Adapting processor grain via reconfiguration

Jecel Mattos de Assumpção Jr[1], Merik Voswinkel[2], and Eduardo Marques[1]

[1] Universidade de São Paulo
Departamento de Sistemas de Computação
São Carlos, Brasil
{jecel,emarques}@icmc.usp.br
[2] HH Research Institute
Wehe-Den Hoorn,The Netherlands
merik@morphle.org

Abstract. Squeak is an open source Smalltalk-80 implementation created to implement high level code that is used as glue between optimized, low level "plug-ins" written in C or a restricted subset of Smalltalk and translated to C. SiliconSqueak is a hardware implementation with course grained processors for the high level code and fine grained "ALU Matrix" co-processors for the plug-ins. When implemented in an FPGA, a given area can either be used for a co-processor or for two more high level cores. The ideal mix varies at runtime as applications go through different phases, so the solution presented in this paper is to reconfigure the system as needed.

Keywords: Reconfiguration, Heterogeneous cores, Smalltalk

1 Introduction

One of the defining features of a programming language is its type system. In a strongly typed language, like Haskell, there is a strict control of which kinds of objects can be used with a given operation while languages like Forth or assembly are untyped and this control is up to the programmer. Many popular languages, such as those derived from C or Pascal have a relatively strong type system with features like type casts and untagged unions to weaken the system under programmer control. While some programmers use the terms "strong typing" and "weak typing" for languages that associate type information with source text variables and runtime objects respectively, we will use the more traditional terms "static typing" and "dynamic typing" in order not to mix what are independent concepts.

Statically typed languages can get good results from relatively simple compilers while dynamically typed languages have traditionally been interpreted, which resulted in a significant performance gap. Since type declarations can be an obstacle to exploring different design options, a traditional software development method has been to write the application initially in a dynamic language (Lisp, for example) and when it is stable to completely rewrite it in a static

language (like Fortran). This is even the case for embedded applications where the initial algorithms might be developed in something like Matlab running on a desktop machine and the final application is a rewrite in C to run on the target machine.

This popular separation of dynamic languages for prototyping and static languages for production is being challenged by two trends: faster hardware via "Moore's Law" can make the performance of even interpreters acceptable for many applications and sophisticated compilation technologies developed for dynamic languages in the 1990s[3] are being increasingly adopted. Section 2 explains how Squeak Smalltalk[4] deals with performance issues and how this reflects on the design of the SiliconSqueak hardware. Section 3 is about the use of runtime reconfiguration of a FPGA implementation of this hardware to match the processing granularity as the application requirements change. Projects that have some features in common with the one described here are mentioned in section 4 and finally section 5 describe the next steps for this project and the results that have been obtained so far.

2 Squeak and SiliconSqueak

One problem with advanced compilation systems is that it takes significant effort to port them to different platforms and running on as many machines as possible was a major goal of the Squeak project created at Apple in 1996 [4]. It was decided that a good interpreter would meet the needs of the project given modern hardware and the fact that in multimedia applications more time is spent in library functions like codecs than inside the language itself. Java was considered as an option for the base language so that the small team could focus on the application while external groups took care of the platform but the development tools were considered too primitive compared with what Smalltalk-80 had nearly 20 years earlier back at Xerox PARC. Since Apple had a very liberal license for those tools from Xerox, they were selected as the starting point for an open source project with the idea that if development were easy enough, external groups could take care of the porting to machines besides the Mac.

Smalltalk-80 is implemented as a virtual machine which has to be simulated on different computers either with dynamic compilation or with an interpreter. Traditional interpreters were implemented in languages such as C, but the original book explaining the language[2] included a complete reference design written in a restricted form of Smalltalk without any object creation other than Integers and with no polymorphism. This made it easy for a programmer to rewrite in Pascal, C or even assembly language but it also made it possible to create a simple tool to translate it (the Squeak group called this Smalltalk subset "Slang" though there are a few actual programming languages with that name) automatically to C. Though Slang doesn't use all of Smalltalk-80's features, it can run on a normal Smalltalk implementation and make use of all the advanced development tools. Once the functionality has been fully verified, the code can be translated to C and then compiled for higher performance. This can be done not

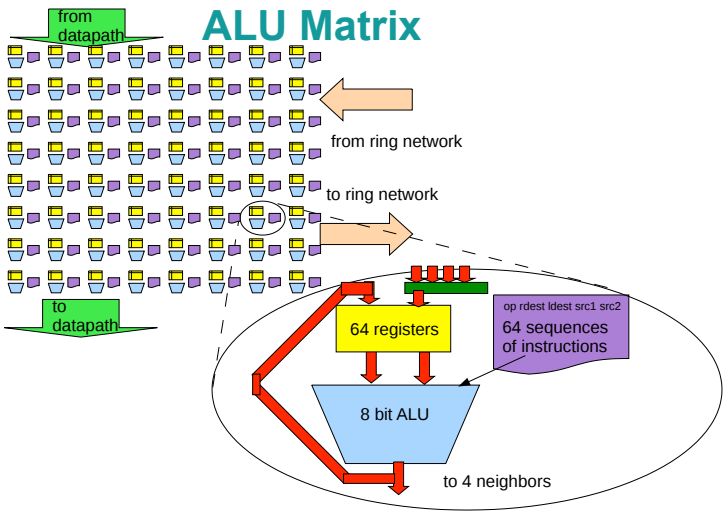

Fig. 1. 64 ALU Matrix

only for the main interpreter, but also for "plug-ins" like the codecs mentioned earlier. Or existing libraries written in other languages can be used, whichever is more convenient.

As planned, Squeak was ported by independent individuals to all the major desktop platforms within weeks of the initial release and to dozens of less known machines over the next couple of years. It is easy to make Squeak run on any soft core with a C compiler, such as the Nios II or the Leon3, to have it available in a FPGA-based reconfigurable system, but a dedicated SiliconSqueak core can be a more efficient option. The goal is to eliminate as much overhead as possible from the interpreter while at the same time including features that make it attractive as a target for adaptive compilation (a part of the project which is outside the scope of this paper, but note that other dynamic compilers have been developed for Squeak in the past and there is currently a very active project called Cog).

Most modern FPGAs are large enough that two or more SiliconSqueak cores can be used at the same time to implement course grained parallelism. This can speed up many kinds of applications, but the plug-ins written in Slang and in C don't benefit from this. So an optional co-processor, called the "ALU Matrix" and shown in figure 1, was developed specifically for that part of the code. The example shows an 8 by 8 matrix of 8 bit ALUs, each with 64 8 bit registers, but each of these parameters can be changed with no impact on the main SiliconSqueak core (though code for the co-processor would have to be recompiled). Besides its local registers, each ALU has its own program memory so that when the main core asks the ALU Matrix to execute sequence 5, for example, the exact operation can be different for each ALU making this fine grained architecture a mix of SIMD and MIMD features. It also combines computation

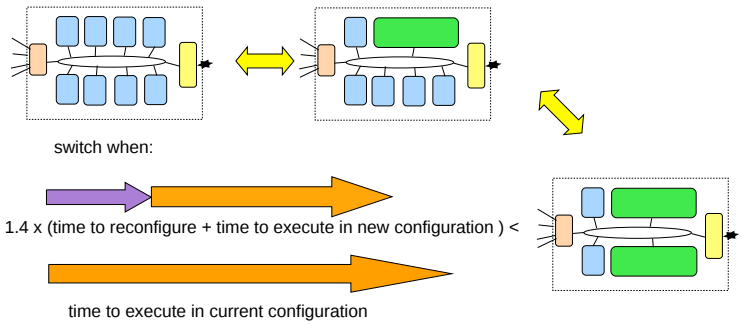

Fig. 2. FPGA reconfiguration at work

and communication in that the instruction names two destination registers: one for the output of this ALU and one for the output of a selected neighbor.

3 Scheduling and Reconfiguration

Blue blocks in figure 2 represent generic SiliconSqueak cores, green depicts a 64 ALU matrix including the adjacent SiliconSqueak core to control it, the orange block is a 4 x 8 Gb/s networking unit to interconnect with neighboring input/output unit or FPGAs and ASIC processors in a mesh with point to point links, yellow is the memory controller properly balanced with the optimal number of cores to avoid bottlenecks to dynamic random access memory (DRAM). Interconnecting of cores, matrices and units is accomplished by a ring network.

With an initial configuration as in the left of figure 2, switching to the option in the middle would replace some of the existing cores both in terms of FPGA area and as an element in the ring networks. If that particular processor was exclusively executing code that will now be done by hardware, there will be a gain in performance. If, on the other hand, it was also executing unrelated code that must now be moved to the other cores then there will be a performance loss no matter how much faster the hardware is than the optimized code. The scheduler should group related code under heavy loads to make it simpler to detect the situation where a software block has one or more cores dedicated to it and so is a candidate for a hardware replacement.

Given that an FPGA that is being reconfigured does not execute anything, the scheduler should deal with time frames N times longer than this inactive period. Besides the reconfiguration time itself, there is the time needed to save all current state to external memory and then the time to restore it (adapting to the new configuration). Since a single core with an ALU Matrix takes up the same FPGA resources as three simple cores, any code which doesn't make use of the co-processor will run roughly three times slower. Any code that does take advantage of the ALU Matrix (code generated by the new compiler), on the other hand, will run X times faster.

$$N > 1.4 \times (1 + (1 - \alpha) \times 3N + \alpha \times \frac{N}{X}) \tag{1}$$

$$\alpha > \frac{(\frac{N}{1.4} - 3N - 1)}{(\frac{N}{X} - 3N)} \tag{2}$$

Where α is the percent of time that code that could use the co-processor takes on the configuration with three simple cores. To avoid needlessly switching back and forth between configurations, a factor of 1.4 adds some hysteresis to the system. Equation 1 shows under what conditions it is profitable to replace three simple cores with a single one having a co-processor. Equation 2 solves for α given X (notice that $\frac{N}{X} - 3N < 0$ given that $X > 1$). So if $X = 6$ (code becomes six times faster with the ALU Matrix) and $N = 10$ (the scheduling time frame is ten times the reconfiguration time) then $\alpha > 84\%$.

$$N > 1.4 \times (1 + (1 - \beta) \times \frac{N}{3} + \beta \times NX) \tag{3}$$

$$\beta < \frac{(\frac{N}{1.4} - \frac{N}{3} - 1)}{(NX - \frac{N}{3})} \tag{4}$$

In equation 3 we have the condition where it is a good idea to replace a SiliconSqueak core including an ALU Matrix with three simple cores. Here β is the percent of the time in which code that uses the ALU Matrix executes in the original configuration (this is different, but related to, α). Given the same $X = 6$ and $N = 10$, then $\beta < 5\%$.

4 Related Works

Designing processors optimized for Java, such as JOP[6], are becoming more popular, as are extensions to conventional processors like the two Jazelle options for the ARM. The Lisp Machines of the 1970s to 1990s are the best known language specific architectures, but there were many designs optimized for Smalltalk including the first machine based on FPGAs, the Manchester Mushroom[8].

Among the many systems that use runtime reconfiguration to reallocate FPGA resources, the BORPH operating system[7] is interesting in its analogy to memory allocation in Unix systems. The Virtex 4 FPGAs used in the initial experiements in this project allow partial reconfiguration which has the advantage of not disrupting the part of the design that stays the same from one configuration to the next. This isn't being used yet in part because of the complexity of the tools required and in part because the plan is to build machines with multiple FPGAs which allows partial configuration at the system level.

Reconfigurable co-processors were used in projects like ADRES[1] or in NEC's IMAPCAR2 chip[5], where groups of four can be used as either a VLIW MIMD node or a SIMD machine.

5 Initial Results and Future Works

The plan is to have a wider range of benchmarks and to use more modern FPGAs.

References

1. Bouwens, F.J., Berekovic, M., Kanstein, A., Gaydadjiev, G.N.: Architectural Exploration of the {ADRES} Coarse-Grained Reconfigurable Array. In: Proceedings of International Workshop on Applied Reconfigurable Computing. pp. 1–13 (2007)
2. Goldberg, A., Robson, D.: Smalltalk-80: the language and its implementation. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1983)
3. Hölzle, U.: Adaptive optimization for Self: reconciling high performance with exploratory programming. Ph.D. thesis, Stanford University (1994)
4. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: The story of Squeak, A practical Smalltalk written in itself. In: Proceedings OOPSLA '97, ACM SIGPLAN Notices. pp. 318–326. ACM Press (1997)
5. Kyo, S., Koga, T., Hanno, L., Nomoto, S., Okazaki, S.: A low-cost mixed-mode parallel processor architecture for embedded systems. In: Proceedings of the 21st annual international conference on Supercomputing - ICS '07. p. 253. ACM Press, New York, New York, USA (2007), http://portal.acm.org/citation.cfm?doid=1274971.1275006
6. Schoeberl, M.: A {Java} Processor Architecture for Embedded Real-Time Systems. Journal of Systems Architecture doi:10.101 (2007)
7. So, H.K.H.: BORPH: An Operating System for FPGA-Based Reconfigurable Computers. Ph.D. thesis, Engineering – Electrical and Computer Sciences, University of California, Berkeley (2007)
8. Williams, I.W.: Using FPGAs to Prototype New Computer Architectures. In: Moore, W.R., Luk, W. (eds.) FPGAs, chap. 6.8, pp. 373–382. Abingdon EE & CS Books (1991)