Communicating Process Architectures 2004
Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green (Eds.)
IOS Press, 2004

The *Transterpreter*: A Transputer Interpreter

Christian L. JACOBSEN
clj3@kent.ac.uk

Matthew C. JADUD
matthew.c@jadud.com

Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK

Abstract. This paper reports on the *Transterpreter*: a virtual machine for executing the Transputer instruction set. This interpreter is a small, portable, efficient and extensible run-time. It is intended to be easily ported to handheld computers, mobile phones, and other embedded contexts. In striving for this level of portability, occam programs compiled to Transputer byte-code can currently be run on desktop computers, handhelds, and even the LEGO Mindstorms robotics kit.

1 Introduction

occam [1] is an excellent language for reasoning about and writing programs dealing with concurrency and parallelism. Robots and other embedded systems are natural applications for occam, as are many programming problems which are often tackled with languages that have insufficient support for expressing concurrency [2]. We believe that writing programs for the LEGO Mindstorms [3], a small robotics platform produced by the LEGO Group, is a natural application of the occam programming language.

occam was originally the language of the Transputer, a microprocessor specifically designed for parallel processing. To encourage adoption of and development on the Transputer, Inmos Ltd. developed the Portakit, a portable occam interpreter [4]. In the spirit of the Portakit, we have written an interpreter that executes the Transputer instruction set. This interpreter, which we call the *Transterpreter*, can be easily built and executed on any platform with an ANSI-compliant C compiler. The Transterpreter currently runs on many operating systems and architectures, including Macintosh OS X (PowerPC), Linux (x86, MIPS), Windows (x86), and the LEGO Mindstorms (running the BrickOS [5] on a Renesas H8/300 series CPU).

In this paper, we begin by describing our pedagogic motivations for developing the Transterpreter. This is followed in section 3 by an exploration of related work regarding occam interpreters and the existing tool chains for compiling occam programs. In section 4 we discuss the architectural and design aspects of the Transterpreter that we believe to be interesting, some simple benchmarking results in 5, and close by briefly discussing future directions of our work.

2 Pedagogic Motivation

occam's syntax provides a concise way of expressing ideas about concurrency and parallelism, and we believe it to be an excellent language for teaching these ideas. However,

students tend to perceive things differently: they tend to see occam as having little or no practical application in the world today. Our goal is to begin combating student misconceptions about occam by providing an enjoyable context to which students can relate. We believe occam on the LEGO Mindstorms will be an excellent starting point for students in their study of concurrency and parallelism.

2.1 The LEGO Mindstorms Robotics Kit

The LEGO Mindstorms Robotics Invention System is a commercial product from the LEGO Group that provides an inexpensive, re-configurable platform for exploring robotics [3]. It has three input ports (where touch, light, and other sensors can be attached), three output ports (for motors), and a two way infra-red port. This is used to download programs to the LEGO as well as enable line-of-sight communication between robots.

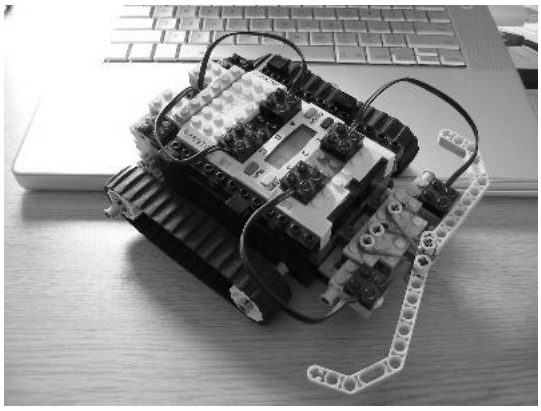

Figure 1: A treaded LEGO robot with two touch sensors

2.2 occam on the LEGO

When programming for the LEGO Mindstorms, students quickly face the difficulty of expressing ideas about concurrency in the procedural or object-oriented languages available. As an example, a robot which is intended to wander around the lab bench while communicating with other robots is difficult for novices to write in languages like C and Java.[1] Furthermore, libraries like JCSP [6] that provide explicit support for occam-style parallelism are too heavyweight (with respect to executable code size and run-time memory requirements, compounded by the already large requirements of the JVM interpreter) and therefore barely usable on machines the size of the LEGO.[2] Programmable robotics kits like the LEGO Mindstorms provide students with a real-world motivation for thinking about parallelism and concurrency; using occam to express those ideas seems like a natural choice in this domain.

We see the Mindstorms as an artefact with which students can begin to observe the abstract world of an executing program in a real and concrete way. A LEGO robot provides a focus for interactions between peers and the instructor, real-world learning situations, and an opportunity for students to have fun in the classroom [7]. To motivate students to think and program concurrently, we believe it is important to begin by creating a challenging and engaging environment for learning to take place in.

3 Related Work

To achieve our goal of running occam programs on the LEGO, we have implemented the Transterpreter, a virtual machine for executing Transputer byte-code. Our work is related to the interpretation of occam, the emulation of the Transputer, and occam compilers for modern architectures.

[1] Personal experience teaching with the LEGO in the classroom.

[2] Conversation with David Barnes regarding the use of JCSP on the LEGO Mindstorms, University of Kent.

3.1 Interpreters and Emulators

The Transterpreter is not the first interpreter of occam that has been written. The Inmos Portakit interpreted the occam1 programming language, and was first released in 1984. It was used to run occam programs before the Transputer was commercially available. The interpreters contained within the Portakit were written in several different languages, including Pascal, BCPL and occam1.

In the early 1990's, Julian Highfield authored a Transputer emulator in C for the Macintosh operating system running on the Motorola 68000 [8]; Highfield's software emulates a Transputer development board that one might have purchased from Inmos for doing Transputer development. The Transterpreter, unlike either of these approaches, is intended to be easily embedded in other applications and ported across multiple architectures.

3.2 Compilers

There are currently two tool chains available for compiling occam programs: the Kent Retargetable occam Compiler (KRoC) and the Southampton Portable occam Compiler (SPOC).

3.2.1 KRoC: The Kent Retargetable occam Compiler

KRoC uses a heavily modified version of the original Inmos occam compiler, and outputs byte-code in an extended version of the Transputer instruction set [9]. This byte-code has been augmented with new instructions to support dynamic memory, 32 levels of priority, mobile channels, and other constructs as introduced in F.R.M. Barnes's doctoral thesis [10]. This code is then further compiled to native code by a separate back-end translator, `tranx86` [11].

Currently, KRoC only produces executable code for the Linux operating system running on the Intel x86 architecture. Although older back-end translators exist (targeting the SPARC and other processors), they are currently out of step with the most recent versions of KRoC and its extensions to the occam programming language [12].

3.2.2 SPOC: Southampton Portable occam Compiler

SPOC [13] predates KRoC, and is a completely separate tool chain; SPOC's implementation has nothing in common with KRoC. Whereas the KRoC tool chain produces native assembly as its output, SPOC is a high-level cross-compiler, converting occam programs into ANSI C. The resulting C programs can then be compiled to binary form on a number of different architectures using a standard ANSI C compiler.

3.2.3 Limitations of KRoC and SPOC

The latest version of KRoC does not currently produce native code for the three most common commercial operating systems in the world today: Windows, Macintosh OS X, and Sun/Solaris.[3] This is largely because of KRoC's reliance on a complex back-end to produce native executables from Transputer byte-code, as well as the need to port the CCSP run-time system to each new target platform [14]. Furthermore, the resulting executable programs must include this run-time, meaning that the size of the smallest executable program that can be produced is 70KB.

[3] As KRoC is open source, it can of course be made to target these in future versions.

While the C produced by SPOC can be compiled to a number of different platforms, it does not support the latest extensions[4] to occam provided by KRoC, such as mobility, and the size of the smallest possible executable on a 32-bit machine is 60KB. By comparison, the smallest Transterpreter executable on a 32-bit machine is 20K, which is three times smaller than the executables generated by either KRoC or SPOC.

3.3 The Transterpreter, a Portable Run-Time

In developing the Transterpreter, we have focused on addressing the limitations of the tools currently available for running and compiling occam programs. While the KRoC front-end is portable, its code generator must be modified for each new platform we might want to support. The Transterpreter is written in strict ANSI C, employing no libraries; as a result, the Transputer byte-code output by KRoC can interpreted on any platform with an ANSI-compliant C compiler. To ease the movement of the Transterpreter from one architecture to another, we have made use of the *autotools* suite to make building the Transterpreter on a new platform a simple, two-step process [15].

The choice to implement an interpreter of Transputer byte-code is in keeping with the current trend in modern programming language implementation. Sun's Java (JVM), Microsoft's C# (CLR), and Perl 6 (Parrot) all follow the same design: a compiler that targets a portable virtual machine. This design is particularly flexible from a language implementation perspective: as KRoC continues to explore new extensions to occam, the Transterpreter can easily be extended to support these new language constructs, which immediately become available on every architecture and operating system to which the virtual machine has been ported.

4 Design and Implementation

The Transterpreter replaces the current back-end of the KRoC tool chain (figure 2). KRoC generates byte-codes which are then transformed further by `tranx86`; using the Transterpreter, those byte-codes are instead processed by the linker (see 4.2) and fed to the Transterpreter, which interprets each instruction, simulating a real Transputer. Around the core interpreter is a portability wrapper that provides operating system dependent hooks for external channels (like `kyb` and `scr`), timers, the loading of byte-code, and graceful error handling.

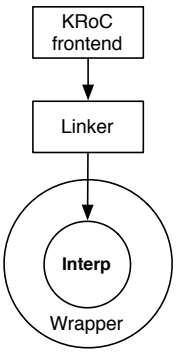

Figure 2: The structure of the Transterpreter

Both the linker and interpreter have been designed with growth and change in mind. The linker is implemented in a micro pass architecture that can be easily be extended to support new transformations of the byte-code. New instructions can be added to the core interpreter, and all operating system specific code has been pushed into a wrapper around the interpreter. When moving between similar (POSIX-compliant) operating systems, minimal changes are required in the wrapper; for more specialised applications, this wrapper may change more significantly.

4.1 One Machine, Two Interpreters

The C programming language can be compiled efficiently (with regards to the size and performance of resulting executable), but programming safely in C is difficult at best. To contend with this reality, we have written not one, but *two* interpreters.

[4] As SPOC is open source, this can also be remedied.

Our first interpreter is written in Scheme [16]. This "interpreted interpreter" gives us a running virtual Transputer in 1200 lines of a safe, high-level language. From a programmer's point of view, Scheme has a simple syntax, and by its nature we are protected from many of the tedious, dangerous aspects of programming—memory management, pointers—faced by developers working in C. Debugging our implementation of the Transputer's scheduler, for example, was made simple by the ability to easily interrupt and inspect everything about the state of the interpreter without resorting to tools like the GNU debugger.

4.1.1 External Channels and the Foreign Function Interface

Our virtual machine is not intended to replace a host OS; as a result, we have focused on making the ANSI C core portable by relying on the operating system and libraries compiled into the wrapper to provide as much external functionality as possible. Both external channels and the foreign function interface are arrays of pointers to procedures; these procedures are intended to be implemented in the wrapper by a developer porting the Transterpreter from one architecture to another [17, 18]. For example, the special input and output channels `kyb` and `scr` currently make use of the standard input and output ports on Linux and OS X via the general external channel mechanism. On the LEGO Mindstorms (running the BrickOS) we attach `scr` to its 5-character LCD; both the UNIX and LEGO solutions require a minimal amount of wrapper code. We can support an arbitrary number of external channels using this mechanism. In the context of the Mindstorms, motors, sensors, infra-red communications, and sound are all easily accessed via this external channel interface. Thinking more broadly about other platforms, we can imagine attaching to network sockets, graphical user interfaces, or a wide variety of hardware devices that are already programmatically accessible via C.

4.2 The Transterpreter Linker: a Stratified Architecture

In designing a linker that would take input from tools like KRoC and produce clean byte-code to execute on the Transterpreter, we stratified our design to accommodate future change. This design extends the UNIX notion of piping together two or more commands to transform data. We have carried this notion to a logical extreme in the implementation of our linker.

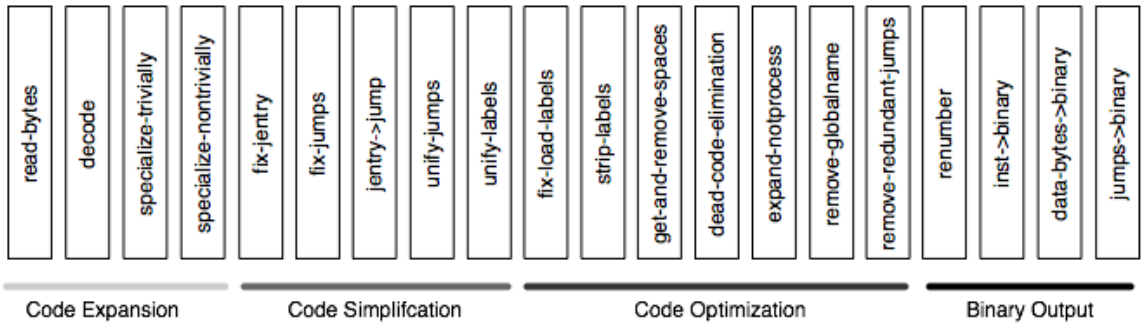

Figure 3: A twenty pass linker, clearly separating out semantic concerns in the linking process

A stratified architecture is appropriate to compilers and compiler-like software. We like to think of compilers and their supporting tools as having as many passes as necessary, where each pass does one thing and one thing only. `tranx86`, the native-code generating back-end of KRoC, has only four passes—input, translation, optimisation, and output. In each pass, `tranx86` does many things to the instruction stream; as a result, dependancies between instructions add significantly to the complexity of the code [11]. By comparison, we isolate these complexities with separate micro passes, gaining conceptual clarity and sacrificing little in run-time efficiency on modern machines.

5 Performance

The Transterpreter was designed to be maintainable and portable; fast execution time was never a primary motivator in the design process. Despite this fact, the Transterpreter, as an interpreter of Transputer byte-code, compares favourably with existing occam implementations. First, we look at the number of source lines of code (SLoC) in KRoC and the Transterpreter as a simple metric related to maintenance. Second, we compare the execution of commstime [19], a common occam benchmark, across KRoC, SPOC, and the Transterpreter.

5.1 Source Lines of Code

Counting the number of lines of code in the linker, the interpreter core, and portability wrapper, there are only 3175 lines of code in the entire Transterpreter project.[5] The equivalent combination of tools in the KRoC tool chain would be the `tranx86` back-end and the CCSP run-time, totalling 29,012 lines of code. Porting the KRoC back-end to a new architecture means porting both of these tools, while porting the Transterpreter only requires modifying or rewriting the wrapper, which is seventy times smaller than the KRoC 1.3 back-end (Table 1).

Table 1: SLoC for occam implementations

Implementation	SLoC
Transterpreter wrapper	416
Transterpreter core	1257
Transterpreter linker	1502
CCSP v1.6	12,480
`tranx86` v0.9	16,532

5.2 Benchmarking

We have two sets of benchmarks: one set generated on an idle Sun v480 with 4GB of RAM and two 900MHz UltraSparc III processors, and one generated on an idle Dell Optiplex GX260 with 512MB of RAM and a 2.4GHz Pentium 4 processor. In all cases, we are comparing different occam compilers and run-times; as a result, the numbers reported are a representative indication of performance, and should not be construed otherwise.

We chose commstime as a simple benchmark that can be run on KRoC, SPOC, and the Transterpreter. It lets us compare two important features of an occam run-time: the time it takes for a context switch, and the time it takes to startup and shutdown a PAR.

5.2.1 SunOS/UltraSparc III

Table 2 shows the time required for context switches and the startup/shutdown time of a PAR in KRoC 1.0 [9], SPOC, and the Transterpreter on a Sun v480. With a sequential delta, SPOC runs 7 times slower than KRoC 1.0. The Transterpreter is 15 times slower than KRoC 1.0, which means it handles a context switch roughly twice as slowly as the SPOC run-time.

The Transterpreter handles context switches much slower than the SPOC runtime; given this, it is surprising to see that the Transterpreter is 10% faster than SPOC on PAR startup and shutdown times. This could perhaps be explained by the difference in structure between the two commstime executables. The Transterpreter is a 1257 line run-time interpreting an

[5] Numbers generated using *sloccount*, `http://www.dwheeler.com/sloccount/`

Table 2: Context switch and PAR startup/shutdown times on a Sun v480

Implementation	Context switch (ns)	Startup/shutdown (ns)
KRoC 1.0	83	13
SPOC 1.3	572	382
Transterpreter	1245	344

array of 1249 bytes of Transputer byte-code; SPOC compiles the same version of commstime into 2898 lines of C (which includes an embedded run-time). Given such differences, anything from the efficiency of the respective schedulers to cache locality could account for this observed performance difference; future work will explore this in greater detail.

5.2.2 Linux/x86

KRoC 1.3 has a context switch time of 114ns on our Linux/x86 test platform, and a PAR startup/shutdown time of 22ns. With a context switch time of 618ns and a PAR cycle time of 200ns, the Transterpreter is, on average, between six and ten times slower than KRoC 1.3.

It is important to note that many new additions to the occam programming language are supported by KRoC 1.3 that are not supported by KRoC 1.0. These new additions include multiple priority levels, mobile channels, and dynamic memory, all of which increase the complexity of the run-time. Additionally, the run-time kernel is no longer hand-crafted assembly; instead, it has been re-written in C [14]. These factors are the most likely explanation for the Transterpreter's relative "improvement" when compared to KRoC 1.3 as opposed to KRoC 1.0.

6 Future Work

We need to achieve unit test coverage in the interpreter, and set up a test harness for running the interpreter on a suite of test cases automatically. The authors acknowledge that this would have ideally driven the implementation of the Transterpreter from the beginning. In terms of growing the interpreter, we will extend the currently supported instruction set to include KRoC's instructions for allocating and using dynamic memory and all its 32 levels of priority.

In addition to supporting additional extensions to occam, we look forward to exploring the use of our run-time on other platforms. Mobile phones, handheld devices, and other ubiquitous computing devices are natural hosts for our portable occam run-time.

Acknowledgments

We would like to thank Fred Barnes for his excellent work on the Kent Retargetable occam Compiler, his willingness to answer questions on all things occam, as well as "real-time" software development to support our needs in compiling occam for 16-bit architectures using the KRoC tool chain. We would also like to thank Dr. Andy King for his comments on an early draft of this paper, our anonymous reviewers, and David Wood, who first suggested "Transterpreter" as a historically appropriate name for our project. Lastly, thanks to our colleagues and friends who listened (suffered) endlessly as we discussed and debated design and implementation issues surrounding the Transterpreter. We claim all remaining errors as our own.

References

[1] Inmos Limited. *occam2 Reference Manual*. Prentice Hall, 1984. ISBN: 0-13-629312-3.

[2] Denis A. Nicole, Sam Ellis, and Simon Hancock. occam for reliable embedded systems: lightweight runtime and model checking. In Jan F. Broenink and Gerald H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 167–172, 2003.

[3] The LEGO Mindstorms homepage, 2004. http://www.legomindstorms.com/.

[4] Inmos Limited. *The occam Portakit Implementors Guide.* Bristol, November 1984.

[5] Markus L. Noga. The legos operating system, Oct 1999. http://brickos.sourceforge.net/.

[6] Peter H. Welch, Gerald H. Hilderink, and Nan C. Schaller. Using Java for Parallel Computing - JCSP versus CTJ. In Peter H. Welch and Andre W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 205–226, 2000.

[7] Matthew C. Jadud. Teamstorms as a theory of instruction. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference*, volume 1, 2000.

[8] Julian C. Highfield. A transputer emulator. http://spirit.lboro.ac.uk/emulator.html.

[9] D.C. Wood and P.H. Welch. The Kent Retargetable occam Compiler. In Brian O'Neill, editor, *Parallel Processing Developments, Proceedings of WoTUG 19*, volume 47 of *Concurrent Systems Engineering*, pages 143–166. World occam and Transputer User Group, IOS Press, Netherlands, March 1996. ISBN: 90-5199-261-0.

[10] Frederick R.M. Barnes. *Dynamics and Pragmatics for High Performance Concurrency*. PhD thesis, University of Kent, June 2003.

[11] F.R.M. Barnes. tranx86 – an Optimising ETC to IA32 Translator. In Alan Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 265–282, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN: 1-58603-202-X.

[12] Ruth Ivimey-Cook. Legacy of the Transputer. In Barry M. Cook, editor, *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pages 197–211, 1999.

[13] M. Debbage, M. Hill, S. Wykes, and D. Nicole. Southampton's portable occam compiler (SPOC). In Roger Miles and Alan Chalmers, editors, *Proceedings of WoTUG-17: Progress in Transputer and occam Research*, volume 38 of *Transputer and occam Engineering*, pages 40–55, Amsterdam, April 1994. IOS Press.

[14] J.Moores. CCSP – a Portable CSP-based Run-time System Supporting C and occam. In B.M.Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering series*, pages 147–168, Amsterdam, the Netherlands, April 1999. WoTUG, IOS Press.

[15] Tom Tromey Gary V. Vaughan, Ben Elliston and Ian Lance Taylor. *GNU autoconf, automake, and libtool*. New Riders Publishing, October 2000.

[16] R. Kelsey, W. Clinger, and J. Rees. The revised5 report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1), Sep 1998.

[17] F.R.M. Barnes. User Defined Channels in occam. Technical report, Computing Laboratory, University of Kent at Canterbury, April 2002.

[18] David C. Wood. KRoC – Calling C Functions from occam. Technical report, Computing Laboratory, University of Kent at Canterbury, August 1998.

[19] Peter H. Welch and Fred Barnes. Prioritised Dynamic Communicating Processes - Part I. In James Pascoe, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, pages 321–352, 2002.