

Adaptive Compilation for Reconfigurable Computers

Jecel Mattos de
Assumpção Jr
Universidade de São Paulo
Departamento de Sistemas de
Computação
São Carlos, Brasil
jecel@icmc.usp.br

Merik Voswinkel
HH Research Institute
Wehe-Den Hoorn, The
Netherlands
merik@morphle.org

Eduardo Marques
Universidade de São Paulo
Departamento de Sistemas de
Computação
São Carlos, Brasil
emarques@icmc.usp.br

ABSTRACT

Dynamic programming languages have become increasingly popular and adaptive compilation, which uses runtime measurements to generate improved code, is a key technology for high performance implementations of such languages. While it has been used in servers and desktops, adaptive compilation has not been as successful in the low end and embedded systems and even less so in the high end such as supercomputers. SiliconSqueak is a parallel, reconfigurable architecture optimized for adaptive compilation which can address both computing extremes. This manycore system includes a mix of basic and extended processors, where these extensions are configurable accelerators. In FPGA implementations the ratio of these changes at runtime.

Categories and Subject Descriptors

B.1.5 [Register Transfer-Level Implementation]: Design; C.1.2 [Processor Architectures]: Multiple Data Stream Architectures; C.5.3 [Computer System Implementation]: Microcomputers-microprocessors, portable devices; D.3.2 [Programming Languages]: Language Classification—*Smalltalk*

General Terms

Design, Languages

Keywords

object-oriented hardware, adaptive compilation, reconfigurable hardware, parallelism, virtual machine, object heap, object table, Smalltalk, Squeak

1. INTRODUCTION

Adaptive compilation is an advanced implementation technology developed in the mid 1990s which is being increasingly adopted by high performance object-oriented programming languages. Its high resource requirements have so far kept it out of low cost or embedded applications. This will become less of a problem in the future thanks to Moore's

Law, but an alternative would be to take adaptive compilation into account when designing a new processor. This was the approach used for SiliconSqueak, which is a hardware implementation of the virtual machine for the Squeak [7] open source Smalltalk-80.

Though future versions of SiliconSqueak might be implemented using custom integrated circuit technology, the current focus is on its use as a "soft core" for systems based on reconfigurable circuits (FPGAs - Field Programmable Gate Arrays). The design is sufficiently compact that a number of such cores can fit into all but the very smallest FPGAs now commercially available (which are much larger than the typical FPGAs of a decade ago as this market can make the best use of Moore's Law). Increased performance through parallelism and the message passing nature of Smalltalk are a natural fit [2, 8] and can combine with adaptive compilation in a synergetic way [3].

For reconfigurable computers, another option for increasing the performance is to design the hardware specifically for the application which will run on it. In the case of a statically compiled language, such as C, even relatively primitive tools can be used to partition the system into hardware and software blocks. The hardware blocks can be implemented in some hardware description language, such as Verilog or VHDL, and the configuration bits generated while the software blocks are compiled into executable binary code. This manual development style is not at all aligned with the spirit of Smalltalk which is about "late binding", or delaying decisions as much as possible. The proposal in this paper is to delay the software/hardware partition until runtime and to base it on exactly the same information used by adaptive compilation. When the characteristics of an application changes over time, there is no single best configuration for the hardware and improved performance can be achieved by continuously modifying it.

This paper is organized as follows: Section 2 gives a brief history of adaptive compilation and its role in high performance software implementations of Smalltalk and Section 3 describes the SiliconSqueak processor and how the hardware design interacts with adaptive compilation. After that, Section 4 is a short introduction to dynamic reconfiguration as well as partial reconfiguration of FPGA based systems and Section 6 is a proposal to extend adaptive compilation to make use of dynamic reconfiguration. Finally, Section 7 describes some related work and Section 8 mentions some

conclusions.

2. EVOLUTION OF ADAPTIVE COMPILATION

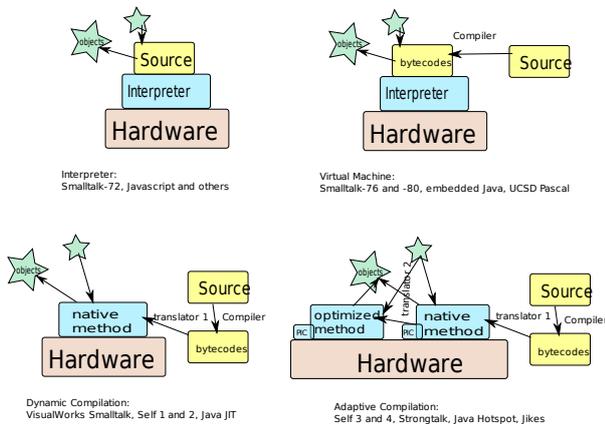


Figure 1: Implementation Technologies

Fig. 1 shows a brief history of the implementation of object-oriented programming languages related to Smalltalk. The simplest option is a pure interpreter, shown in the upper left corner of the figure. The only code directly executed by the hardware is the interpreter itself, which deals both with the source code and the various runtime objects to generate the expected result. Though interpreters are normally classified as translators, it would be more accurate to consider them to be simulators instead.

While the first Smalltalk implementations (called Smalltalk-72 and -74 as they were created in 1972 and 1974, respectively) made good use of the fact that they were pure interpreters by allowing the syntax of the language to be defined dynamically, the performance was not very good and the extra flexibility proved to be a problem in the long run as people couldn't understand each other's code. For the 1976 version of the language, Dan Ingalls condensed the most interesting results of the first four years of wild exploration into a simple and fixed syntax which could be compiled for an imaginary, or virtual, machine. This is shown in the upper right corner of Fig. 1. While the instructions (called bytecodes) for this virtual machine were interpreted on each actual computer on which it ran, the low level nature of these instructions (no need for lexical analysis, parsing syntax, and so on) resulted in a significantly improved performance.

A very interesting feature of the computers at the Xerox Palo Alto Research Center (PARC) where Smalltalk was being developed was that they were highly reconfigurable. The hardware itself was fixed but as generic as possible and the actual behavior was determined by the microcode that was loaded into the machine. Unfortunately, almost no commercial computers had that kind of flexibility. The introduction of dynamic compilation in [4] was intended to allow stock hardware to efficiently implement Smalltalk-80. This is illustrated in the lower left corner of the figure. The key idea is to have multiple representations of the same data and to switch

between them at runtime as needed. So the sources are initially compiled into bytecodes (what they called v-code) and the first time a method needs to run it is translated into the native code (n-code) of the underlying hardware instead of being interpreted. This n-code is stored in a cache and so does not need to be translated again the second time it is called. If the compiler is simple and fast so that compilation is not too much slower than interpretation and if methods tend to be used several times then the gain in performance can be tremendous. In the Java world runtime compilation is known as JIT (for "Just In Time" compilation, a term borrowed from Japanese manufacturing techniques).

The most obvious performance boost of runtime compilation is the execution of native code instead of interpreted bytecodes, but the most important one is that some information that previously had to be considered dynamic can now be treated as static. When the value does change, the code can simply be compiled again and incorporate the new value as if it were constant. When Self (an even simpler and more radical version of Smalltalk) was developed, the only way to have a reasonable performance was to take this idea as far as possible [1]. Instead of generating a single version of n-code for a given v-code method, "customization" was the generation of several versions, each optimized for a different context. This is a special case of partial evaluation. The general theme of the first Self implementation was to trade memory for speed when possible. A significant optimization is inlining, which replaces a send (call) with the body of the code being called. This further transforms variables into constants and allows more optimizations to be done.

It was previously mentioned that the compiler for a JIT system should be nearly as fast as the interpreter it replaces, and while that was the case for Self 1 the more sophisticated compiler developed for Self 2 caused noticeable pauses during interactive execution. So the system had impressive benchmark results, making a dynamic system like Self perform numeric code at half the speed of C (while checking for overflows, invalid indexes, and so on) but it was not as fun to use as other Smalltalks. Software systems have a high temporal locality, however: 20% of the code tends to execute for 80% of the time. The solution is to use the fast, but bad, compiler for most code and only call the slow, good compiler for the few methods that really matter [6]. This is the solution shown in the lower right corner of Fig. 1. The Polymorphic Inline Caches, or PICs, that can be seen next to the n-code in the figure are a way of speeding up message sends (those that can't be inlined away) but which have the interesting side effect of collecting type information seen at runtime in each part of the code.

The "adaptive" in adaptive compilation is not only about selecting the right methods to optimize (which is why Sun's Java compiler with this technology is called HotSpot) but also about generating code specific to the types of objects actually used in the application, which can be very different from one run to the next. The Self 2 system obtained its impressive results using type analysis. This is a costly method (which is why it was so slow) that traces all possible data flows to find the best estimate for possible types in each part of the code even for systems, like Self and Smalltalk, which have no type declarations in the source code. The

analysis will yield the same result as long as the sources are not changed. In contrast, the good translator in Self 3 gets its type information by examining the PICs, which it can do since it is only called for code which has been running for a while. This type information is a subset of that found by type analysis, so the generated code must always include a check for new types that were not previously seen. In that case the code can simply be translated once more with the updated information. This allows the good translator to be nearly as fast and simple as the bad translator, but it needs the latter to do its job first and then for the code to run a few times before it can get the type feedback that it needs.

One way to think of the optimizations performed by the compiler is to suppose that the hardware can efficiently run code that has characteristics similar to Fortran, while the programmer is most productive in a more evolved language like Smalltalk. So the compiler must undo all of the advances from Fortran to Smalltalk: it inlines code (undoing factoring), it replaces polymorphism with switch/case statements (the PICs, see Fig. 2), it replaces generic code with multiple variations (undoing factoring and generalization) and, as has been said, it replaces dynamic tests with static information whenever possible.

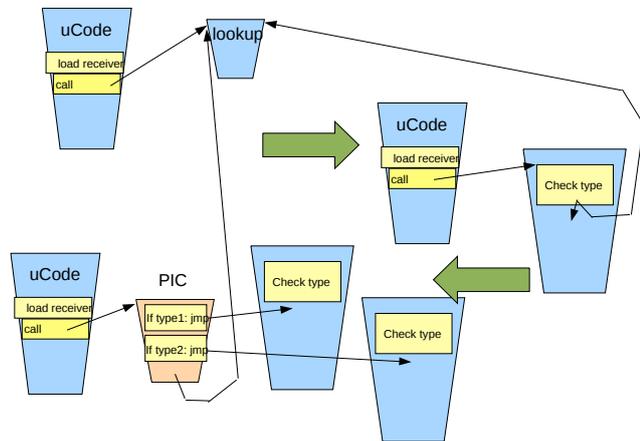


Figure 2: Polimorphic Inline Cache (PIC)

3. SILICONSQUEAK

For every language that is implemented using some kind of virtual machine (APL, Lisp, Forth, Smalltalk, UCSD Pascal, Modula-2, Java), there have been attempts to build actual hardware versions. In the case of Smalltalk, the microcode-based Xerox PARC machine on which it was first developed might be considered as such. In general, these projects did not have the scale necessary to keep up with software implementations as commercial processors advanced exponentially. Fortunately, FPGAs and designs implemented in them do keep up with the advances in microelectronic fabrication technology, and given the very low development costs there has been a renewed interest in language specific processors.

An interesting feature of the open source Smalltalk called Squeak is that it is implemented in itself [7]. When Smalltalk-80 was originally released by Xerox, the specification for the virtual machine (VM) was written in the form of a Smalltalk

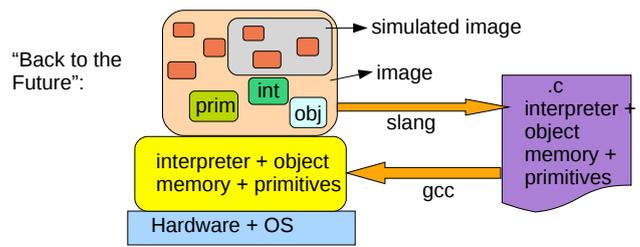


Figure 3: memory and primitives

program. This was not written in the normal Smalltalk style but in a very limited way so it would be easy to translate to Pascal or C, which several people did. But given that it was also perfectly valid Smalltalk, it would be possible to run this code inside another Smalltalk and, when computers became fast enough in the 1990s, this was actually done in the Hobbes project. The approach selected for Squeak, shown in Fig. 3, was to write the virtual machine in a subset of Smalltalk (called Slang, but it isn't really a separate language) and debug it by running on top of Squeak (or Apple Smalltalk before Squeak was finished) and then have that automatically translated to C when done. This allows the VM to be compiled to native code. Other subsystems, such as graphics or sound, can use the same technology to improve the performance through the creation of plug-ins for the VM.

Though it is tempting to think that a hardware version of the VM could be generated by automatically translating the Slang code to VHDL or Verilog, the result would be far too large to be usable. It is possible to split the VM into three separate parts: the bytecode interpreter, the object memory and the set of primitives. If a hardware version of the bytecode interpreter (the little green box in Fig. 3) and a few key primitives allowed the virtual instructions to run about as fast as the native instructions of other processors, then the Slang version of the object memory and the remaining primitives would have a performance similar to their translated-to-C-and-compiled-to-native versions. So only the hardware necessary to replace the bytecode interpreter, implemented as a mix of hand designed blocks and microcode, is actually needed and that can occupy a very small area in a FPGA.

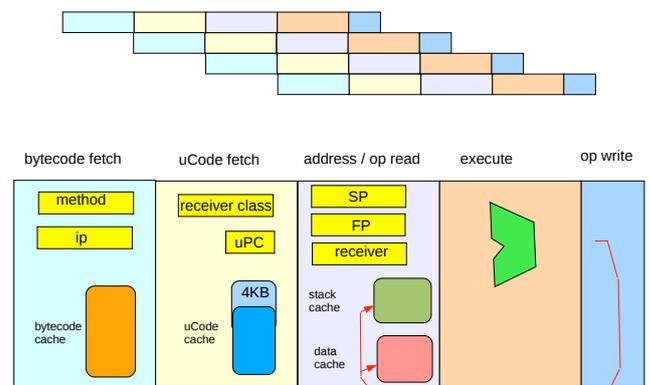


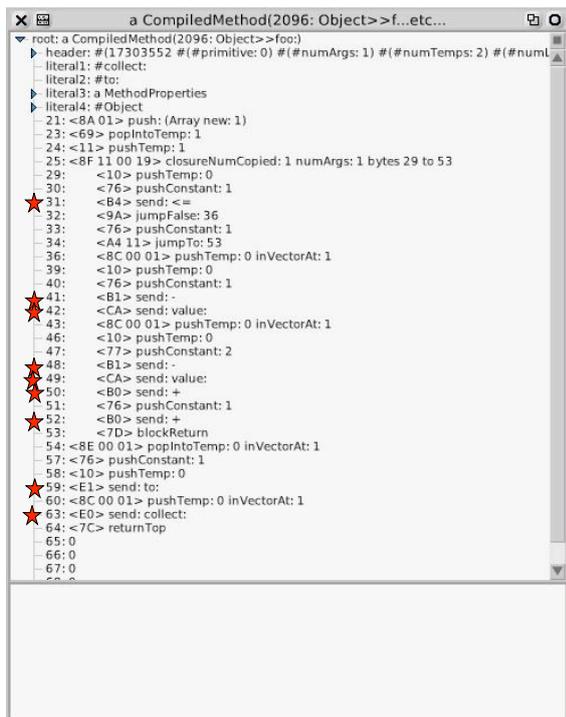
Figure 4: Bytecode interpreter pipeline

The key to high performance is the use of specialized memories to obtain the highest possible bandwidth. The four different cache memories shown in Fig. 4 are a good example of this. This sequence Smalltalk code:

```
foo: n
| nfib |
nfib := [:i | i <= 1 ifTrue: [1]
        ifFalse: [(nfib value: i - 1)
                  + (nfib value: i - 2)
                  + 1]].

^(1 to: n) collect: nfib
```

is translated into the bytecodes of Fig. 5:



★ Indicates the send bytecodes

Figure 5: Output of the source to bytecode compiler

Bytecodes can be executed at the rate of one per clock cycle in a four and a half stage pipeline (here depicted with colored blocks). A bytecode cache holds the source instructions for the current method, and the next byte fetched is used to jump to one of 256 small hand-coded routines that perform the bytecode actions. Many bytecodes can be actually performed by just one microcode within one clockcycle, sometimes two. These reside in a reserved area within the microcode cache. Message send microcodes are longer, however, but will still perform an order of magnitude faster than VM's without SiliconSqueak. Though optimized for Squeak, the processor can execute any set of bytecode instructions (for Java or Python, for example) by simply reloading that reserved area. The term "microcode" implies a limited memory, but the microcode cache can fetch code from external memory that can be as large as needed. And this code will

normally generated by the compiler rather than hand coded (except for the bytecodes themselves). So SiliconSqueak can emulate almost any language or processor this way. Microcode can be even jump to bytecodes and therefore call back into Squeak source code.

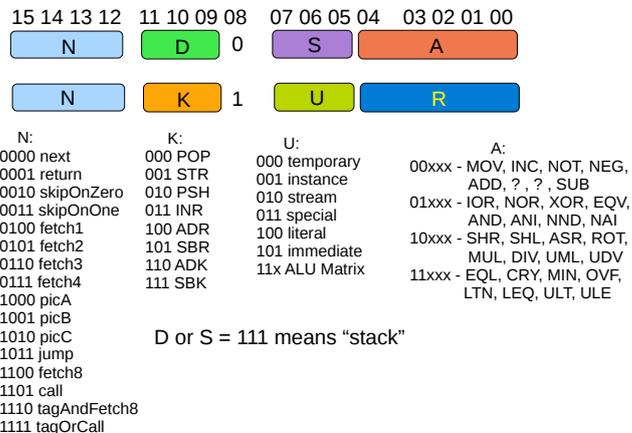


Figure 6: Microcode instruction format

The two formats for microcode instructions can be seen in Fig. 6. It offers a mix of two address register to register instructions and one address stack instructions. There is some of the flavor of the macrocode in the first Lisp machines, but there is not a further level of instructions below this one. Instead of some kind of "nanocode", the level below is in the form of virtual registers: the registers addressed by microcode instructions are mapped to positions in the various caches, specially the stack and the data cache.

As described so far, SiliconSqueak is an interpreter. Though some adaptive compilation systems do replace the initial compiler with an interpreter, they need a way to collect information that would normally be generated by the instrumented native code. It is possible to add PICs to Squeak, as suggested by [5], and these can be used as input for a translator that would generate microcode for the system hotspots (see Fig. 7). The close match between the actual hardware and the source bytecodes greatly simplifies the implementation of this translator and the organization of the instruction cache eliminates the need to generate code to check for the uncommon cases.

4. DYNAMIC, PARTIAL RECONFIGURATION

While some FPGAs use antifuse or Flash technology to store their configurations, the most popular option is to use RAM. This means that the configuration must be reloaded from an external source every time power is lost, but it also means that different configurations could be loaded at different times. Some projects have explored this to multiplex the hardware among different functions (between a transmitter and a receiver, for example) over time. Most FPGAs can only be configured after being reset or at power up, but there have been projects of circuits that would hold multiple configurations and be able to switch between them dynamically.

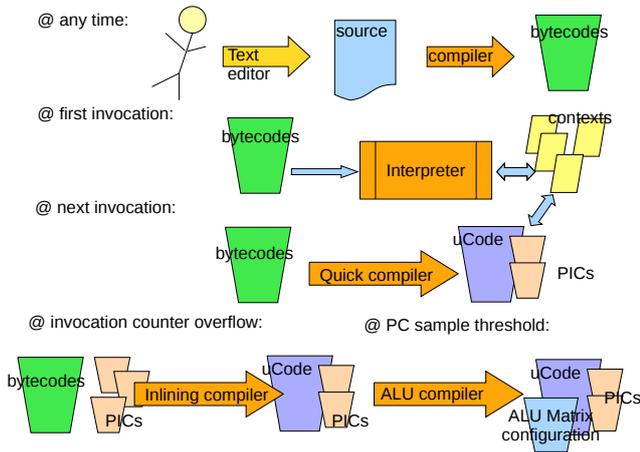


Figure 7: Adaptive compilation to PIC and Microcode and extended to configuration of the ALU Matrix

Starting with their Virtex family of chips, Xilinx has offered the possibility of partial reconfiguration where only a fraction of the logic blocks and routing circuits are reprogrammed while the rest of the FPGA remains as it was before [13]. Boards with multiple FPGAs can be designed to offer this capability even if the individual FPGAs themselves can only be configured as a whole, so this is an area that has also been explored using chips other than Xilinx.

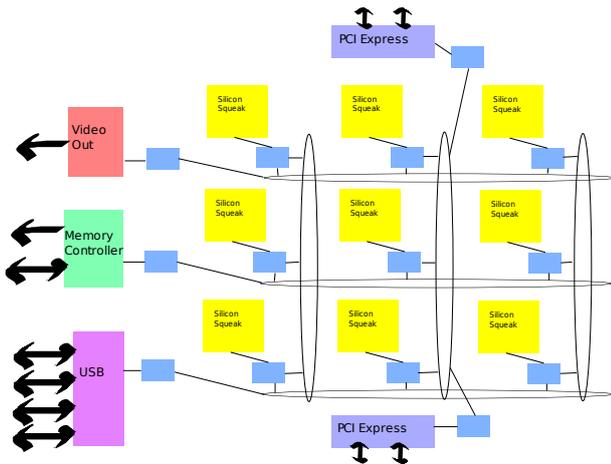


Figure 8: Typical SiliconSqueak SoC

It is more useful to be able to reprogram part of an FPGA if the rest of it can continue to operate normally while this goes on. This combination of dynamic and partial reconfiguration is very challenging and the tools to allow this are still rather primitive and hard to use.

5. EXTENDING ADAPTIVE COMPILATION

Fig. 8 shows the system on a chip (SoC) using an arbitrary nine SiliconSqueak cores connected through six ring networks in a 2D torus architecture. Everything shown is inside a single FPGA. Current FPGA's allow for more than 200 cores but have a less than optimal balance between core

cache and external memory bandwidth. An average of 8 cores seems the optimal in most FPGA on the market today as in Fig. 9.

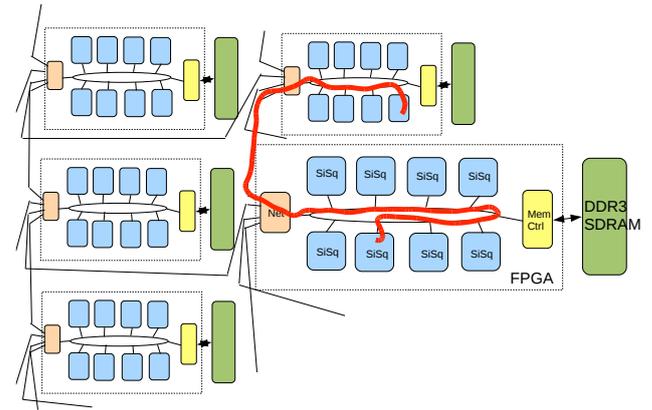


Figure 9: inter-core message send

The application must be written in a certain style to be able to take advantage of the multiple cores, but since the normal message send and return bytecodes are used for inter-processor communication, this is not a very severe restriction. For applications written as “kernels” connected by streams of data, it is easy for the adaptive compilation system to determine the most critical blocks. For the blocks which remain the hotspots even as optimized code, it would be possible to replace the software implementation with highly optimized software for reconfigurable but fast hardware like the 64 ALU matrix unit (Fig. 11) and the rest of the system would not notice as long as the communication protocol remained the same. So the extended adaptive compilation system would start with an interpreter with PICs, have one or more translators generating optimized native code and as a final step a translator would generate optimized code for one of a fixed set of hardware coprocessor units from a library and the translator would also replace several general SiliconSqueak cores with this hardware from the library for the most critical blocks by reprogramming a part of the FPGA. In an ASIC the hardware is fixed, so here the translator would generate optimized code for one of the free hardware coprocessor unit elsewhere on the ASIC.

Blue blocks in Fig. 10 and Fig. 9 represent generic SiliconSqueak cores, green depict an 64 ALU matrix including the adjacent SiliconSqueak core to control it, the orange block is a 4 x10 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, matrixes and units is accomplished by a ring network.

With an initial configuration as in Fig. 9, the newly generated hardware 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.

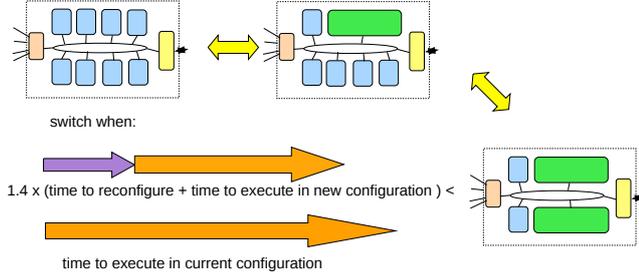


Figure 10: FPGA reconfiguration at work

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 coprocessor 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.

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

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

Where α is the percent of time that code that could use the coprocessor 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 coprocessor. 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 $\alpha > 84\%$.

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

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

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 $\beta < 5\%$.

For adaptive compilation to be truly extended to hardware generation it would have to be possible for the runtime system to take the bytecodes as its source and translate them to a hardware description language and then be mapped (called synthesis by vendors) onto the final FPGA configurations bits of fixed FPGA logic blocks. With current technology this would take a long time, which must be considered in any scheduling decisions. A more serious limitation is that the detailed information about the configuration bits for all commercially available FPGA systems is not available and so can't be incorporated into our adaptive compilation system. These bits are generated by proprietary closed source tools running on normal PCs (not on SiliconSqueak). Creating our own FPGA system with an open source mapping system is beyond our reach as it would require a substantial investment in creating the VLSI hardware. Also, translating any arbitrary code to transistor logic efficiently is beyond the state of the art. Another important limitation is that hardware generation would only apply to programmable hardware like FPGAs. As ASIC hardware is still an order of a magnitude faster than FPGA's, another solution was chosen that would be usable in both FPGA and ASIC and take all other limitations into account. We choose to let the adaptive compiler regenerate the software for a highly optimized coprocessor unit and in the case of an FPGA also reprogram the FPGA with a different balance of cores and coprocessors.

Given the limitations with automatic hardware generating it was considered reasonable to impose restrictions (not unlike the Slang solution for the Squeak VM) on the mapping of arbitrary transistor logic blocks for this project. So an approximation was chosen where a set of blocks are pre-compiled offline and made available as a unit, a logic block library for the adaptive compilation system, which then either merely loads the proper one when available or gives up when it is not. A first subset of transistor logic blocks is implemented for this project in the form of groupings of a 64 ALU matrix unit. Other libraries we allow for in the adaptive compiler as future extensions, each successively less generic in nature, are a ring network architecture unit, bit-slice units, vector processing unit, floating point unit, graphics pipeline unit, move processor unit., cryptographic unit, reconfigurable asynchronous logic automata units, string comparator unit and associative memory. A future design could also allow for a different balance of ring network width, number and speed of network unit links and memory controller bandwidth. Another direction would be to allow for different types of generic cores optimized for other programming languages insofar that the current fixed microcode is only efficient for bytecode languages.

The 64 ALU matrix implemented in this project shown in Fig. 11 is a generic set of ALU's with 64 types of instructions that can be pre-configured with bits into a matrix. The 8 bit wide ALU's have carry bits to allow for concatenation into wider words up to 512 bits wide. The flow between ALUs is facilitated by 16 input and output registers that are reconfigurable by SiliconSqueak cores before and during

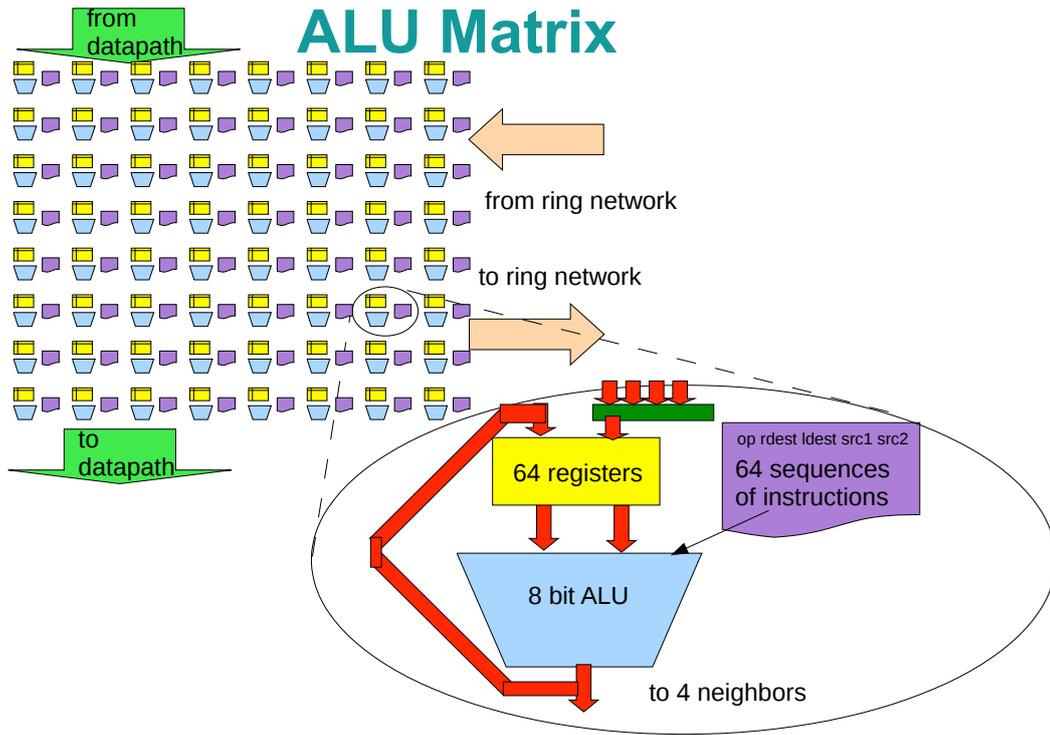


Figure 11: 64 ALU Matrix

execution flows but we restricted the flow from 8 into only 2 directions to make efficient use of available FPGA logic block and memory structures. Allowing for all possible directions would more than double the amount of FPGA logic block needed, tipping the balance into having multiple generic SiliconSqueak cores than ALU matrixes. An optimal balance is where both units need equal logic block amounts.

6. DISTRIBUTED OBJECT MEMORY AND PARALLELISM

A core accesses only objects in its local cache, transparently retrieved by message send from local DRAM attached to the SoC but also from other caches or DRAM memory attached to any other core or Soc connected through the external links (see Fig. 9), even from remote virtual machines running on different hardware or across the internet. This globally and transparent retrieval is accomplished by the ring network architecture on the Soc and by the external communication links. All objects have a object memory reference and the transport hardware distinguishes automatically if an address is local or remote object reference into the object heap. Remote messages sends, retrieving or storing objects, traveling from ring to link to ring to object memory heap through the hardware network interconnection fabric. A certain remote core can receive messages addressable through external networks like the internet. Here a hardware message send can re-enter a core to be encapsulated by a software method as a packet for an external protocol like IP or ethernet packets. In this way all the cores and special purpose processing units can transparently address the entire object heap memory of an entire distributed system of manycore processors. Note

however that it can not address any (shared) memory locations, just object references from the object heap. Moving an object from a local part of the object heap into a remote part of the object heap and then sending a message to it makes the method operating on the object be executed in parallel. Almost transparent parallelism can now be introduced by the adaptive compiler by moving around objects to load balance between methods executing on objects. Future sends can further optimize use of parallel constructs

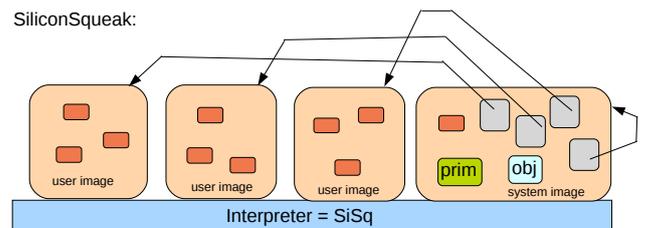


Figure 12: distributed object heap

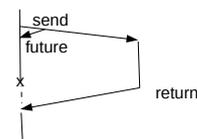


Figure 13: future message send

7. RELATED WORK

The Renaissance Project at IBM Research by Sam S. Adams and David Ungar resulted in a version of the Squeak VM which runs on 56 of the 64 cores of the Tiler64 chip. The modifications were almost entirely to the object memory part of the VM, yet very interesting parallel execution results were obtained without introducing any new parallelism constructs in the Squeak language [11]. They did introduce parallel constructs in another language on top of the Squeak VM [12].

BORPH [10] is an operating system that dynamically allocated FPGA areas to precompiled blocks. Software adaptation layers allow these blocks to be mixed with normal software filters on a Unix command line.

The Sonic-on-a-Chip video processing system [9] developed some custom partial reconfiguration tools to compensate for the limitations of the official tools from Xilinx. These limitations are not as severe in the more recent FPGA families, however.

8. CONCLUSION

Adaptive compilation can significantly increase the performance of object-oriented programming languages and can work well with parallel systems, which themselves help increase performance. The high costs associated with multiple and complex compilers at runtime have so far restricted this technology to desktop and server environments.

A microprocessor that could implement both a high performance interpreter with instrumented execution and also had simple to generate code and efficient uncommon case detection could greatly reduce the costs of adaptive compilation (by only ever translating a small fraction of all code executed), and these are all design goals for SiliconSqueak.

With an adaptive compilation scheme in place and an application written in the form of communicating modules, nodes in a network of SiliconSqueak cores could be dynamically replaced with dedicated hardware for the most critical modules for the best possible performance. The hardware could change continuously as the application requirements evolve.

9. REFERENCES

- [1] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 146–160. July 1989.
- [2] J. M. de Assumpção Jr. O sistema orientado a objetos merlin em máquinas paralelas. In *V SBAC-PAD: Simpósio Brasileiro de Arquitetura de Computadores, Processamento de Alto Desempenho*, pages 304–312. 1993.
- [3] J. M. de Assumpção Jr. Adaptive compilation in the merlin system for parallel machines. In *WHPC'94 Proceedings - IEEE/USP International Workshop on High Performance Computing*, pages 155–166. 1994.
- [4] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 279–302. Jan. 1984.
- [5] M. Haupt, R. Hirschfeld, and M. Denker. Type feedback for bytecode interpreters. In *Proceedings of the Second Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2007)*, pages 17–22. July 2007.
- [6] U. Hölzle. *Adaptive optimization for Self: reconciling high performance with exploratory programming*. PhD thesis, Stanford University, 1994.
- [7] D. Ingalls, D. Ingalls, T. Kaehler, T. Kaehler, J. Maloney, J. Maloney, S. Wallace, S. Wallace, A. Kay, and W. D. Imagineering. Back to the future: The story of squeak, a practical smalltalk written in itself. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, pages 318–326. ACM Press, 1997.
- [8] M. Noakes, D. A. Wallach, and W. J. Dally. The j-machine multicomputer: An architectural evaluation. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 224–235, 1993.
- [9] N. P. Sedcole. *Reconfigurable Platform-Based Design in FPGAs for Video Image Processing*. PhD thesis, Department of Electrical and Electronic Engineering, Imperial College of Science, Technology and Medicine, University of London, Jan. 2006.
- [10] H. K.-H. So. *BORPH: An Operating System for FPGA-Based Reconfigurable Computers*. PhD thesis, Engineering – Electrical and Computer Sciences, University of California, Berkeley, 2007.
- [11] D. Ungar and S. S. Adams. Hosting an object heap on manycore hardware: an exploration. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, pages 99–110, New York, NY, USA, 2009. ACM.
- [12] D. Ungar and S. S. Adams. Harnessing emergence for manycore programming: early experience integrating ensembles, adverbs, and object-based inheritance. In *SPLASH '10 Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, New York, NY, USA, 2010. ACM.
- [13] Xilinx. Virtex-4 configuration guide. Technical Report UG071, Oct. 2007.