Concurrency: The Next Generation

Damian Dimmich, Christian Jacobsen and Matthew Jadud
Computing Laboratory
University of Kent
Canterbury, CT2 7NZ
{djd20,clj3,mcj4}@kent.ac.uk

1. INTRODUCTION

Concurrency is needed everywhere. In emerging platforms such as wireless sensor networks, where a large number of tiny physically separate computational devices must be able to act as a coherent unit. In the multi-core processors, which are being introduced in consumer devices[26, 1, 24] in response to the ever growing demand for performance, due to the physical limitations of current processor architectures[13, 27, 25, 29]. In high-performance scientific computing, where concurrency already plays a large role, and in teaching, to ensure that future software engineers have the skills required to use these new technologies. The need for safe and robust concurrency in all these areas is fundamental.

The Transterpreter[20] is a small portable runtime for occam-pi [35], which runs on a wide range of devices, from small sensor nodes to high-performance clusters, diminishing the boundaries between small and large platforms. occam-pi provides a consistent set of rich, robust and mathematically backed concurrency primitives which scale well, not only with program complexity, but also with device size. As concurrent systems become commonplace, the need for languages supporting these notions of concurrency is growing rapidly.

2. BACKGROUND

Multi-core processors have finally started permeating desktop computing, and concurrency must follow suit. Many of today's applications are not written to take advantage of the increased processing power provided by multiple processors, having previously relied on the fast paced increase in processor clock frequency. It is important that new applications are written with concurrency in mind, as it will be the number of processors, as opposed to clock frequency, which is going to multiply in the future[10, 7].

Many languages are not well suited for taking advantage of concurrency, including C and C++, where concurrency is implemented as a low level library. These libraries require developers to invest a large amount of energy to manually ensure the correctness of their code. While this leaves great scope for error, the possibility of more subtle interactions between the compiler and threaded code as described in [6], are perhaps of greater concern. Even modern languages such as Java, where concurrency has been engineered in, do not necessary offer great deal of assistance to the developer. The provided primitives may work at a level that is too low for practical use, may not scale with complexity, and may in fact provide a broken implementation[37, 16].

High-performance computing is experiencing a shift towards the development of application components which are not performance critical, such as the overall control logic, using high level languages. Instead of developing an entire application in relatively low level languages such as C or Fortran, a scripting language such as Python is used, gluing new or existing high-performance code into an application. This can significantly improve development time, while still allowing the application to leverage highly optimised native code[4, 5]. Making such a shift does not necessarily have adverse affects on performance. The use of Java and Common Language Infrastructure (CLI/.NET) hosted languages is being actively explored within the high-performance community[34, 31].

Concurrency is not only an important concept in the context of performance, but can also be used effectively as a programming model, providing a compositional and modular abstraction, as is the case with CSP[18]. These properties can make concurrent languages accessible to applications where concurrency is a natural part of the problem specification. Wireless sensor networks, robotics and many other areas, such as operating systems, fall into this category[22, 21, 3].

The introduction of concurrency into languages with referential transparency is a well explored area. It can be relatively easy to automatically parallelise such languages by, exploiting the implicitly available concurrency. In practice, annotations, compile and runtime measures must be used to ensure reasonable performance, and to constrain the granularity of computation. Several concurrent implementations of popular languages such as Haskell and Prolog exist[23, 17, 15]. Erlang, which has firm roots in functional programming, but specifically targets distributed applications, is an example of a language designed with concurrency as primary motivator. New languages such as Fortress[2] also use the notion of side effect free computations to enable concurrent execution.

In order for concurrency to be safe, scalable and efficient the the language, compiler and runtime must be able to assist the programmer in detecting and avoiding dangerous concurrency practices, as noted by Tony Hoare in [19]. Therefore, concurrency should be major design consideration at all stages of a languages development, and would do well to have a sound formal and mathematical foundation. The language and runtime should be small and portable, to facilitate scaling across a wide range of devices, and the implementation needs to be able to efficiently support the levels of concurrency offered by the language. In order to meet these demands we have developed the Transterpreter virtual machine, an execution environment for the occam-pi language.

3. THE TRANSTERPRETER

The Transterpreter is a small and portable runtime for occam-pi, an explicitly concurrent programming language with roots in CSP[18] and the Pi-Calculus[28]. The occam-pi language and its tools, strive to provide the developer with a practical environment for the development of concurrent programs, in the spirit of Tony Hoare's statements in [19]. Rich synchronisation primitives and parallel abstraction provides support for extremely fine grained concurrency, enabling an occam-pi program to have tens of thousands of interacting processes running concurrently on a single machine. This machine may well be part of a larger distributed occam-pi program[30].

To support such large scale concurrency, runtimes for occam-pi must be able to juggle a large number of processes with an extremely low overhead. Operating system processes and threads can not be used directly, as they are generally used to manage in the order of tens of processes, and therefore have overheads which are too high to support the desired levels of concurrency. A fast runtime, written explicitly to provide low overhead scheduling and synchronisation for occam-pi processes, is used instead.

Traditionally occam-pi has been compiled directly into native code, through platform specific assembly, and linked together with a runtime, using the KRoC toolchain[36]. The runtime has either been written entirely in assembly for a specific platform, or using C, with platform dependent portions to enable fast execution. The Transterpreter parts with this tradition, and instead uses only portable ANSI C for its implementation[20].

The Transterpreter is a bytecode interpreter, and therefore sacrifices some performance in return for portability. Furthermore, the Transterpreter has not been prematurely optimised, its initial focus instead being correctness and viability as a platform for future research. Even when considering these constraints, the performance of the Transterpreter is acceptable for a broad range of applications.

One particular measure of performance of an occam-pi runtime is the time it takes to switch from one process to another, the context switch time, an important measure when running millions of processes. Benchmarks run on an unloaded P4 3.2GHz system has shown that the Transterpreter has a context switch time of approximately 430 nanoseconds. This is factor of 30 away from the context switch time provided by a natively compiled KRoC program, which can perform a switch in approximately 14 nanoseconds.

Both literature on interpretation[12], and initial JIT experiments suggest that the Transterpreter can improve its performance to a state where it is approximately a factor of 2 – 10 away from that provided by native code. Further experiments, in which the Transterpreter scheduler has been used as a runtime for natively compiled occam-pi code, has shown that context switch times of approximately 8 nanoseconds are possible. The native code compilation is performed through translation of bytecode to C, and subsequent compilation by the GCC compiler, using no platform specific code.

4. THE NEXT GENERATION

The Transterpreter has become a mature runtime for the occam-pi programming language. It is more than this however, it is a viable platform, enabling future research and educational activities surrounding concurrency.

The Transterpreter has already found use in teaching[21]. It has been used on the LEGO Mindstorms robotics kit in the extra curricular Cool Stuff in Computer Science course, to introduce concurrency to audiences with little prior programming experience. The use of robotics and concurrency in a teaching environment have been employed in the Concurrency Design and Practice course at the University of Kent, and the Extreme Multiprogramming course at the University of Copenhagen. Both have used the RoboDeb VMWare Player virtual machine[38], which provides a complete Linux environment, preconfigured for writing occam-pi programs using the Player/Stage library and robotics simulator[14].

A large amount of educational material was produced to support these courses. This material has been made generally available, but in order to further the Transterpreter's potential role in education, the provision of additional self-supporting teaching materials is planned. Efforts are also underway to improve the current support for the LEGO Mindstorms, and to provide support for the soon to be released LEGO Mindstorms NXT.

The Transterpreter is currently in active use in several research areas. In the area of wireless sensor networks, it is used to explore novel approaches to providing concurrency on small, interconnected computational devices, which currently seem to only offer impoverished concurrency support[22]. Originally motivated by the educational aspects, the use of occam-pi in robotics control is being examined in terms of Brooks subsumption architecture[8]. The use of the Transterpreter as a sandboxed environment on the Minimum Intrusion Grid[33] is being actively evaluated, along with check-pointing support and its use in fault tolerance. This research has provided the inspiration for the Scientific Bytecode project, which aims to provide a virtual execution environment for high-performance computing. This environment will provide explicit support for accessing high-performance, native library procedures, while allowing control and distribution logic to be written using higher level abstractions.

Concurrent languages, such as occam-pi, must be able to take advantage of the increasing levels of parallelism found in computer architectures. SMP support is currently being added to the Transterpreter, building on existing research in this area[32]. Research into the use of occam-pi on emerging heterogeneous processor such as the Cell Broadband Engine™[11] is also being undertaken. Such architectures may benefit from programming models which are unified across the entire device. This research may also apply to more conventional heterogeneous systems, such as the common combination of general purpose desktop CPUs with high-performance specialised graphics GPUs. It may even be possible to leverage this research in high-performance computing, where mixed FPGA/CPU systems are being introduced[9]. Of particular research interest is the topic of code and data distribution across these computational spaces.

Various strategies are being investigated to improve the performance of the virtual machine such as portable native code generation for specialised processors such as the Cell Broadband Engine™. Just In Time compilation, native code compilation through C and GCC, and direct GCC backending are being evaluated. The feasibility of 8-bit device support for the Transterpreter is also under investigation.

The Transterpreter is a platform for research in its own right. It is small, modular, well documented and written to be easy to understand and extend. It is written to be a vehicle for the implementation of future ideas within the scope of occam-pi. Currently the Transterpreter uses the existing KRoC compiler for generating bytecode, while small exploratory occam-pi compilers targeting the the Transterpreter natively, have already been demonstrated. A further open research area focuses on creating a novel micropass architected compiler, providing a language research framework which is as flexible and extensible as the Transterpreter itself. This is however no small undertaking, and in the first instance, a small prototype compiler, focused at providing support for specific research avenues is planned. The fundamental goal is to replace the aging KRoC tools with a toolchain that can be easily used, understood and extended by researchers, developers and students, while being used in concurrency and language research, in teaching, and for solving practical problems.

The future work presented is going to form the bases for grant applications, and publications to reputable conferences and journals.

www.transterpreter.org

5. REFERENCES

[1] Advanced Micro Devices, Inc. Multi-core processors— the next evolution in computing. White paper, 2005.

[2] Allen, Chase, Luchangco, Maessen, Ryu, Steele, and Tobin-Hochstadt. The fortress language specification, version 0.903. Technical report, Sun Microsystems, 2006.

[3] F. Barnes, C. Jacobsen, and B. Vinter. RMoX: A raw-metal occam experiment. In *Communicating Process Architectures 2003*, volume 61 of *Concurrent Systems Engineering Series*, pages 269–288. IOS Press, September 2003.

[4] D. Beazley and P. Lomdahl. Feeding a large scale physics application to Python. *Proceedings of the 6 th International Python Conference*, October 1997.

[5] D. M. Beazley and P. S. Lomdahl. Lightweight computational steering of very large scale molecular dynamics simulations. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 50. IEEE Computer Society, 1996.

[6] H. Boehm. Threads cannot be implemented as a library. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 261–268. ACM Press, 2005.

[7] S. Y. Borkar, P. Dubey, K. C. Kahn, D. J. Kuck, H. Mulder, S. S. Pawlowski, and J. R. Rattner. Platform 2015: Intel ® processor and platform evolution for the next decade. Technical report, Intel Corporation, 2005.

[8] R. A. Brooks. A robust layered control syste for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.

[9] Cray, Inc. Cray xd1 supercomputer for reconfigurable computing, 2006.

[10] M. Creeger. Multicore CPUs for the masses. *Queue*, 3(7):64–ff, 2005.

[11] D. Pham, et al. The Design and Implementation of a First-Generation CELL Processor. pages 184–185. IEEE International Solid-State Circuits Conference, ISSCC 2005, February 2005.

[12] M. A. Ertl, D. Gregg, A. Krall, and B. Paysan. Vmgen — a generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.

[13] J. Frenkil. A multi-level approach to low-power ic design. *IEEE Spectr.*, 35(2):54–60, 1998.

[14] B. Gerkey, R. Vaughan, and A. Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the International Conference on Advanced Robotics (ICAR 2003), Coimbra, Portugal, June 30 - July 3, 2003*, pages 317–323, 2003.

[15] G. Gupta, E. Pontelli, K. A. Ali, M. Carlsson, and M. V. Hermenegildo. Parallel execution of Prolog programs: a survey. *ACM Trans. Program. Lang. Syst.*, 2001.

[16] P. B. Hansen. Java's insecure parallelism. *SIGPLAN Not.*, 34(4):38–45, 1999.

[17] T. Harris, S. Marlow, and S. P. Jones. Haskell on a shared-memory multiprocessor. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 49–61. ACM Press, 2005.

[18] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[19] C. A. R. Hoare. Hints on programming language design. *State of the Art Report 20: Computer Systems Reliability*, pages 505–534, 1974.

[20] C. L. Jacobsen and M. C. Jadud. The Transterpreter: A Transputer Interpreter. In *Communicating Process Architectures 2004*, pages 99–107, 2004.

[21] C. L. Jacobsen and M. C. Jadud. Towards concrete concurrency: occam-pi on the lego mindstorms. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*. ACM Press, 2005.

[22] M. C. Jadud, C. L. Jacobsen, and D. J. Dimmich. Concurrency on and off the sensor network node. *SEUC 2006 workshop*, 2006.

[23] S. P. Jones, A. Gordon, and S. Finne. Concurrent haskell. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–308. ACM Press, 1996.

[24] R. Kalla, B. Sinharoy, and J. Tendler. IBM Power5 chip: a dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, March 2004.

[25] L. B. Kish. End of Moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, pages 144–149, September 2002.

[26] G. Koch. Discovering multi-core: Extending the benefits of Moore's law. July 2005.

[27] D. Mallik, K. Radhakrishnan, J. He, C.-P. Chiu, T. Kamgaing, D. Searls, and J. D. Jackson. Advanced package technologies for high-performance systems. *Intel Technology Journal*, 9(4):259–272, November 2005.

[28] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999. ISBN-10: 0521658691, ISBN-13: 9780521658690.

[29] T. N. Mudge. Power: A first class design constraint for future architecture and automation. In *HiPC '00: Proceedings of the 7th International Conference on High Performance Computing*, pages 215–224. Springer-Verlag, 2000.

[30] M. Schweigler. Adding Mobility to Networked Channel-Types. In *Communicating Process Architectures 2004*, pages 107–126, 2004.

[31] L. A. Smith, J. M. Bull, and J. Obdrz&#225;lek. A parallel java grande benchmark suite. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 8–8. ACM Press, 2001.

[32] K. Vella. *Seamless Parallel Computing on Heterogeneous Networks of Multiprocessor Workstations*. PhD thesis, University of Kent at Canterbury, December 1998.

[33] B. Vinter. The Architecture of the Minimum intrusion Grid (MiG). In *Communicating Process Architectures 2005*, 2005.

[34] W. Vogels. Hpc.net - are cli-based virtual machines suitable for high performance computing? In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 36. IEEE Computer Society, 2003.

[35] P. Welch and F. Barnes. Communicating mobile processes: introducing occam-pi. In *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, Apr. 2005.

[36] P. Welch and D. Wood. The Kent Retargetable occam Compiler. In *Parallel Processing Developments, Proceedings of WoTUG 19*, pages 143–166, March 1996.

[37] P. H. Welch. Java Threads in the light of occam/CSP. In *Proceedings of WoTUG-21: Architectures, Languages and Patterns for Parallel and Distributed Applications*, pages 259–284, 1998.

[38] http://robodeb.transterpreter.org/.