THE SYMBOLICS VIRTUAL LISP MACHINE OR USING THE DEC ALPHA AS A PROGRAMMABLE MICRO-ENGINE [1]

(EXTENDED ABSTRACT)

P.T. Withington, Scott McKay, Gary Palter [2]

Symbolics Inc.
6 Concord Farms
Concord, MA 01742-2727, U.S.A.
ptw@Riverside.SCRC.Symbolics.COM
1+ 508 287 1000

Paul Robertson [2]

Dynamic Object Language Group
2 Parsonage Hill
Haverhill, MA 01832, U.S.A.

INTRODUCTION

Symbolics' Genera system represents the accumulation of nearly three and a half decades of software engineering tools, written in several dialects of the Lisp language and several object-oriented extensions to Lisp. Traditionally, Genera has run on a "Lisp Machine", a line of custom hardware workstations designed specifically to execute the Lisp language. Because of the limited market, this custom hardware has never been able to take advantage of cutting-edge technology such as is available in commodity, RISC-based workstations. At the same time, non-standard hardwares such as the Lisp Machine have fallen into economic disfavor. Nonetheless, Lisp's (and the Lisp Machine's) capability in prototyping, evolutionary problem solving, and super-complex problems has retained a small but dedicated market.

In response to market pressure to provide Genera's capabilities on commodity hardware, Symbolics chose to implement an "emulator" that would execute the Lisp Machine instruction set on a standard platform, rather than to port the approximately 1.5 million lines of Genera source code to a single Lisp dialect, as would be necessary to take advantage of a native Lisp

[1] Symbolics® and Genera® are registered trademarks and Virtual Lisp Machine™ and VLM™ are trademarks of Symbolics, Inc. DEC® is a registered trademark and Alpha™ and AXP™ are trademarks of Digital Equipment Corporation.

[2] Author's electronic mail addresses are ptw@Riverside.SCRC.Symbolics.COM, SWM@Stony-Brook.SCRC.Symbolics.COM, Palter@Stony-Brook.SCRC.Symbolics.COM, and PRobertson@ACM.ORG, respectively.

compiler on the same platform. It was felt that this approach would result in a shorter time-to-market while still preserving the robustness and flexibility of Genera. At the same time, the emulator approach has meant that the most important features of the Lisp machine in support of Lisp have also been preserved. In particular, the tagged memory architecture supporting dynamic typing, the fast exception handling for complete instruction semantics, and read and write barriers in support of automatic storage management have been preserved.

The DEC Alpha RISC architecture offers a full 64-bit memory architecture and multiple-issue instruction execution. Using the 64-bit (byte-based) address space we were able to emulate within a single OSF/1 process both the 40-bit tagged data and 32-bit (word-based) address space of the Lisp Machine. With the multiple-issue instruction execution we were able to take the approach of using the Alpha as a programmable micro-engine and write the emulation of the Lisp Machine instruction set as if we were writing micro-code. We were successfully able to utilize the OSF/1 operating system facilities to replace the I/O subsystem of the traditional Lisp Machine at a high level, thus gaining significant I/O performance improvement. The user-program accessible memory protection facilities of OSF/1 allowed us to successfully emulate most of the hardware features of the Lisp Machine in support of "ephemeral" garbage collection.

The performance of the first version of the Virtual Lisp Machine running on a DEC Alpha AXP 500 is very competitive with the performance of Symbolics' most advanced custom hardware. (This performance can be expected to improve proportionally as DEC introduces more powerful versions of the Alpha.) We believe that a large part of this performance is due to the "micro-program" approach of the emulator, which utilizes the instruction and data caches of the Alpha in a quite different fashion than a traditional compiled version of the same Lisp program might. The full paper discusses the details of our design decisions, implementation, benchmarks, and some of the surprising results we observed in tuning the emulator to the Alpha architecture.

BACKGROUND

The sole purpose of a Lisp Machine is to support the execution of the Lisp language in hardware. To that end, all architectural features find their roots in Lisp. The memory architecture is "object-oriented"– every memory word contains an object in the form of a data-type (tag) and representation, either immediate (data) or as a reference (pointer) to the representation of the object. In addition to objects, there are a small set of "special markers" used for storage-management bookkeeping (mutating objects), monitoring (data and call tracing), and debugging (uninitialized data). The hardware understands "primitive" type representations.

One of the features of Lisp Machines (one of the technologies that allowed its creation) is the use of microcode to implement complex instructions closely tuned to the language-level concepts of Lisp. In the '70s, when the Lisp Machine was born, compiler technology was poor and microcoding simplified the compiler writer's task while maintaining performance. Today, advanced compiler technology appears to obviate the need for microcoding and complex instruction sets, but we discovered that the continual climb in processor clock rates (and resulting increasing mismatch between memory speeds and instruction execution rate) may mark the return of micro-programming as a valid implementation technology. In this case, by micro-programming, we simply mean another layer of abstraction between the underlying hardware execution unit and the instruction execution model the compiler has as a target, what has also been termed an interpreter.

A second enabling technology of the Lisp Machine, is the use of "tagged memory". While modern compiler technology (in particular, flow-analysis, type propagation, and block compilation) combined with careful type declarations on the part of the programmer can provide very competitive Lisp implementations

on standard architectures, it is not in the tradition of Lisp to require type declarations. The use of tagged memory to support dynamic typing and generic operations allowed the Lisp machine to give competitive performance in the absence of carefully declared types, a key feature in support of its rapid prototyping capability. Supporting tagged memory was an important goal of our emulator.

In the full paper, we give an overview of Lisp Machine architecture, its origin and current state. We discuss our earlier studies that had determined emulation on 32-bit address and data machines would have unacceptable performance. The advent of true 64-bit workstations allowed us to revisit that study and determine that a competitive product was possible.

DESIGN

We knew there were a number of challenges to be overcome in developing an emulator, but the performance bottleneck in all our studies was the sophisticated memory model of the Lisp Machine. The "object-ness" of memory is built in to the hardware at the lowest level. All memory accesses are data-sensitive, in support of dynamic typing, monitoring, and garbage collection. We knew from other work in the field that comparable garbage collection models were being supported using the simple page-protection features underlying most operating systems. The challenge was to convert the complex but venerable Genera garbage-collector to such a scheme with minimal changes. The full paper discusses the details of the conversion of the garbage-collector, including moving some of its supporting routines to "microcode" (i.e., implementing them as new instructions in the software emulator). The emulation of the data-sensitive memory operation is examined in detail and several generations of refinement show how we were able to take better and better advantage of our understanding of the Alpha instruction set.

The full paper also discusses other aspects of the emulator design and some of the lesser goals such as sharing data with other processes running under OSF.

IMPLEMENTATION

We built a prototype of the emulator in C, but it quickly became obvious that we could not achieve the level of performance desired in C. Examination of code emitted by the C compiler showed it took very poor advantage of the Alpha's dual-issue capabilities. A second implementation was done in Alpha assembly language and is the basis for the current product.

We built a number of tools (in Lisp, running on Genera) that supported the level of complexity of the assembly language program we were attempting. A translator was built that allowed us to use Lisp as a macro language. One of the primary benefits of using Lisp was that we could use all our normal development tools, including incremental patching, even though we were working in another machine's assembly language. Even more beneficial, however, was that early on in the project, we were able to easily graft on to the translator a cycle-counting tool that allowed one to easily and automatically "preview" any code fragment and see its total cycle cost, dual-issues that were taken or missed, and any free stall slots. Because this tool was integrated directly with the Genera editor, we were able to pro-actively optimize our code, right from the start. The full paper describes this tool in more detail, with examples, and compares it with tools that have recently become available from DEC that attempt to automatically re-organize executable files. It is our claim that our tool, because of its interactive nature, offers many more opportunities for optimization.

The architecture of the emulator as designed and eventually implemented is examined in detail in the full paper. Among the interesting details is the substitution for the hardware-supported display of an X-window interface. This came nearly for free, as earlier projects had already

developed a compatibility package that allowed Genera to use any X-server as a display. Similarly, earlier hardware projects to create co-processor systems, where our custom hardware would run inside another workstation, had already defined an architecture that allowed us to easily substitute OSF disk and network services for real hardware. One interesting aspect of this substitution, however, is that the emulator process acts as an independent host on the network (it is emulating a complete workstation) despite sharing underlying network services with the Alpha workstation.

PERFORMANCE

In order to compare the performance of the emulator against our custom hardware, we used the standard "Gabriel" Lisp benchmarks, plus our own "large program" and "user interface" benchmarks. For the most part, the benchmarks gave consistent results, but we were puzzled a number of times when changes that clearly had a global effect of reducing overall cycle counts showed inexplicably poorer benchmark results. After much puzzling and studying, we eventually realized that the Alpha's direct-mapped cache had to be taken into consideration. This resulted in an effort to organize our code, effectively into micro-code "overlays", by ensuring that commonly called routines and their siblings were placed contiguously, and that they would not collide with themselves in the cache. Similarly, the data structures of the emulator were reorganized and placed carefully in memory.

Version 1.0 of the emulator exceeded our initial performance goals, achieving nearly the performance of our high-end custom workstation. It is currently in use at a number of customer sites and being used internally for software development work. A second version is underway, where we are attempting to further exploit some of the lessons we learned in the first implementation.

CONCLUSION

It is our contention that although our Lisp code is interpreted, our performance may approach or even surpass that of compiled Lisp code on the Alpha because our interpreter is acting almost like a micro-program when it remains resident in the primary caches. The full paper examines some anecdotal evidence that supports this theory, gives some empirical data we have gathered that also supports this theory, and compares the idea with studies performed elsewhere showing the effect of cache-misses on high clock-rate CPU's. We speculate that as execution clock rates continue to increase (exceeding the speed of even first-level cache technology) we may come full-circle in computer architectures and return to a situation where micro-programming and tagged data are again valuable implementation techniques.