Porting an Interpreter and Just-In-Time Compiler to the XScale Architecture

Malte Hildingson

Dept. of Informatics and Mathematics
University of Trollhättan / Uddevalla, Sweden
E-mail: tds00mahi@thn.htu.se

Abstract

This exploratory study covers the work of porting an intermediate language interpreter to the ARM-based XScale architecture. The interpreter is part of the Mono project, an open source effort to create a .NET-compatible development framework. We cover trampolines together with the procedure call standard, discuss memory protection and present a complete implementation of atomic operations for the ARM architecture.

1. Introduction

The biggest problem with porting software is finding which parts of the software reflect architectural features of the hardware that it runs on [1]. The open source movement has been a benefactor in the sense that standards for porting open software have been in demand and developed.

There are a number of different strategies for compiling software to a specific platform; the easiest and most straight-forward way is to have all the development tools installed on the machine on which the software is intended to execute. The second option is to cross-compile the software on a machine of different architecture with tool-chains created to compile source code targeting a specific architecture. A third alternative, perhaps not as widely used but which may be preferable at times, is the use of emulators.

In this exploratory study the cross-compilation approach was used; the compilations were conducted on an Intel x86-compatible [2] computer which was set up to cross-compile towards the targeted Intel XScale-equipped [3] iPAQ [4]. Both machines were running the GNU/Linux [5] operating system with Red Hat [6] and Familiar [7] distributions, and the compilation suite at hand was the freely available GNU [8] Compiler Collection (GCC) [9]. Software configuration management in UNIX [10] environments traditionally rely heavily on makefiles, in which the correlation between platform specific details and different compilation options and procedures are reflected. To ensure that the compilation of code conformed to the targeted environment, the makefiles had to be adjusted accordingly. This task was hugely simplified by tools such as autoconf and automake which performed the necessary routines for the target platform given required input parameters, and created makefiles which ensured that the code was compiled properly.

2. Background

The Mono [11] project, launched in July 2001 by Ximian Inc. [12], is an effort to create an open source implementation of the .NET [13] development framework. The project includes the Common Language Infrastucture (CLI) [14] virtual machine, a class library for any programming language conforming to the Common Language Runtime (CLR) [15] and compilers that target the CLR. The virtual machine consists of a class loader, garbage collector and an interpreter or a just-in-time (JIT) compiler, depending on the configuration. The virtual machine is released in two configurations, an interpreter named Mint and a just-in-time compiler known as Mono. An early prototype for a next generation just-in-time compiler surfaced during a shorter period of time and was known as Mini, it was later accepted as the primary just-in-time and renamed Mono like its predecessor.

The Common Language Infrastructure together with the C# programming language [16] were made standards in December of 2001 by the European Computer Manufacturers Association (ECMA) [17], a Geneva-based standards organisation. The specifications that were the foundations of these standards were submitted in August, 2000, by Microsoft [18], Hewlett-Packard [4] and Intel. The standards had also in turn been sent to the International Organisation for Standardization (ISO) [19] with the intention to gain recoginition by a number of governmental organisations that does not recognise the ECMA standard. ISO later ratified the standards in April, 2003.

Surrounding Mono are also various third-party software libraries, acting as a glue between the traditional environment implemented in the C-language and the Common Lan-

guage Runtime. A number of binding libraries already exist, for example the ones for two highly popular portable graphical application programming interfaces, the Gimp Toolkit (GTK) [20] and Trolltech's QT [21]. And just a 'stones throw away' are bridges to whole desktop applications or indeed desktop environments. An effort to implement a binding to the GNOME [22] desktop has already been initiated and goes under the name of GNOME.NET.

Several other just-in-time compilers have already been ported to the ARM [23] and thus also XScale architecture, most for the Java [24] language; on the Java front there even exist processors capable of directly processing Java bytecode. For the CLI, Microsoft has ported its .NET runtime to the proprietary Windows CE platform [25], and the DotGNU [26] effort Portable.NET has an ARM-enabled runtime, targeting GNU/Linux, early in its development. In effect, this study were to enable the Mono project to have a fully-featured runtime for the ARM-based XScale processor often used in handheld devices and embedded solutions.

3. Implementation of Mint, the Interpreter

An ARM port of Mint, the Mono interpreter, mainly focusing on the Microsoft Windows Embedded platform had already been initiated by Mr. Sergey Chaban. The port had, however, not been maintained for a lengthy period of time and structural changes within the runtime itself had left it in a nonoperational condition. Thus the work became an effort to reinstate and improve its functionality.

Given that this was at a very early stage, it was here the majority of the preparatory studies were undertaken. A major part of these studies involved the ARM Procedure Call Standard (APCS) [27], to understand and be able to actively take part in the work with the trampolines. When constructing a trampoline everything involving a procedure call is done manually. Registers are stored, the procedures parameters are passed and return values are handled. This made full understanding of the ARM assembly language and the ARM Procedure Call Standard a necessity to accomplish the task. The ARM Procedure Call Standard defines how subroutines may be written, compiled and assembled separately, yet still work together. It states the contract between the calling and the called routines with definitions of how memory should be allocated by the caller in which the routine may execute, how the called routine shall preserve the memory of the caller across the call and what rights the called routine have to alter the memory-state of its caller.

The ARM instruction set is in general similiar to related RISC variants, for instance those of the Sparc [28] and PowerPC [29] processors. This made the PowerPC port of the Mint, at least initially, an anticipated template for the ARM code. However, the PowerPC port were not as well evolved as the x86 ditto, and so it eventually became the x86 port that was further analysed and became the inspiration to the interpreter's ARM-specific code.

```
mono_create_trampoline
mono_create_method_pointer
```

Figure 1. Mint architecture specific functions

The Mono interpreter defines only two functions (figure 1) to be implemented by each architecture. The trampolines are created to bridge Common Intermediate Language (CIL) [14] method invocations to a C-language function calls. When *mono_create_trampoline* is called it is passed a pointer to a data structure defining the method signature of the method to be bridged. It is also under contract to return a *MonoPIFunc* pointer, which in essence is a C function pointer which executes the created trampoline.

```
typedef void (*MonoPIFunc)
  (MonoFunc callme, void *retval,
  void *obj_this,
  stackval *arguments);
```

The trampoline is basically a function with instructions to make a new function call in a higher level of abstraction - there is the pointer to the function to call, passed in the parameter *callme*; there is a pointer *retval*, which is assigned the address to any return value received from the function, next is a pointer to the object that invoked the method in the CIL context. Finally there are the parameters in the *arguments* list that are passed to the targeted function.

The trampolines are created by emitting native instructions to executable memory. The work with creating code generation macros and functions have double significance as they are also used with the just-in-time compiler. Code for instruction emission had already been well designed, as the port had already been initiated, which helped to keep the remaining work more coordinated.

Since a trampoline is a function, whose task is to call a function, it needs to conform to the call procedure standard. In the ARM case it is needed to store variable registers (see complete register listing in figure 2) together with the stack pointer and link registers in the function prologue. A function must always be initialised with a prologue. To restore the registers state before the function returns there is also a function epilogue which cancels out the prologue and sets the program counter to the return address. This is the very last instructions in a function as the next step of the program counter will cause it to return.

Function parameters are passed in registers, revealing ARM's RISC heritage. ARM does however limit these registers to the first four, if there are any following parameters they are passed onto the stack. As the ARM registers are 32

bit data structures any 64 bit value is passed over two registers, if only one should fit, the second half is pushed onto the stack. Smaller values are widened to 32 bits in a way that keeps the sign and range of the value.

A function's return value is passed in the same manner, however a return value may only occupy the four registers at most; any larger data structures must be passed indirectly, in memory, via an address pointer.

```
r15     Program Counter
r14     Link
r13     Stack Pointer
r12     Intra Procedure Scratch
r11     Variable / Frame Pointer
r10     Variable / Stack Limit
r9      Variable / Static Base
r8      Variable
r7      Variable / Thumb Work
r6      Variable
r5      Variable
r4      Variable
r3      Argument / Result / Scratch
r2      Argument / Result / Scratch
r1      Argument / Result / Scratch
r0      Argument / Result / Scratch
```

Figure 2. ARM registers

3.1 The Trampoline's Type Marshaling

Another key issue concerning the trampoline creation that needed further understanding was the marshaling of the different data types. The Common Language Infrastructure supports the Common Type System (CTS) [14] to define data types used. The CTS supports two general categories of types; value types and reference types. Value types are types that directly contain their own data, and they are allocated on the stack or inline in a structure. Value types can either be built-in, user-defined or enumerations. Reference Types do not directly contain their values but are just references to the memory address where the actual data is stored. These types are allocated on the heap and include self-describing types such as arrays or class types (class types can in turn be divided into user-defined classes, boxed value types and delegates), pointer types or interface types.

Structural modifications within the runtime's metadata had rendered the ARM-specific code for trampoline creation unfit. New string marshaling had been introduced to the trampolines and these were essentially the changes that prevented old code from compiling successfully. The adaptations to the new string marshaling gave the result that strings could now directly be passed as references, rather than the previous implementation were the strings were reallocated memory and thus created a certain overhead.

Value types marshaling had also changed, but this did not produce any errors during compilation, instead the process of software debugging commenced. Eventually the source of a number of smaller runtime failures could be traced back to the trampoline; like how it eventually was found that value types were missing a level of indirection and were marshaled by its reference instead of its value, which in turn caused an attempt to access illegal memory. The problem was dealt with by retrieving the reference - the step which had been done all along - and then by simply adding an extra step loading the value from the reference.

The other function to be implemented by all specific architectures, *mono_create_method_pointer*, much like *mono_create_trampoline* creates a pointer to a native function that may be used to call a method - it takes a single MonoMethod parameter to specify this method.

4. Implementation of Mono, the Just-In-Time Compiler

The just-in-time compiler works a little differently than the interpreter and has a wider range of architecture-implemented functions (listed in figure 3). Rather than just creating trampolines and method pointers like the interpreter it also deals with instructions and instruction selections, register allocation and exception handling.

```
mono_arch_output_basic_block
mono_arch_emit_prolog
mono_arch_emit_epilog
mono_arch_patch_code
mono_arch_call_opcode
mono_arch_allocate_vars
mono_arch_local_regalloc
mono_arch_create_jit_trampoline
mono_arch_get_throw_exception
mono_arch_handle_exception
mono_jit_walk_stack
ves_icall_get_frame_info
ves_icall_get_trace
mono_arch_cpu_optimizations
mono_arch_regname
mono_arch_get_allocatable_int_vars
mono_arch_get_global_int_regs
mono_arch_instrument_mem_needs
mono_arch_instrument_prolog
mono_arch_instrument_epilog
```

Figure 3. Mono architecture specific functions

The just-in-time compiler supplies a number of instructions that are easily mapped into processor instructions. For some specific architectures it is necessary to add new instructions that relate more closely, it is also possible to map a sequence of several CPU instructions to a single compiler instruction. An instruction can be defined either as an instruction in a tree representation or as a low-level CPU instruction. Tree instructions are represented with BURG [30] rules while the CPU instructions are specified in a machine description file. The machine description file is processed at compile-time to build a data structure that later easily can be used from C code. The ability to replace one instruction with a sequence is useful and perhaps fortunate as ARM - like older processors as the classical Motorola [31] 650x series [32] [33] - lacks an instruction for division.

By porting the interpreter prior to the just-in-time compiler the native code generation is already implemented and tested, there are also functions for emitting epilogues and prologues. What is new in the function *mono_arch_emit_epilog* is the notion of leaps between managed and unmanaged code, this adds extra exception handling in form of storing all registers and establishing a reference to the last managed frame. As opposed to the interpreter's emissions the just-in-time compiler copies the instructions to an executable memory area after the epilogue. This in turn means that any instructions working with relative memory offsets must have their offsets recalculated.

4.1 Exception Handling

The just-in-time compiler also features a set of sophisticated exception handling routines which do unwinding (the call stack code path might go through both managed and unmanaged code) and calling *catch* blocks. Mono also catches signals from the operating system to generate exceptions; segmentation faults for instance occur when a disallowed operation is attempted on a chunk of memory. This signal is suppressed and relayed with a *NullReferenceException*.

5. Memory Protection

Any procedure call requires a given amount of instructions to pass parameters, handle return values and store away registers. In the trampoline these instructions are emitted to an allocated section of memory. In turn, this means that the memory also needs to be executable in order for the trampoline to be successful. The rights of any allocated memory is initialized by the underlying operating system, and thus vary from platform to platform. In order to gain certainty that a newly acquired memory space is given suitable permissions, it is necessary to actively request for them.

Study showed that protection of memory allocated by the standard function *malloc* was variable depending on the operating system. The behaviour was analysed on two differently configured handheld devices; a Sharp Zaurus [34] running the Linux-based Embedix operating system, and the iPAQ with the Familiar Linux distribution. The analysis effectively revealed that the memory allocated by the Familiar Linux-equipped device allowed read, write and execution, whereas Embedix prevented execution of the memory. It is understood that QNX [35], which also is an embedded operating system, prohibits execution of allocated memory.

The POSIX.1b [36] standard defines the function mprotect for controlling the permissions of memory sections. This function takes three parameters, a pointer to the memory, the size of the memory and the requested permissions for the memory. The definition also states that the memory pointer must be aligned to the memory page size used by the operating system. To find this out POSIX.1 [10] aptly defines the function *sysconf* to read certain system properties, one of them being the page size. A pointer is easily aligned with a standard alignment formula.

```
pointer = (void *) (((int)
          pointer + page_size
          - 1) & ~(page_size - 1));
```

6. Shared Memory

The Mono runtime's IO-layer utilises shared memory for interprocess communication. Memory is mapped onto a file in the filesystem under a specified location inside the users home directory, and the memory is then shared by unrelated processes. This presented itself as problem when attempted together with the JFFS2 [37] file system.

The Familiar Linux distribution uses the JFFS2 file system, a file system implemented by Red Hat based on the design concepts of the prior JFFS. JFFS2 is especially designed for flash devices, and to be reliable even for unpredictable environments such as handhelds or embedded appliances. The filesystem never truncates or alter data in an existing file but always saves the original as a backup; when the new file is successfully written to the original is marked for garbage collecting and is later collected by the file system's built-in garbage collector. It is because of features such as this that JFFS2 does not allow memory sharing mapped onto a file in the file system.

Soon, this problem was the subject of a bug report, and as a result the runtime was made less vulnerable to underlying support of shared memory. Of course, there are other routes to take on this; one possibility would be to use a RAM filesystem to share memory on. The problem was deployment and easy installation of the runtime. There is no general way to know where to find the location of this RAM

filesystem, or even if there are any at all, and to consider all setups and their filesystem layout would prove virtually impossible.

Instead, the decision was taken to leave this unsolved for the time being. It was estimated that it would not be required for a handheld environment, at least not initially in this early stage of development, to support memory sharing for unrelated processes.

7. Atomic Operations

The runtime defines an API for creating architecture-dependent atomic operations. Like many of the implementations for other architectures it was decided to create these operations with inline assembly. It is only with assembly instructions that it is possible to perform operations that take advantage of the CPU's own atomic operations. Mono defines seven functions to use for atomic operations, listed in figure 4.

```
InterlockedCompareExchange
InterlockedCompareExchangePointer
InterlockedIncrement
InterlockedDecrement
InterlockedExchange
InterlockedExchangePointer
InterlockedExchangeAdd
```

Figure 4. Mono's atomic operations

Atomic operations are defined as operations which are uninterruptable and which perform without interaction from any other thread. As the ARM instruction set only contains one truly atomic operation (the swap instruction which swaps two CPU registers and returns the value of the first) the functions that were defined in the runtime had to be implemented by hand. The two functions *InterlockedExchange* and *InterlockedExchangePointer* did not need to be manufactured this way however, as they could directly take advantage of the existing swap instruction.

The other functions needed assure that they could conform to the properties of atomic operations. It was important that they only fullfilled their task if it could be proven that no other thread had intervened or the results would likely wind up invalid. To do this, a small loop was implemented for all functions, the input would get copied, tested and calculations performed. The result would then be swapped back into the input register and at the same time the input register would be checked if it had already changed, as if that was the case another thread must have interrupted the currently executing one and completed another calculation after it had first been copied and before it had been swapped back. In the event of something indeed going wrong, the last calculation would be rolled back, the loop would iterate and the input register would once again be copied, tested and calculated. If no failure was detected, the loop would end and the function return (see Appendix B).

When implementing these functions characteristics of ARM assembly were exercised to gain code paths with less branches and better optimization (see Appendix A). Conditional instructions are instructions which only execute if a certain condition have been met, and are supported by ARM and Itanium [38] architectures. This is faster than branching as branching indirectly causes the CPU's pipeline to stall [39].

8. Packaging for Distribution

It was suggested that binaries of the Mono port should be packaged and made public in similar fashion as the already existing packages for various other Linux distributions. This would provide general testing of the application and stimulate the interest for an open source, .NET compatible runtime for mobile devices. It was decided to use the existing packaging system already in use by both Familiar Linux and Zaurus handhelds - iPKG [40].

iPKG is a lightweight package management system intended for environments with limited resources much like handheld devices, with strong influences from the packaging system used by the Debian Linux distribution [41]. Packages are archived much like their Debian counterparts and are distributed mainly by server feeds.

9. Results

The functionality of the ARM port of Mono can be verified in a number of ways. While it can be discussed with what methods software are ultimately tested and their implementation validated, this exploratory study focused on the proof of concept and real-world usability of the application. To test Mono one of the more complex applications available was executed - the Mono C# compiler, MCS, compiling arbitrary sized C# applications. The compiler exercises a wide range of the frameworks API and is in itself a considerable test for the runtime.

The Mono class library is however mainly written in CIL code and is designed for good integration and reliable execution. GTK# - the GTK glue library - proved to be a good candidate to test delegates - C# language, type-safe method pointers - and platform invocation (commonly referred to as P/Invoke) of native application libraries to a wider extent than that by the compiler. GTK# relies on P/Invoke to directly communicate with the native GTK graphical widget toolkit, the underpinnings of the GNOME desktop. On

the Familiar Linux handhelds, GTK finds its form inside the GPE library. GPE includes and extends the GTK functionality, in a similar fashion to GNOME but aimed towards platforms with far less resources. To compile and run graphical applications with Mono on the iPAQ with Familiar Linux gave the substantial bonus of testing these properties with minimal effort.

9.1 Toolchain Evaluation

During the porting effort two distinctly different toolchains were used, both of the GNU compiler suite. One of the two versions used were GCC 3.2.1 compiling towards a GNU C-library (glibc) of version 2.3.1. This toolchain was published on the Handhelds.org [40] file area and originally intended for the Familiar Linux platform. While this toolchain was initially used, it was quickly acknowledged that the Zaurus devices were shipped with an older version of glibc and that the software were incompatible. Instead a toolchain provided by the OpenZaurus [42] project was acquired, including GCC version 2.95.3 and compiling towards glibc 2.2.3. Several ARM improvements had made it into the GNU compiler between the two versions and by some smaller tests it was possible to determine which one would prevail in producing the more efficient executable (see Appendix C).

The difference in performance between the executables from the two toolchains did differ, although the difference did not prove to be too hefty - the main advantage is for the GCC 3.2.1 toolchain during actual outputting. The analysis was performed without any compile-time optimisations to produce a generic output.

10. Discussion

What has been successfully implemented and is working as a stable application is Mint, the interpreter. The work on the interpreter was limited to trampoline creation and method pointer creation, and were therefore more evident to comprehend and contribute to, and along the way garner knowledge to improve both system design and programming experience. There were minor details regarding memory protection implementations across various systems and also the design and implementation of the atomic operations.

The documentation of the second-generation just-in-time compiler was sparse as the new design had very recently been released. Obviously this did not speed up the learning process. This has recently improved however, several well written guides regarding design and portability have been published which will help future work on ARM, but also on new architectures. As the RISC architecture family is renowned for its similarities it is also possible that this port will help developers support more operating systems on related architectures.

The work, to make Mono run on handheld devices, is a step along the way to make the CLI more universal and make applications compiled for it available across a variety of platforms. This port could likely lead to the design of a smaller framework aimed towards handhelds and embedded devices much like the .NET Compact Framework [13] is today.

The interest for Mono on handheld devices has already started to flare. Hopes are high that Mono will become a common and useful tool, also on small handheld devices.

11. Conclusion

Mint the Mono interpreter is now working and has successfully been run on several devices and running several applications in an attempt to validate its functionality. It has compiled C# source code and run graphical applications with the GTK# library.

12. Acknowledgements

Thanks to Mr. Sergey Chaban for helping out with various issues that came along and for giving valuable feedback, to Mr. Miguel de Icaza and Mr. Paolo Molaro for valuable information and documentation on Mini, to Mr. Richard Torkar and Mr. Steven Kirk, Ph.D., for comments on this paper, and of course to all the other Mono developers.

References

[1] S. Pemberton, "The ergonomics of software porting. automatically configuring software to the runtime environment –or– everything you wanted to know about your C compiler, but didn't know who to ask." in *49*. ISSN 0169-118X: Centrum voor Wiskunde en Informatica (CWI), 31 1992, p. 20. [Online]. Available: citeseer.nj.nec.com/pemberton92ergonomics.html

[2] (2003) Intel Inc. [Online]. Available: http://www.intel.com

[3] "The Intel PXA250 Applications Processor," White Paper, Intel, 2002.

[4] (2003) Hewlett-Packard Inc. [Online]. Available: http://www.hp.com

[5] E. Siever, S. Spainhour, J. P. Hekman, and S. Figgins, *Linux in a Nutshell*. CA, USA: O'Reilly & Associates, Inc., 2000.

[6] (2003) Red Hat Inc. [Online]. Available: http://www.redhat.com

[7] (2003) The Familiar Project. [Online]. Available: http://familiar.handhelds.org

[8] (2003) GNU. [Online]. Available: http://www.gnu.org

[9] (2003) GNU Compiler Collection. [Online]. Available: http://gcc.gnu.org

[10] *Single UNIX Specification*, IEEE-SA Std. 1003.1, 2001.

[11] (2003) The Mono Website. [Online]. Available: http://www.go-mono.org

[12] (2003) Ximian Inc. [Online]. Available: http://www.ximian.com

[13] (2003) .NET. [Online]. Available: http://www.microsoft.com/net

[14] *Common Language Infrastructure (CLI)*, ECMA Std. 335, 2002.

[15] (2003) The Common Language Runtime. [Online]. Available: http://www.gotdotnet.com/team/clr/about_clr.aspx

[16] *C# Language Specification*, ECMA Std. 334, 2002.

[17] (2003) European Computer Manufacturers Association. [Online]. Available: http://www.ecma-international.org

[18] (2003) Microsoft Inc. [Online]. Available: http://www.microsoft.com

[19] (2003) International Organisation for Standardization. [Online]. Available: http://www.iso.org

[20] (2003) The GIMP Toolkit. [Online]. Available: http://www.gtk.org

[21] (2003) QT. [Online]. Available: http://www.trolltech.com/products/qt

[22] (2003) GNOME. [Online]. Available: http://www.gnome.org

[23] (2003) ARM Ltd. [Online]. Available: http://www.arm.com

[24] G. Steele, J. Gosling, and G. Bracha, *Java Language Specification, 2nd Edition*, B. Joy, Ed. MA, USA: Addison-Wesley Pub Co, 2000.

[25] (2003) Windows Embedded. [Online]. Available: http://www.microsoft.com/windows/embedded

[26] (2003) DotGNU Project. [Online]. Available: http://www.dotgnu.org

[27] *The ARM-THUMB Procedure Call Standard*, ARM Ltd. Std. SWS ESPC 0002 B-01, 2000.

[28] (2003) SPARC. [Online]. Available: http://www.sparc.com

[29] (2003) PowerPC. [Online]. Available: http://www.ibm.com/chips/products/powerpc

[30] R. Henry, C. Fraser, and T. Proebsting, "Burg - fast optimal instruction selection and tree parsing," in *SIGPLAN'92, Conference on Programming Language Design and Implementation*, 1997.

[31] (2003) Motorola Inc. [Online]. Available: http://www.motorola.com

[32] L. A. Leventhal, *6502 Assembly Language Programming*. NY, USA: McGraw-Hill Osborne Media, 1986.

[33] J. J. Carr, *6502 User's Manual*. NJ, USA: Prentice Hall PTR, 1984.

[34] (2003) SHARP Zaurus. [Online]. Available: http://www.zaurus.com

[35] (2003) QNX Software Systems Ltd. [Online]. Available: http://www.qnx.com

[36] *POSIX.1b Realtime Extensions*, IEEE-SA Std. 1003.1b, 1993.

[37] D. Woodhouse, "Jffs : The journalling flash file system." [Online]. Available: http://sources.redhat.com/jffs2/jffs2-html/jffs2-html.html

[38] "Intel Itanium Architecture Software Developer's Manual," White Paper, Intel, 2002.

[39] "Intel PXA250 and PXA210 Processors - Optimization Guide," White Paper, Intel, 2002.

[40] (2003) The handhelds.org project. [Online]. Available: http://www.handhelds.org

[41] (2003) Debian GNU/Linux. [Online]. Available: http://debian.org

[42] (2003) OpenZaurus. [Online]. Available: http://www.openzaurus.org

A. Atomic Operations Source Code

```
static inline gint32
InterlockedCompareExchange(volatile gint32 *dest, gint32 exch, gint32 comp)
{
        int a, b;
        __asm__ __volatile__ (    "0:\n\t"
                                  "ldr %1, [%2]\n\t"
                                  "cmp %1, %4\n\t"
                                  "bne 1f\n\t"
                                  "swp %0, %3, [%2]\n\t"
                                  "cmp %0, %1\n\t"
                                  "swpne %3, %0, [%2]\n\t"
                                  "bne 0b\n\t"
                                  "1:"
                                  : "=&r" (a), "=&r" (b)
                                  : "r" (dest), "r" (exch), "r" (comp)
                                  : "cc", "memory");
        return a;
}
static inline gpointer
InterlockedCompareExchangePointer(volatile gpointer *dest,
                                  gpointer exch, gpointer comp)
{
        gpointer a, b;
        __asm__ __volatile__ (    "0:\n\t"
                                  "ldr %1, [%2]\n\t"
                                  "cmp %1, %4\n\t"
                                  "bne 1f\n\t"
                                  "swpeq %0, %3, [%2]\n\t"
                                  "cmp %0, %1\n\t"
                                  "swpne %3, %0, [%2]\n\t"
                                  "bne 0b\n\t"
                                  "1:"
                                  : "=&r" (a), "=&r" (b)
                                  : "r" (dest), "r" (exch), "r" (comp)
                                  : "cc", "memory");
        return a;
}
static inline gint32
InterlockedIncrement(volatile gint32 *dest)
{
        int a, b, c;
        __asm__ __volatile__ (  "0:\n\t"
                                "ldr %0, [%3]\n\t"
                                "add %1, %0, %4\n\t"
                                "swp %2, %1, [%3]\n\t"
                                "cmp %0, %2\n\t"
                                "swpne %1, %2, [%3]\n\t"
                                "bne 0b"
                                : "=&r" (a), "=&r" (b), "=&r" (c)
                                : "r" (dest), "r" (1)
                                : "cc", "memory");
```

```
        return b;
}
static inline gint32
InterlockedDecrement(volatile gint32 *dest)
{
        int a, b, c;
        __asm__ __volatile__ (  "0:\n\t"
                                "ldr %0, [%3]\n\t"
                                "add %1, %0, %4\n\t"
                                "swp %2, %1, [%3]\n\t"
                                "cmp %0, %2\n\t"
                                "swpne %1, %2, [%3]\n\t"
                                "bne 0b"
                                : "=&r" (a), "=&r" (b), "=&r" (c)
                                : "r" (dest), "r" (-1)
                                : "cc", "memory");
        return b;
}
static inline gint32
InterlockedExchange(volatile gint32 *dest, gint32 exch)
{
        int a;
        __asm__ __volatile__ (  "swp %0, %2, [%1]"
                                : "=&r" (a)
                                : "r" (dest), "r" (exch));
        return a;
}
static inline gpointer
InterlockedExchangePointer(volatile gpointer *dest, gpointer exch)
{
        gpointer a;
        __asm__ __volatile__ (  "swp %0, %2, [%1]"
                                : "=&r" (a)
                                : "r" (dest), "r" (exch));
        return a;
}
static inline gint32
InterlockedExchangeAdd(volatile gint32 *dest, gint32 add)
{
        int a, b, c;
        __asm__ __volatile__ (  "0:\n\t"
                                "ldr %0, [%3]\n\t"
                                "add %1, %0, %4\n\t"
                                "swp %2, %1, [%3]\n\t"
                                "cmp %0, %2\n\t"
                                "swpne %1, %2, [%3]\n\t"
                                "bne 0b"
                                : "=&r" (a), "=&r" (b), "=&r" (c)
                                : "r" (dest), "r" (add)
                                : "cc", "memory");
        return a;
}
```

B. Atomic Operation Flow Analysis

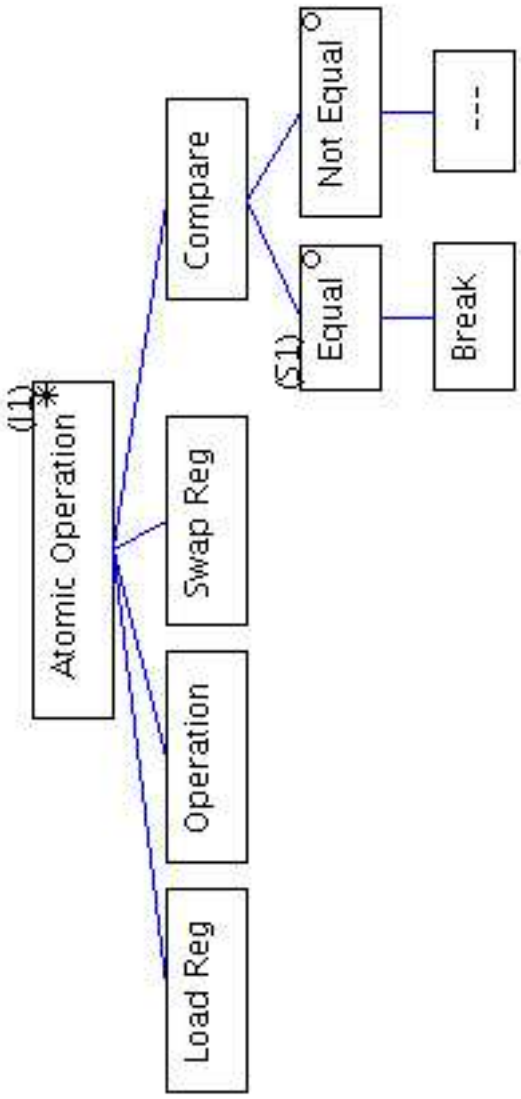

Figure 5. JSP diagram of atomic operations

C. Toolchain Comparison

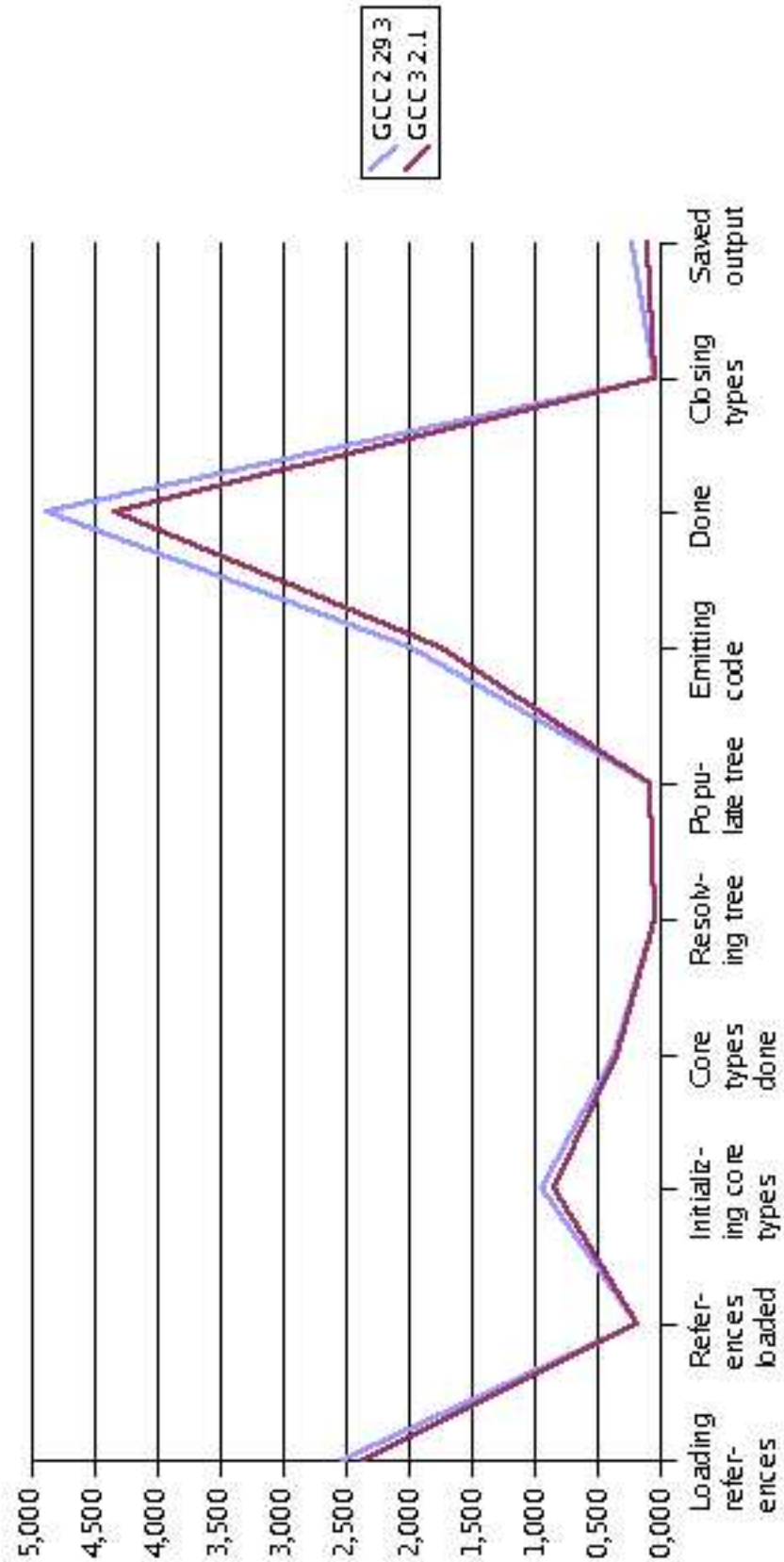

Figure 6. Graph