Adaptive Compilation in the Merlin System for Parallel Machines

Jecel Mattos de Assumpção Jr *

Abstract

This paper describes the use of Adaptive Compilation, a system of optimizing code based on information gathered while running the code itself, to match the parallelism of the application to the characteristics of the underlying hardware.

1 Introduction

Object oriented programming is not usually associated with high performance computing (for very good reasons), yet it is a software model that matches nicely the hardware of distributed memory parallel computers. The Merlin Project[AJ93] is an object oriented operating system designed around the Self language developed at Stanford University. It was designed to run on a variety of computers, including the MS8702[AJ92] (a 64 node mesh distributed memory prototype built at the Laboratório de Sistemas Integráveis of the University of São Paulo) and clusters of networked workstations.

The next section will trace de development of pure object oriented languages up to the development of Adaptive Compilation, the key technology for high performance implementations. An overview of the Self programming language will help the reader grasp the issues discussed in the rest of the paper.

The Merlin System, while based on the work done at Stanford University, has several unique features relevant to parallel machines, as described in the part titled "The Merlin Runtime Environment".

The main part of the paper shows how Adaptive Compilation can use actual runtime information to optimize the generated code for a given hardware configuration and system load. As these conditions change dynamically, the system might recompile parts of an application for an overall gain in performance.

*internet: jecel@lsi.usp.br

2 A Brief History of Adaptive Compilation

Smalltalk-72[GK76] was the first implementation of a pure object oriented programming language. It was very dynamic and, as a result, rather slow. The syntax itself was determined at run time, which made it impossible to compile and hard for people to understand each other's code. A major change was made in the 1976 version of the language: a simple, fixed syntax was chosen and the implementation changed to a two step execution - text was compiled to the machine language of a "virtual machine" with an interpreter simulating this machine on the physical computer at runtime. The virtual instructions defined a simple stack machine (with the addition of complex send instructions) and are called "bytecodes" due to their length. Most current Smalltalk implementations are virtual machine interpreters, so it is not surprising that pure object oriented languages are considered inefficient.

Several projects tried, in the early 80s, to implement the virtual machine in hardware. This did not, however, result in speed improvements on the order of those obtained with SOAR[UBF+84], Smalltalk On A RISC, from the University of California, Berkeley. That group added a minimal hardware support for objects (which can be seen in today's commercial SPARC processors) and compiled source text directly to native machine code. An interesting technique used in this system was the "inline cache", invented by L. P. Deutsch, to replace costly message send lookups with simple subroutine calls whenever possible. This works very well where the flexibility of Smalltalk's *polymorphism* is not actually used.

The inline cache worked by compiling a message send to a call to a method lookup routine. At runtime, when the target method is found, the original "call lookup" instruction is patched to a direct call to the newly found method. That method has a small header that will check the object type (class) to see if it is what was expected and call the lookup routine if it is not. So the next time the same code is executed, the system will "guess" that the target method will be the same as the last time and call it directly. Only if the guess is wrong will the inefficient lookup be done.

A major problem with compiling Smalltalk directly to RISC code turned out to be the size of the resulting methods. A Smalltalk system has a lot of code, and a fourfold decrease in its density has a negative impact on performance due to a high virtual memory overhead. A compromise was dynamic translation[DS84], which we will call *dynamic compilation*. The source text is translated into bytecodes. The first time a method is called, it is not interpreted but compiled to native machine instructions which are stored in a cache and executed. The next time the same method is needed it is simply executed from the cache, with a considerable increase in average performance. When the cache becomes too full, some of the compiled codes are simply discarded. They will have to be recompiled if they are needed later, but this has been shown to be faster than saving the large native code to disk given the relative speeds of disks and microprocessors.

The programming language Self[US87] is closely related to Smalltalk, but is based on fewer, more general, concepts. It can be described as a prototype based language with "message passing at the bottom". Self objects are structured as a series of named slots. When the object receives a message, it will lookup the slot with the name indicated in the message and either simply returns the slot value if it is a simple object (we call this a data slot), executes the code if the slot contains a method object or, if the slot has an assignment primitive, it changes the object's corresponding data slot to the value of the message argument (assignment and data slots are related by name - the assignment slot "x:" changes the data slot "x"). In this simple model each object is self-describing and there is no notion of classes. The lack of classes means that objects can only be created by copying (cloning) existing objects (the prototypes) and then modifying them. The assignment slots can be used to change the new object's *state*, but there are some programming primitives (_AddSlot: and _DeleteSlot:) that allow more fundamental modifications to the object.

To avoid code duplication, an object may inherit from one, or more, parent objects. When a message is received and the object has no corresponding slot, the parents will be searched (recursively) and the located method will be executed with the original object as the receiver. The parents are indicated using specially tagged data slots. These slots can still be used in the normal ways, which means you can change parents "on the fly" to implement *dynamic inheritance*, an interesting (but inefficient, so far) feature.

While Smalltalk's virtual machine is rather conventional with a few additions, Self is radical in its exclusive use of message passing. The Smalltalk code fragment "x $\leftarrow$ x + 1" would translate into: push instance variable (x), push constant (1), send message (+) and pop into instance variable (x). The Self equivalent, written as "x: x + 1" and meaning "*self* x: *self* x + 1" (hence the language's name), generates the following bytecodes: send message to self (x), push constant (1), send message (+) and send message to self (x:). As message passing is the bottleneck in the implementation of pure object oriented languages, it would seem Self would be much slower than Smalltalk. There are advantages to this approach: the virtual machine is much simpler (just eight different instructions) and it forces a coding style that leads to a greater reuse of code.

With dynamic compilation, message sends that would access a data slot or an assignment slot can be inlined into the sending method's native code for a speedup of 4 to 160 times[Cha92]. This is only possible because the receiver of these messages is *self* which has a known "type" at runtime (when the compiler is run). When inheritance is taken into account, however, this is no longer true as *self* will be different for objects which inherit this method but vary in more ways than simply data slot values. The solution is to compile a different version of native code for each inherited use of the same method, what we call *customized compilation*[CU89]. This is a practical approach only because the compiler is called for those cases which are actually needed at runtime, just a tiny fraction of all possible combinations. Without customization, Self would be around three times slower as very little optimization would be possible due to the different contexts in which a method could be called.

As Self's compiler grew in sophistication[CU90][CUL89], overall performance edged closer to highly optimized languages like C, but interactive use became worse as runtime compiler pauses became longer. As Self programs have no explicit type information, the compiler must work very hard to extract the implicit information available to generate high quality code. An interesting optimization used by the Self compiler is an extension of the inline cache used in Smalltalk - the polymorphic inline cache (PIC)[HCU91]. This replaces the call to the lookup code with a series of type tests similar to a C switch statement. If the message receiver has the same type as in one of the previous message sends, then the correct native code will be called directly. Otherwise, the lookup routine is invoked to find the correct method and extend the PIC to include the new type. This extends the advantages of the inline cache to polymorphic message sends too, which are typical in object oriented programs. If too many different types are seen in a piece of code, the PIC can discard the oldest types or even be entirely replaced with a simple call to the lookup routine.

A side effect of the PICs is that they accumulate information about the types of objects actually used in various points in the compiled code. This information is a subset of that obtained by the compiler with its sophisticated analysis. It is this fact that makes *adaptive compilation* a practical strategy: compile the methods with a crude (but fast) compiler, run the code for a while and then compile it again using the PICs as the source for type information. Both the first and subsequent compilers can be simple and fast, and only methods that are frequently called need go through the re-compilation process - this directs the most effort to where it will do the most good.

With adaptive compilation, the code is tuned to the actual runtime conditions, including program input. The native instructions for an application might be very different, for example, depending on whether the user is editing color images or black and white ones. This technique can use information that might be too costly to obtain statically.

From the programmer's point of view, there is only one type: the object. But the compiler must be more discriminating if it is to turn out acceptable code. A careful observation of the actual usage patterns in Self will show that many objects are exact clones of each other except for the values in the data slots. A practical implementation will, therefore, separate an object into two parts: a holder for the data slot values and a container for all the rest, which we will call the object's *map*. When an object is cloned, only the first part is copied; sharing the map among many objects is a tremendous space saving strategy. It also makes up for the lack of classes to the compiler by allowing it to treat these "clone families" as the equivalent of type. The full flexibility of Self is preserved, however, as any object can be changed at any time by the programming primitives. In this case, a new map will be created and the object will start a new clone family, leaving all instances of its former "type" unchanged.

3 The Merlin Runtime Environment

The Merlin operating system was designed for easy to use, personal workstations. It is a distributed system that can pool the resources of a networked cluster of such workstations to solve performance-critical problems. A distributed memory multicomputer, like the 64 node MS8702, can also benefit from Merlin's parallel computing features. These hardware configurations consist of many communicating processors, each with its own local memory. While most popular programming models require a global, shared memory, Self's objects interact exclusively by exchanging messages - each object has its own private memory. Self is clearly a good match for this type of machine.

The support for parallelism in Self, as developed at Stanford and Sun[1], is a simple thread package written almost entirely in Self. There are process objects, each with its own stack and, while individual stack frames can be manipulated via activation objects, these are created on demand when debugging. The compiler and PICs make effective use of the processor's native stack allocation abilities to speed up message sends.

In contrast, Merlin associates a single thread of execution with every object. When an object receives several messages, they are stored in a queue from where the objects fetches them one at a time, executing them to completion with no interleaving. The message passing semantics are "wait by necessity"[Car90], which makes it as compatible as possible with sequential Self while exposing a usable degree of parallelism in applications. The idea is that every send returns immediately with a temporary answer called *future object* or simply future. When the real answer is returned by the message receiver, it replaces the future and all continues as if it had been returned as the result of the original message send. If a method tries to send a message to the future object before it is replaced, it will block until then so that futures are never actually seen by the calling code. The programmer does not usually worry about what can, or cannot, be executed in parallel.

This simple model would, unfortunately, lead to deadlocks whenever an object sent a message (even indirectly) to itself, hardly a rare occurrence in a language called Self. These cases are detected specially and allowed to "borrow" the execution thread from the calling method in a way consistent with sequential execution.

[1] Self 3.0 for SparcStations, sources and related papers can be obtained by anonymous FTP from self.stanford.edu

Simply browsing through Self source code, it is easy to distinguish three types of messages: those whose result will be the receiver of another message, those whose result will be an argument of another message and those whose result will be discarded. The first case implies strictly sequential execution in our model, for the returned future will be sent a message immediately. In the second case, the result of the message might be need only much later, allowing it to be calculated in parallel with the calling method. The result is never needed in the last case, which seems to imply an opportunity for parallelism. This would normally work, but would lead to surprising behavior relative to sequential Self in several cases, and so is defined as blocking until the (irrelevant) result is ready. A static analysis of Self 3.0 source reveals:

41,880 messages are used as a receiver, 43%

23,329 messages are used as argument, 24%

31,573 messages are "top level", 33%

So, even with top level messages defined as sequential, one in every four sends potentially adds to the parallelism of a program. Many practical factors will drastically reduce this number. Even so, the inherent parallelism this model yields is far greater than the number of available physical processors. One goal of adaptive compilation in Merlin, then, is to determine what potential parallelism is not actually used and to change it to sequential messages (a perfectly acceptable transformation in the wait by necessity model) to reduce its overhead.

One important role that classes have in Smalltalk is the ability to see instances as objects — that is, as structured entities in a computer's memory — and not as the abstraction they represent. To implement this *reflective computing* in Self, it is possible to create a "mirror" on an object. While the object itself can only accept messages related to its abstraction (a window only understands window messages, for example), the mirror can receive messages that ask about the number of slots in its reflectee, their names, and all kinds of related information. In Merlin, reflection is extended to the dynamic aspects of object execution as well, with a structure similar to the Apertos system[Yok92][YTM+91] from Sony Laboratories[2]. An objects computational resources such as CPU usage, memory and disk allocation and message transport mechanisms are represented by *meta-objects* which can be accessed, when necessary, through a mirror. Each object also has an owner, which allows it to be protected when operating in a network with several different users.

Three very different types of objects make up the implementation: normal Self objects, maps and contexts. Normal objects are where most application level information is stored. The maps have structuring information about their objects and serve as a portal to the meta-level of the system. Contexts store information about on-going computation and do double duty as future objects

[2]papers on the Apertos Operating System may be obtained by anonymous FTP from scslwide.sony.co.jp

too. When a message is sent, a new context is created and queued for execution at the receiver object. A pointer to this context is immediately returned to the original method, where it serves as a place holder for the eventual answer. While a message cannot be sent directly to this future without blocking the sender, a mirror can be used to manipulate it as a context (normally for debugging). The result of this design is that all aspects of computation can be examined or changed within the framework of a single language, so that features like object migration can be redefined by the programmer when necessary.

4 Adaptive Compilation on Parallel Machines

An extreme case of a hardware environment for Merlin would have several networked workstations and one or two highly parallel machines like the MS8702 or an Intel Paragon. The abstract model for this system is a set of communicating computational nodes, though they may have varying memory and processing capacities and might even have different native instruction sets. A single node may have several (identical) processors, but this will be considered equivalent to a single processor node with a higher capacity for this discussion, even though this is a vast simplification.

Though every node is logically connected with every other node, the available bandwidth between them may vary orders of magnitude. A critical element in a program's performance, then, is the distribution of objects among the physical nodes. Two conflicting needs serve as the guiding forces for this distribution: objects which communicate will do so more efficiently if placed in the same node (specially if the compiler can inline away most of these message sends), but objects must be in different nodes if they are to actually execute in parallel. While Merlin's reflective structure allows the object distribution system to "peer under the hood" to obtain communications pattern, the information about actual parallelism in the running code must complement this if the distribution policy is to optimize system performance.

Future objects allow synchronization to occur according to the application's data dependencies. As the future may be freely passed around as a normal object, several methods (possibly none of them the one that sent the original message which created the future) might try to send it a message and block before the future's reply is ready. When that happens, all of the blocked methods will have their reference to the future updated to the reply and will be unblocked. The future cannot be simply discarded, however, as there may be more references to it floating around in the system. If a method tries to send a message to the (ex-) future now, it will have its reference updated and continue without blocking. The garbage collector will eventually update all references to the future (including those that are never used) so that the future's memory can be reused.

A very important metric on program execution is the ratio of methods that are blocked on a future when it returns relative to the number of those that first try to use it after that. For those messages that were described as synchronous in the previous section, it can be directly seen from simply looking at the source that this ratio would be infinite (1/0) while for top-level messages it would be undefined (0/0). For the remaining messages, this must be measured by the runtime system, with the smaller numbers indicating that it would be desirable to run the sender and receiver of the message in parallel. For objects on the same node, this number will be very dependent on the scheduler's policies and may not be a good indication of what will happen if one of the objects migrates to another processor. As it is a low cost feedback, however, it meshes well with the current implementation's goal of doing it fast and readjusting it later.

The first compilation does not use inlining — it generates code which is correct (but slow) and that exposes the full potential for parallelism in the program. Subsequent compilers transform most message sends into sequential ones, specially those that can be inlined. The native code is optimized for the runtime conditions and hardware configuration, but is less sensitive to changes in these conditions as the inlined message sends no longer contribute to the calculated system execution profile. When the environment changes wildly from what was implicitly built into the code, this can be simply flushed from the compiled code cache to force a recompilation adapted to the new parameters.

Other factors, not discussed in this paper, have a great impact on object distribution and adaptive compilation. The protection system, for example, allows the system to identify objects that cannot change during some program's execution. The distribution system can use this fact to allow copies of these objects while the compiler may treat all of the object's slots as constants and generate very efficient code.

The key to all of these techniques is to measure what is going on, to make things that happen often as fast as possible and to make rare things possible. The full flexibility of the language is preserved, but applications that use only features present in simpler systems do not have to pay the cost of this flexibility. This allows a single programming environment to be used for all aspects of a distributed application.

5 Conclusions

Adaptive compilation introduces a performance penalty in the form of runtime compilation pauses. This is more than compensated by the availability of a large amount of information on the actual runtime characteristics of a program to be used in the optimization of generated code. The costs of obtaining such information beforehand from the source text would be excessive, or would require severe restrictions that would considerably reduce the programming environment's flexibility.

Rather than do a costly analysis to extract parallelism from sequential code, Merlin's approach is based on a model with a large amount of "natural" parallelism and relying on adaptive compilation to optimize away that which exceeds the hardware's capacity. This frees the programmer from low level details such as hardware configurations or explicit data distribution.

References

[AJ92] Jecel Assumpcao Jr. Supercomputador orientado a objetos. In *Anais do IV SBAC-PAD*, pages 337–345, Sao Paulo, SP(Brazil), October 1992. In Portuguese.

[AJ93] Jecel Assumpcao Jr. O sistema orientado a objetos merlin em máquinas paralelas. In *Anais do V SBAC-PAD*, pages 304–312, Florianópolis, SC(Brazil), September 1993. In Portuguese.

[Car90] Denis Caromel. Concurrency and reusability: From sequential to parallel. *Journal Of Object-Oriented Programming*, 3(3):34–42, Sep/Oct 1990.

[Cha92] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, Stanford University, March 1992.

[CU89] Craig Chambers and David 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, Portland, OR, June 1989. Published as SIGPLAN Notices 24(7), July, 1989.

[CU90] Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 150–164, White Plains, NY, June 1990. Published as SIGPLAN Notices 25(6), June, 1990.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF, a dynamically- typed object oriented language based on prototypes. In *OOPSLA'89 Conference Proceedings*, pages 49–70, New Orleans, LA, October 1989. Published as SIGPLAN Notices 24(10), October, 1989.

[DS84] L. Peter Deutsch and Allan 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, January 1984.

[GK76] Adele Goldberg and Alan Kay. Smalltalk-72 instruction manual. Technical report, Xerox PARC, Palo Alto, 1976.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented programming languages with polymorphic inline caches. In *ECOOP'91 Conference Proceedings*, pages 21–38, Geneva, Switzerland, July 1991.

[UBF+84] David Ungar, Ricki Blau, Peter Foley, Dain Samples, and David Patterson. Architecture of SOAR: Smalltalk on a RISC. In *Eleventh Annual International Symposium on Computer Architecture*, pages 188–197, Ann Arbor, MI, June 1984.

[US87] David Ungar and Randall B. Smith. SELF: The power of simplicity. In *OOPSLA'87 Conference Proceedings*, pages 227–241, Orlando, FL, October 1987. Published as SIGPLAN Notices 22(12), December, 1987.

[Yok92] Yasuhiko Yokote. The apertos reflective operating system: The concept and its implementation. In *OOSPLA'92 Conference Proceedings*, 1992. also SCSL-TR-92-014 of Sony Computer Science Laboratory.

[YTM+91] Yasuhiko Yokote, Fumio Teraoka, Atsushi Mitsuzawa, Nobuhisa Fujinami, and Mario Tokoro. The muse object architecture: A new operating system structuring concept. *ACM Operating Systems Review*, 25(2):22–46, April 1991.