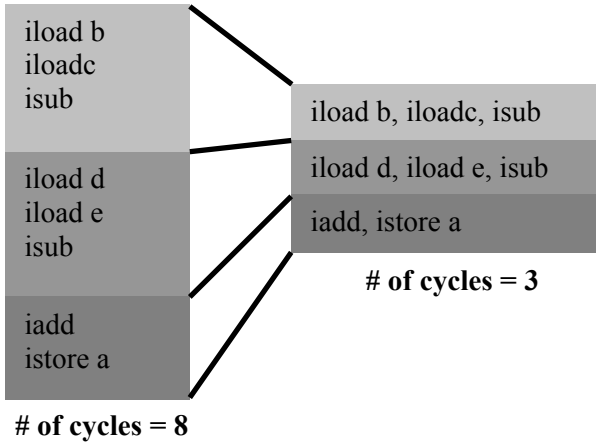


Figure 2 An example illustrating instruction folding.

entries need to be written out to memory. This is often in the critical path of the program requiring subsequent instructions to be stalled until this happens. PicoJava has a workaround for this. The programmer can set high- and low-watermarks that prompt the flushing out (or reading in) of a certain number of cache elements to (or from) memory proactively. Note that this setting has to be done statically, and fails to use any dynamic, auto-tuning mechanism that naturally adapts to the dynamic program behavior.

One of the advances in the picoJava-II architecture is the use of *instruction-folding* [2, 3]. In this technique, certain sequences of instructions (such as say `load load add store`) are collapsed into a single RISC like instruction during decode. An example of instruction-folding for a sequence of eight instructions is shown in Figure 2.

Instruction-folding is done to exploit the random access provided by the stack cache and to reduce movement of data for the most common groups of instructions. Unfortunately, the permissible groups of instructions that can be folded in this manner are limited in number and scope. In addition, not all redundant data moves are avoided. Although redundant moves between instructions in a group are avoided, there is still forwarding of data through the registers in the stack cache between groups.

The approach we propose is a generalization of instruction folding based on the Tomasulo algorithm [9]. By using a forwarding scheme through reservation stations, we are able to get rid of the stack cache. The reservation stations along with the store buffers serve as the mechanism for data forwarding between instructions. Since there is no explicit stack cache, there is no need to explicitly keep it consistent with the instruction flow. *The reservation stations, the regular load-store mechanism (especially hoisting loads over outstanding stores) and the data cache work in a concerted manner to automatically manage the stack in a dynamic fashion.* In the following section we describe this scheme in greater detail.

3. The BOOST Micro-architecture

The Boost micro-architecture seeks to incorporate a basic Tomasulo-like micro-data flow engine into a stack core to dynamically extract instruction-level parallelism. The sequential semantics of stack machines makes this a challenging problem.

3.1. Description of the BOOST architecture

The Boost architecture extends the basic picoJava-II processor core with features to support out-of-order execution. A block diagram of the architecture is presented in Figure 3.

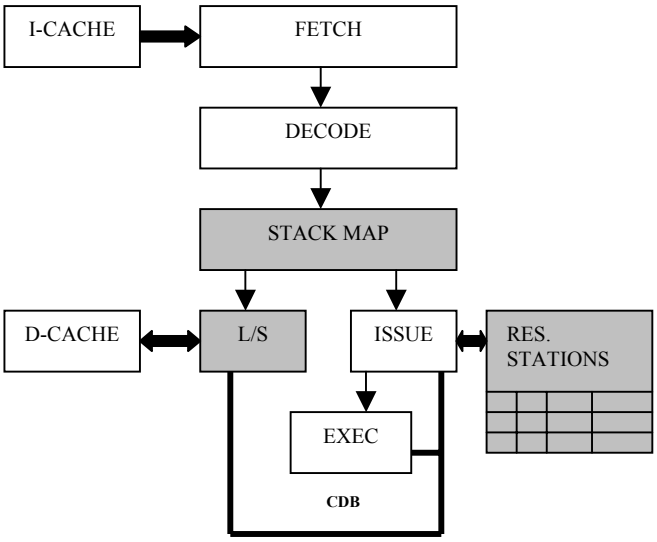


Figure 3 Block diagram of the BOOST micro-architecture

The basic pipeline decomposition of BOOST is very similar to that of picoJava-II. Two notable additions are *reservation stations* (to store instructions waiting to execute) and a *stack map* (to associate memory addresses with reservation station entries). Reservation stations in the Boost architecture play a role similar to their function in superscalar RISC machines. An instruction is ready to execute if all its data operands are ready and a functional unit is available.

The *stack map* is a table that associates a stack address with a reservation station entry. The destination address of each instruction corresponds to a reservation station entry for that instruction, and this association is stored in the stack map. Its analogous counterpart in the superscalar RISC framework would be the architected register file. Traditionally, register files have been thought of as an area of storage for fast access to data that is most frequently accessed for computation. However, with the inception of superscalar architectures, *the actual data is hardly ever present in the registers themselves*. Rather, most of the data is constantly in flight and passed between dependent instructions through reservation station entries. The register file instead serves as a mapping to associate absolute data addresses (as specified in the instruction as source and destination registers) and their corresponding aliased address (the reservation station entry that will produce the data for that register). The architected register file contains actual data only when there is a need to preserve state before an interrupt or context switch.

In BOOST, the stack map plays this role of mapping memory addresses to their corresponding reservation station entry, similar to the function of the architected register file in RISC. This way, the stack map facilitates '*address renaming'*, an important aspect of out-of-order execution. The register file in RISC additionally serves to capture state of the execution before a context switch. However, this is not required of the stack map since context state is inherently preserved in the stack.

The load/store buffer in BOOST is similar to its role in typical RISC processors. It is a buffer for outstanding loads and stores from/to the D-cache. However, potentially every instruction can cause a load or store since all operations are memory transactions in the BOOST stack architecture. Further, a majority of the instructions in stack programs are explicit loads and stores. Hence, the role of the load/store buffer is critical to efficient execution. A good percentage of memory accesses are bypassed through the reservation stations and stack map without hitting the load/store unit or D-cache. For those instructions that miss in the stack map, the store buffer is checked to determine if the required data is present. The D-cache is visited only when a miss in both the stack map and the store buffer occurs.

With out-of-order execution comes the potential to issue multiple instructions in parallel and improve throughput of the architecture. However, this significantly increases the complexity of the DECODE stage to decode these instructions. This problem is further complicated due to the fact that the dependencies between these instructions are implicit in the stack framework. In the RISC paradigm, dependencies are explicitly specified through the absolute location (register or memory) where instruction operands can be found. However, with stacks, all operand

The Adolescence of an Instruction - An Illustrative Decode example

(P): Parallel operation (S): Sequential operation

Instruction Package:

Load 2	Store 0	Add	Load 3	Load 0	Add	Add	Store 0

Starting Parameters: **OPTOP** = aaaa0010H **VARS** = aaa90000H (word addressing used)

Step 1: Lookup relative change to OPTOP for each instruction **(P)**

+1	-1	-1	+1	+1	-1	-1	-1

Step 2: Compute absolute change to OPTOP for each instruction **(S)**

+1	0	-1	0	+1	0	-1	-2

Step 3: Compute OPTOP for each instruction **(P)**

aaaa0010	aaaa0011	aaaa0010	aaaa000f	aaaa0010	aaaa0001	aaaa0010	aaaa000f

OPTOP at start of next package equals aaaa000eH

Step 4: Calculate absolute addresses of Sources and Destination **(P)**

Source 1:

aaa90002	aaaa0010	aaaa000f	aaa90003	aaa90000	aaaa0010	aaaa000f	aaaa000e

Source 2:

-	-	aaaa000e	-	-	aaaa000f	aaaa000e	-

Destination:

aaaa0010	aaa90000	aaaa000e	aaaa000f	aaaa0010	aaaa000f	aaaa000e	aaa90000

Step 5: Find dependences **(P)**

Each instruction is preassigned a token - the number of a free reservation station. If an instruction does not have a destination (such as instructions that modify control registers), the preassigned token is unused, free for future use.

t0	t1	t2	t3	t4	t5	t6	t7

A priority mux is used to assign each source the token (if available) of the closest preceding instruction in the package writing to the source address. (This is similar to hardware used in GPRs.)

Source 1:

aaa90002	**aaaa0010**	aaaa000f	aaa90003	**aaa90000**	**aaaa0010**	**aaaa000f**	**aaaa000e**

Source 2:

-	-	aaaa000e	-	-	**aaaa000f**	**aaaa000e**	-

Destination:

aaaa0010	aaa90000	aaaa000e	aaaa000f	aaaa0010	aaaa000f	aaaa000e	aaa90000
t0	t1	t2	t3	t4	t5	t6	t7

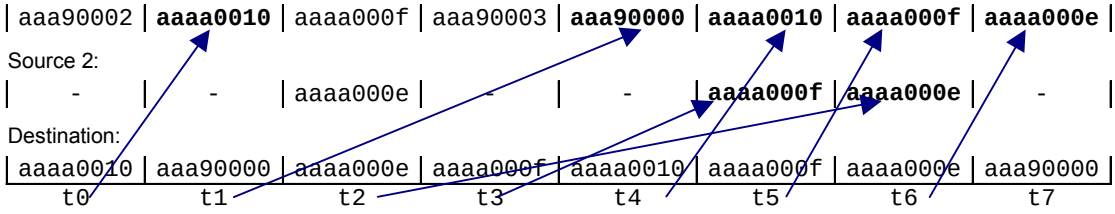
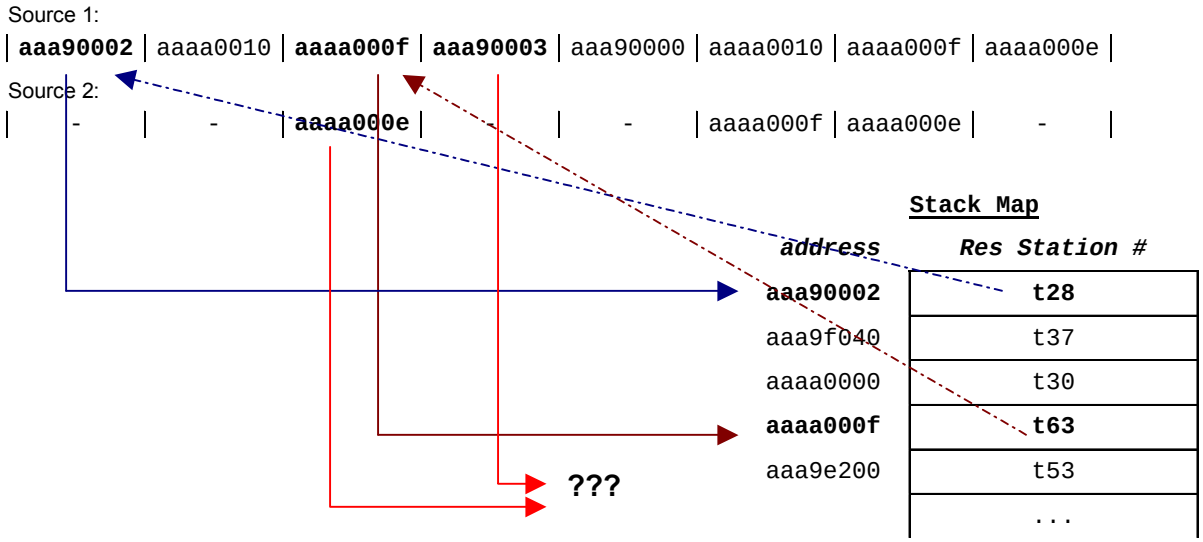


Figure 4 Illustrative Decode example

Step 6: Stack Map lookup **(P)**

If source isn't found within the package, the address is looked up in the stack map.



Step 7: Issue Loads **(P)**

If source address isn't found in the stack map, an implicit load is issued to the load store unit with a uniquely generated virtual token (vt). Special optimization for loads: use the destination token itself instead of the virtual token.

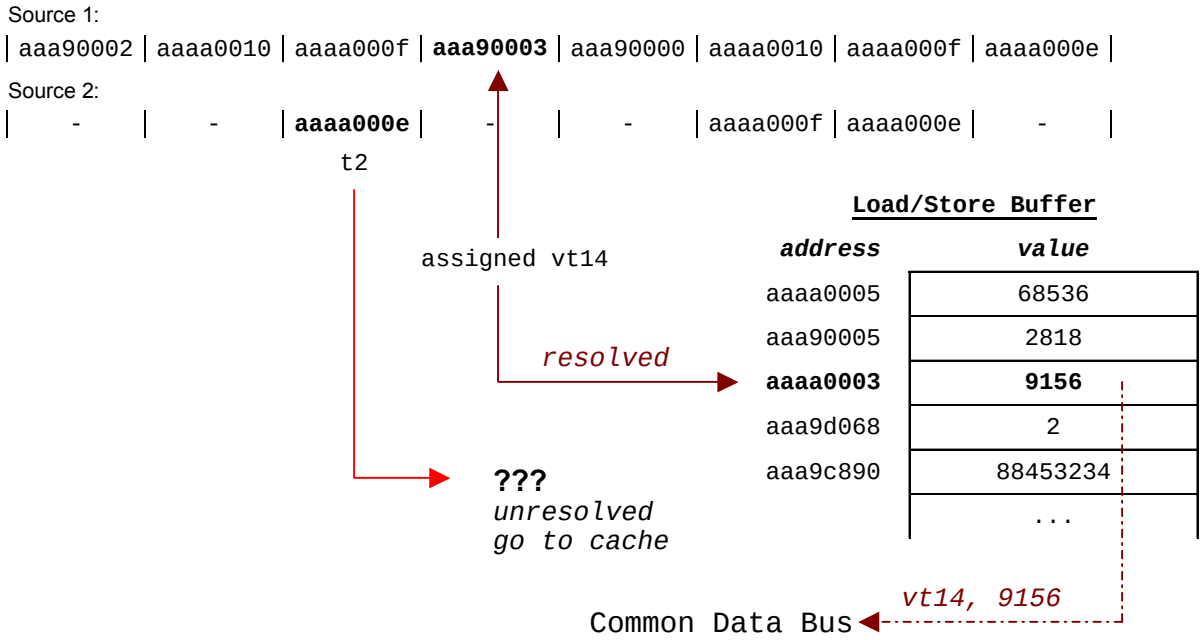


Figure 4 Illustrative Decode example, cont.

dependencies are based on variable positions relative to the top of stack. Additionally, the number of instructions that can be issued in parallel is also limited by the amount of instruction level parallelism (ILP) that be extracted from a stack program. To make the BOOST architecture feasible, we propose an efficient scheme for decoding multiple instructions and resolving operand dependencies in parallel to attain a reasonable degree of ILP.

A comprehensive example of instruction execution in the BOOST datapath is illustrated in Figure 4. Specifically, the strategy for resolving operand dependencies in the decode stage is illustrated. A micro-architect may split DECODE into multiple pipeline stages; we partitioned DECODE into two stages in our design: DECODE and STACK MAP. There are two kinds of dependencies handled in DECODE: (a) dependencies between instructions issued in the same packet (a packet is all instructions issued in parallel at the same cycle) and (b) inter-packet dependencies between instructions issued at multiple cycles. Once the instruction has been issued to the reservation station, the subsequent processes of execution and write-back is very similar to the style of RISC.

The major advantage of register and memory-memory architectures is the random access to data. However, in stack machines, all operations are restricted to the top two operands on the stack. The bottleneck this creates is the constant movement of data values to/from the top of stack before any interesting computation is performed on them. For example, 47 % of the instructions executed on the JVM stack are local variable load/stores [2]. The folding optimization discussed earlier reduces 6 % of all instructions on average. Nevertheless, it is clear that this movement of data to/from top of stack dominates overall percentage of bytecodes executed.

The introduction of reservation stations and stack map as a means for out-of-order execution simultaneously *alleviates this problem of data traffic in stack architectures*. Combined with the load/store buffer, these three entities provide a powerful abstraction of the stack itself. They are a way to store and address operands for execution and can effectively replace the stack cache used in traditional stack architectures.

4. Evaluation

The most important question that we wanted to answer is *how much parallelism could our algorithm extract from a stack instruction stream*. In order to do that, we could either define our own stack instruction set architecture or use a pre-existing one. Although with the first approach we could have had some flexibility with respect to defining the instruction semantics, we chose to go with the latter; by using an existing set-up, we had access to working compilers, and possible benchmark programs. Given the near-absence of commercial stack machines, the Java Virtual Machine (JVM) seemed like a good ISA. Unfortunately, since the JVM is primarily designed to be executed on a software machine, quite a few of the common instructions do not map well to hardware. Luckily for us, the PicoJava project at Sun had a working tool chain and a trace generator that had the trap emulation routines for these instructions.

4.1. Simulation Framework

To test the viability of our proposed architecture we turned to simulation. At first, we decided to write a micro-architectural simulator, but while designing the simulator we realized that we would have to make many arbitrary decisions about the micro-architectural details (for example deciding how many common data buses to have and how to arbitrate among the competing functional units). These decisions would not be germane to the basic feasibility study that we were hoping to do and furthermore would be quite meaningless outside the context of actually building such a processor. Therefore, we settled on a less accurate but more flexible simulation scheme.

The idea was to design a trace simulator to capture logical dependencies between instructions in the program trace and examine the limit on the instructions per cycle (IPC) possible. The BOOST architecture from Figure

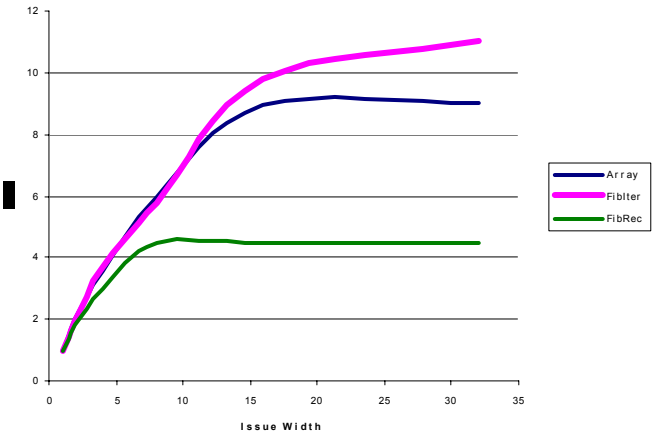


Figure 5 IPC vs. Issue Width for Micro-benchmarks

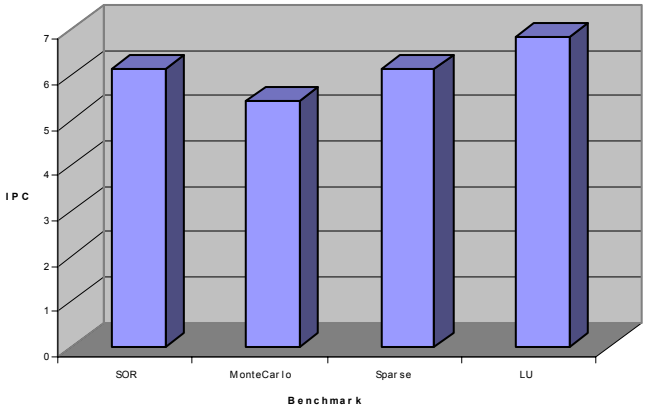


Figure 6 IPC for 4 SciMark 2.0 Benchmarks with 8-wide instruction issue.

3 was modeled in this simulator based on many assumptions about micro-architectural features. The major limitation in our model was the assumption of *no structural hazards*; this implied infinite reservation stations and functional units, and no contention for the common data bus. From a practical standpoint, this is a serious limitation since such structural hazards are a constraining factor on the limit of ILP that can be extracted. Hence, *the result of our simulation studies only find an upper bound on the parallelism possible*. Other parameters in our model were set to emulate the picoJava-II core. Some other less constraining assumptions were (a) constant 10 cycle miss penalty in L1 cache (b) 100 % hit rate in L2 cache (c) branches predicted with 100% accuracy (d) the fetch stage only accesses one cache-line at a time.

4.2. Benchmark Suite

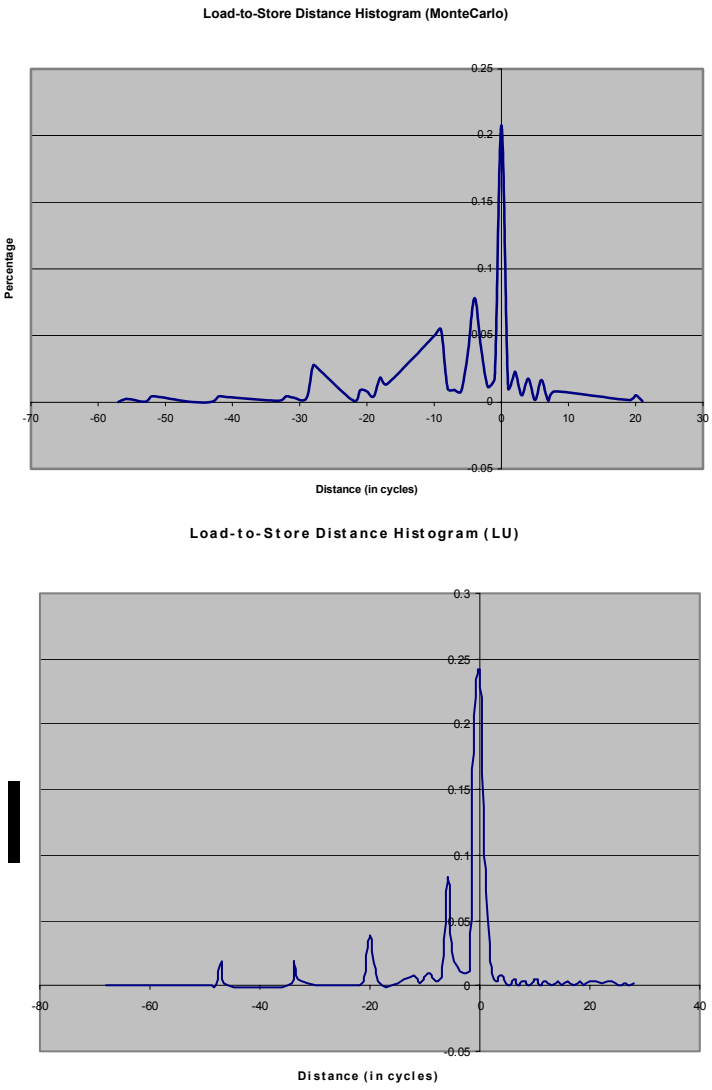
Choosing a benchmark suite turned out to be harder than we thought. Most of the commercial Java benchmarks are designed to evaluate the performance of Java Virtual Machines. They tend to exercise the graphics libraries, the networking layer, threads, etc. Since the PicoJava environment doesn't have even a operating system, we could not use the standard JVM benchmarks.

Hence, some benchmarks from the SciMark (Java) 2.0 suite (SOR, Sparse, LU, and Monte Carlo simulations) were used to evaluate our model [8]. Apart from this, we tested our model on some hand-written micro-benchmarks comprising of Fibonacci computation (iterative and recursive) and array manipulation (without any loop-carried dependence).

4.3. Simulation Results

The first parameter investigated was the amount of instruction level parallelism (ILP) possible as a function of the issue width. The number of instructions committed per cycle (IPC) is an approximate measure of the number of instructions executed in parallel (ILP). A graph of IPC as a function of the issue width for the three micro-benchmarks is displayed in Figure 5.

As expected, there is a bound on the amount of ILP possible in all the three programs. This is due to the inherent logical dependencies between instructions that constrain how many of them can be executed simultaneously. The IPC of the programs saturate beyond a certain instruction issue width. The array manipulation and iterative Fibonacci benchmark are highly parallelizable code and hence reach a greater IPC compared to recursive Fibonacci. The IPC levels obtained for these programs are decently high; but it must be noted these are micro-benchmarks and the simulation model is not realistic.



Figures 7 and 8
Histogram of distances between dependent instructions

Typically, one RISC instruction corresponds to approximately 1.88 stack instructions [1]. Modern superscalar RISC machines allow upto 3 parallel instructions per cycle without complicating the decode logic. To be compatible with RISC machines in this measure, stack machines must expect to commit up to 5.7 instructions per cycle. To achieve this IPC, the number of instruction issues required per cycle must be between 7 and 8 (from Figure 5). Based on this statistic, we have incorporated an 8-wide instruction issue for the BOOST architecture. The ability of an 8-wide instruction issue to support a IPC of around 6 was confirmed for various benchmarks in the SciMark 2.0 suite (Figure 6). This is a reasonable issue width that can be efficiently handled by our decode mechanism presented in Figure 4. The presence of structural hazards in practical designs will restrict the maximum IPC possible – a constraint we have not been able to currently analyze due to the limitations in our simulation model. Nevertheless, pending further investigation, we believe it is possible to achieve parallelism compatible to modern superscalar machines in stack architectures.

The strategy to introduce out-of-order execution to extract parallelism in stack architectures also provides a solution to the data traffic to/from top-of-stack, as data dependencies can now be bypassed via address renaming and reservation stations. This significantly improves system performance by obviating redundant memory transfers (considering that 50% of executed instructions are memory transfers to/from top of stack). This is an architecturally elegant solution compared to instruction folding used in picoJava-II. The stack is now abstracted into the reservation station, stack map and load/store buffer and the overhead due to management of a hardware stack cache is avoided.

In order to verify this, we analyzed the relative *distance* (clock cycles) between the commit of a data store and the corresponding issue of a dependent data load in the SciMark 2.0 programs (Sparse, Monte Carlo, SOR and LU simulations). Most stack instructions do a load to or store from memory and the dependency between memory addresses is inherent in stack programs. The results of this analysis are illustrated in the graphs presented in Figures 7 and 8.

These graphs represent the percentage of instructions for different values of distance between a data load and a previous store (the area under the curve is effectively 1). It is observed that in all these benchmarks, the maximum percentage of instructions exhibit a distance less than or equal to zero. For instance, a distance of –20 cycles means that the source operand for that particular instruction will be available only 20 cycles in future, implying that

the operand is currently being computed or waiting in a reservation station. This denotes that most operands can be bypassed between reservation stations without any invocation to memory. The instructions with distances greater than zero imply that their operands have already been computed and hence cannot be passed via reservation stations. It is only in these cases that instructions have to fetch their operands from lower levels of memory, namely the store buffer or the D-cache (if miss in the store buffer). However, the percentage of such instructions significantly less, endorsing the reduction in memory traffic due to unneeded loads and stores.

5. Conclusion and Future Work

In this paper we have presented a novel scheme to implement out-of-order execution in stack machines. To our knowledge this is the first such scheme proposed in the literature. We believe that the proposed scheme not only exposes the instruction-level parallelism in the stack code, but also provides an elegant solution to two problems in existing stack machines: namely, the problems of redundant data movement and of having to manage a separate stack cache.

Our preliminary experiments with a suite of numerical benchmarks show that it is possible to extract significant parallelism from stack code. Furthermore, this parallelism (quantified in terms of IPC) appears large enough to make out-of-order stack machines competitive with general-purpose-register machines (which have a lower instruction count than stack machines).

There are a couple of obvious ways of extending this work. One would be to refine the simulator to make it more accurate (perhaps even design a preliminary micro-architecture). Another would be to run the simulations over a larger variety of benchmarks form different application domains. A harder task would be to quantitatively compare out-of-order stack machines with GPR machines in some generalized setting.

Finally, to put this paper in perspective, it might help to consider the following. Over the past decade, the unrivalled success of the x86 architecture has convinced many that the particular choice of ISA in designing a microprocessor doesn't matter anymore. The internal organization of the microprocessor has little to do with the façade that the instruction set exposes. Extending this principle, there is no reason why a microprocessor that exposes a stack instruction set should internally be organized as a stack machine. Indeed, one of the contributions of this paper is that *not* organizing it as a stack machine allows us to extract more instruction-level parallelism and obtain performance competitive with GPR machines. Given the inherent elegance of stack ISAs (in terms of conceptual economy) and the practical advantages (denser code, more 'natural' code[2], faster context switches, etc.), we tentatively suggest that perhaps stack ISAs might be the ISA of choice for future processor designers.

Acknowledgements

We thank David Culler for guiding us in the direction of this work and for having faith in architecture.

References

[1] P. Koopman Jr., "Stack Computers – The New Wave". © 1989, Mountain View Press.

[2] N. VijayKrishnan, "Issues in the Design of a JAVA Processor Architecture". PhD dissertation, University of South Florida, Tampa, FL-33620. December 1998.

[3] R. Radhakrishnan, D. Talla, and L. K. John. Allowing for ILP in an embedded Java processor. In Proceedings of the 27th International Symposium on Computer Architecture, pages 294--305, June 2000.

[4] R. Radhakrishnan, N. Vijaykrishnan, L. John and A. Sivasubramanium, "Architectural issues in java runtime systems," Tech. Rep. TR-990719, 1999.

[2] Since most compiler generate stack-based code for languages such as C with re-entrant functions, most GPR machines wind up emulating stack machines to a significant extent.

[5] "picoJava-II™ Microarchitecture Guide", Part No.: 960-1160-11, March 1999, © 1999, Sun MicroSystems.

[6] "picoJava-II™ Programmer's Reference Manual", Part No.: 805-2800-06, March 1999, © 1999, Sun MicroSystems.

[7] Harlan McGhan and Mike O'Connor. "PicoJava: A direct execution engine for Java bytecode". Computer, 31(10):22--30, October 1998. (p 106)

[8] "SciMark 2.0 Benchmarks", http://math.nist.gov/scimark2/.

[9] R. Tomasulo. "An efficient algorithm for exploiting multiple arithmetic units", IBM J. Research and Development, 11:1, January, 1967. (p 25-33).