Automatic Architecture Description Language (ADL)-Based Toolchain Generation: The Dynamic Linking Framework

Rafael Auler¹, Alexandro Baldassin¹, Paulo Cesar Centoducatte¹

¹Computer Systems Laboratory – Institute of Computing – University of Campinas
P.O. Box 6176 – 13.083-970 – Campinas – SP – Brazil
rafael.auler@lsc.ic.unicamp.br, {alebal, ducatte}@ic.unicamp.br

Abstract. In order to succeed in the fulfillment of clients ever increasing expectations, embedded systems companies are entering the area of multitasking environments. As memory is still a primary concern to achieve a good tradeoff between system performance and cost, we cannot rule out dynamic linking techniques since they reduce memory consumption at runtime. Using parameters from an ADL model of the processor, it is possible to generate a complete software toolchain for the platform under development, including simulators and binary tools. In this paper we focus on the elaboration of the dynamic linker component of our work-in-progress compiler toolchain generator. More precisely, we describe the modifications to the platform loader and linker so as to support dynamic linking. We present a case study for the ArchC ADL and show how a complete dynamic linking environment is generated for the ARM architecture based on its ADL model, followed by an analysis of the performance impact.

1. Introduction

The increasing level of integration has made it possible for industry to build entire systems on a single die, giving rise to the System-on-a-Chip (SoC) revolution. This trend has generated demand for sophisticated tools to enable whole system simulation at a high level of abstraction. To address this problem, researchers have proposed the use of Architecture Description Languages (ADLs) [Mishra and Dutt 2008], capable of describing a design at high level of abstraction and, at the same time, being flexible enough to allow design space exploration.

ADL tools are not restricted to the simulator generation, but also include the synthesis of tools for software development such as assemblers, linkers and even compilers. Only with these tools the designer actually has a thorough support to effectively assess an integrated processor design, since a simulator is of limited use if there is no support to develop the software for the system. Speed is exactly the emphasis put into the initial stages of designing a hardware platform, when the hardware team must supply a functional model of simulators and tools to allow the software team to begin production early in the development process.

The ArchC project [Azevedo et al. 2005] provides tools whose role is, given an ADL processor model, to produce simulators, assemblers, linkers and debuggers. The generated simulators are compatible with the SystemC language [Grotker 2002], which means the processor module may be integrated as part of a bigger platform. The ArchC extension for automatic synthesis of assemblers is called acbingen [Baldassin et al. 2008],
and enriched the description capability of the ArchC architectural model with the possibility of assembly language syntax definition for a specific Instruction Set Architecture (ISA). Based on this information, an assembler can be automatically generated for a target architecture. The generation of binary tools relies on the GNU binutils framework [Pesch and Osier 1993], allowing the target machines to share the same interface.

The acbingen tool is also capable of automatically synthesizing simple linkers for a given target architecture properly described in ArchC. Such linkers are restricted to put together different portions of relocatable code to build the final executable and can only handle static libraries. While static libraries are still adequate for most uses in a simple software environment, we cannot restrict ourselves to this scenario when targeting a more complex platform. For example, in a multitask system where the amount of duplicated code is large, it is of paramount importance to have support for dynamic libraries. This duplication arises from common code when the same libraries are used in two or more processes. Thus, it is necessary to use linkers that handle shared dynamic libraries to reduce main memory usage. In order to use dynamic libraries, it is also important to have simulators with the ability to load and manage shared libraries.

The goal of this paper is to enable the automatic synthesis of dynamic linkers for an ArchC modeled target. A lot of the work spent in the dynamic linking method is deferred until runtime. Therefore, the second part of this work is devoted to adapt the previous ArchC simulator loading mechanism to understand dynamic libraries dependencies. To do this, not only it is necessary to find and load the requested libraries, but it is also necessary to perform runtime binding of undefined symbols in the application with defined symbols in the library. Finally, we must dissociate the machine-dependent part of the process to easily identify the information needed to retarget the simulator to an arbitrary target (in fact, any target described with the ArchC language). As a guide to this task, the Application Binary Interface (ABI) defines much of these machine dependent details and it is useful to identify where ABI information is being used in the linker-loader system.

We make the following contributions:

- Identification of the relevant information needed as parameters in an ADL model to enable dynamic linking tools retargeting;
- Creation of a dynamic linker to the ArchC work-in-progress compiler toolchain generator, and preparation of generated simulators with the capacity of loading target machine dynamic libraries;
- Support for the GNU C library (glibc) [Free Software Foundation 2009] as the official C library for the future production quality compiler toolchain generator in the ArchC framework. This is a major step compared to the previous version, which was based on Red Hat newlib [Red Hat 2008], since it is a considerably more complete implementation, used even in desktops.

The remainder of this paper is organized as follows. Section 2 provides related work. Section 3 presents an overview regarding the whole system. Section 4 discuss the compile-time linker. Section 5 describes motivation and implementation for the runtime linker and loader. Section 6 presents our experimental results, and Section 7 concludes.
2. Related work

Static shared libraries appeared as the first solution to the problem of code (or library modules) sharing among programs. The address space management for the system that supports this kind of library is rigid and must be well known at application compile time, thus complicating software projects and maintenance. Also, current off-the-shelf operating systems do not support this approach.

The use of dynamic libraries, enabled by a Memory Management Unit (MMU) for virtual addresses, allows flexible use of instruction memory, supporting on-demand loading and shared memory among multiple processes. But the implementation of dynamic linker is not trivial and adds a significant degree of complexity to the linker.

The problem of automatic synthesis of compilers based on ADL models has been extensively explored since the decade of 1990, and its foundations are based on earlier research on code generator generation. This large task has been addressed by the projects LISA [Hoffmann et al. 2002], Flexware CodeSyn [Liem et al. 1994, Liem et al. 1995], AVIV [Hanono and Devadas 1998], RECORD [Leupers and Marwedel 1997], CHESS [Lanneer et al. 1995], EXPRESSION [Halambi et al. 1999] and MADL [Qin et al. 2004]. Our work explores the related topic of binary tools retargeting based on ADL models, a necessary step to provide support for the generation of a compiler suite. Specifically, we analyze the dynamic linking support.

Papers addressing compiler retargetability using the LISA language [Hohenauer et al. 2004, Ceng et al. 2005] mention its capability of generating assembler and linker, but omit further details. This is similar to the other ADLs aforementioned. In contrast, Casarotto et al. [Casarotto and dos Santos 2006] reports precisely how linker retargetability was achieved using the ArchC ADL, and the idea was also enhanced to an ADL independent approach [Baldassin et al. 2007], although this technique is limited to static linkers.

The application of machine-independent linkers has also been discussed in the field of executable editing tools, such as ATOM [Srivastava and Eustace 1994], EEL [Larus and Schnarr 1995] and DIABLO framework [Van Put et al. 2005]. The latter was also used in memory footprint reduction [De Bosschere 2008]. However, the reported solution focuses in size optimization and differs from the runtime approach proposed by our dynamic linker.

Lee et al. [Lee et al. 2006] show valuable information concerning memory reduction using shared libraries in embedded environments, and memory reduction of up to 35% was reached. Their technique was adapted to MMU-less embedded systems, hence it differs from ours as it cannot use dynamic linking and is targeted at simpler environments. This suggests that in more complex embedded systems where a MMU is available our approach is also valid.

3. Overview

We present a block diagram of our long-term goal in Figure 1, a production quality toolchain generated for an ADL modeled target. The C source code is handled by the compiler frontend, llvm-gcc of the LLVM framework [Lattner and Adve 2004]. This
frontend is itself a retargeted gcc to produce LLVM intermediate code. At that instant, the \texttt{llc} backend generates target assembly source, handled by the assembler and linker. The work presented in this paper enhances the binary tools support for dynamic linking in the retargeted GNU assembler and GNU linker. Since we do not have yet the compiler generator, for validation purposes we substitute the LLVM based compiler framework with a gcc cross compiler for the target platform. More details are given in Section 6.

Figure 2 presents a more detailed view of the whole dynamic linking system and includes also the simulator for running the toolchain produced binaries. The system design is split into three distinct subsystems, which integrate into the general dynamic linking system. These three subsystems also communicate with the ArchC subsystem to obtain architecture-dependent information. The compile-time linker (b), runtime linker (c) and ArchC parts (d) are depicted in the diagram. To help the comprehension of the whole system, the ArchC generated assembler (a) was also included in the diagram. Its role is to retarget GNU assembler [Elsner 1993], that is, produce an assembler capable of generating object code for the target machine. For further details we refer the reader to the technical report [Auler et al. 2009].

In the diagram, dotted boxes represent products and processes inputs. Regular boxes represent processes. This paper examines the details of the “linker synthesis logic”, contained in the compile-time linker subsystem and the acrtld (ArchC runtime linker and loader), contained in the runtime linker subsystem. This first subsystem processes a linker script and object files from the assembler, creates shared libraries and executables, which may depend on other shared libraries. The second subsystem, acrtld, loads executables and shared libraries to enable its simulation by using the ArchC generated simulator. The simulator will interpret target machine (described in the model analyzed by the ArchC frontend) instructions loaded into memory.

The bfd library (libbfd) plays a major role in the compile-time linker subsystem. Its responsibility is to integrate the handling of different object files seamlessly, in a manner independent of the object file format and machine of the individual inputs to produce the linked executable. The GNU linker program, \texttt{ld}, can be seen as a script frontend to
use libbfd, which, in turn, does all the linking tasks. The library may be retargeted by writing a backend which extends the linking engine with a new pair `<machine-object file format>`. Part of the techniques described in this paper focuses in creating a new libbfd backend for a pair `<arbitrary ArchC machine-ELF>`. We chose Executable and Linking Format (ELF) as the object file format by its notorious majority in the Linux world, its inherent extensibility and support for dynamic linking.

4. Compile-time linker

The compile-time linker subsystem is responsible for automatically generating a complete linker for an arbitrary machine described in an ArchC model. The generated application is an ordinary linker that also can generate shared libraries and link regular objects against shared libraries. In the latter case, we have an executable that depends on shared library code. We use the term “compile-time linker” to distinguish between the other part, the “runtime linker and loader”, although these parts may also be simply called “linker” and “loader”.

4.1. Relocations

Relocations are annotations used by the linker to patch the code when a section is repositioned in the final executable. Relocations are as important to a linker as instructions are to an assembler. In the context of this paper we distinguish between static and dynamic relocations.

Static relocations are created by the assembler when generating a relocatable object file, and must be fully processed by the linker when the final executable is built. Therefore, the final executable must not contain any static relocations. By contrast, dynamic relocations are generally simpler than static relocations because the related object files are already processed by a compile-time linker. Dynamic relocations only exist in dynamic libraries or final executables linked against a shared library. Consequently, to
be able to extend a linker with dynamic library generation capability, it is necessary to define new relocation types for the ArchC linker (that is, the linker generated by ArchC) in addition to the existing static ones [Baldassin et al. 2007].

Nevertheless, there are only a few dynamic relocations needed and they are not related with any machine dependent encoding, because they always patch pointers to code or data, not a specific instruction. For this reason, regarding relocation types, it is only indispensable to known the pointer size for the target architecture. In the case of ArchC, this information is inferred by ac_wordsizes directive. There is a reasonable explanation why dynamic relocations patch only pointers, and therefore simplify our general purpose relocation design: all data accesses gains an extra level of indirection (it is necessary to dereference a pointer to access data) when using Position Independent Code (PIC) necessary in dynamic libraries, and this indirection implies explicit pointers.

ELF object file organization into sections and relocations is illustrated in Figure 3. In this diagram, the relocation is represented by the struct Elf32_Rela. This is the name used in the header file elf.h, which is available in the Linux kernel headers. The field r_offset has 32 bits and stores the virtual address (and not the file address) of the target, that is, of the instruction to be patched. The field r_info has the same size and has two kinds of information: the referred symbol whose address is unknown at relocation creation time (stored as an index to an element of the symbol table, contained in a special ELF section) and the relocation type. Each relocation type has a code, called relocation code, which is defined by the ABI of each architecture. Because these codes are machine dependent, they complicate our retargetable design (the solution is discussed in section 5.2 and uses a conversion tool when necessary). The last field stores the addend, or offset, applied to the virtual address of the referred symbol. This addend is frequently omitted with the use of the struct Elf32_Rel. In this last approach, the addend is stored directly in the relocation target as a precomputed value, but is not used in this paper.

4.2. PLT and GOT

In order to understand the machine dependent information needed to retarget the compile-time linker, it is important to review the basic mechanism which represents how dynamic symbol binding is actually implemented in ELF object files.
In the Section 4.1, we briefly exposed the concept of PIC and how an extra level of indirection is introduced to enable the deferred linking. In order to segregate read-only pages, which may be shared among multiple processes, and global data read-write pages, which are shared copy-on-write, ELF uses two special sections: PLT and GOT. The GOT, or Global Offset Table, includes several entries with the size of a pointer (a virtual address) and is referred by PLT when the shared library needs to call a dynamic bound function or directly by PIC code when the shared library accesses global data. These entries contain the addresses of the objects whose exact location will only be known at runtime, when the library sections are positioned in the process address space.

PLT, or Procedure Linkage Table, is an extra code section (and thus, read-only) created specially by the linker when a program calls a function located in a dynamic object (pertaining to a dynamic library). Its contents are small target machine code fragments. This code loads an address from the GOT section and jumps to it, completing the function call. This is the reason why all dynamic relocations patch only the GOT table, independently of the access type being code or data.

To summarize, Figure 4 shows a diagram of a call to the `printf` function when a program is linked with a dynamic version of the C library. The dynamic relocation section contains an annotation instructing the runtime linker to patch the GOT table with the address of the symbol `printf`. When the C library is loaded, its dynamic symbol table defines the so far unknown symbol `printf`. The runtime linker patches the executable GOT with printf address, and the program call will jump to the PLT. As explained, the PLT code will load the now patched address from GOT and jump to the correct location of the `printf` function.

In the context of binary tools retargetability, it is necessary to know, for an arbitrary architecture, how the PLT is built if we plan to generate a dynamic linker for the target. To compose the GOT, we already have the pointer size information necessary for relocations. Our framework requires an additional file describing what instructions are used in the PLT, which are usually part of the system ABI. This semi-automatic approach
enables a correct dynamic linker to be derived from the architecture description, in contrast to a fully automated method for guessing a functional code sequence to build a jump table in the target architecture.

4.3. Libbfd backend generation characteristics

After the target dependent knowledge is extracted from the machine model, the next step is to generate a complete libbfd backend in C code. The binary file descriptor (bfd) library accomplishes all linking related tasks, including dynamic library creation and regular object linking against dynamic libraries, and constitutes the GNU linker core. It is mandatory to inform libbfd how to deal with a new target, and this is done through the addition of a new backend to the source tree.

Our generated backend is not restricted in functionality and has all the features of other regular backends, including dynamic symbol versioning. This was necessary to support glibc [Free Software Foundation 2009], the GNU C Library, which is very demanding on the linker and was our primary objective for testing and validation. The basis for this decision is because the C library is large and cannot be neglected when targeting memory footprint reduction.

5. Runtime linker and loader

5.1. Motivation

While the compile-time linker described in Section 4 deals with the generation of the binary files, the runtime linker and loader (or simply “loader”) is used in the software execution and testing phases of the design flow. This component is necessary for program simulation when a complete platform comprising the operating system is not yet available. The ArchC generated simulator typically runs Linux user mode programs by servicing system calls and redirecting them to the host operating system. If the user mode program, available in an ELF object file, is linked with a shared library, then it is necessary to have a runtime linker and loader.

The GNU runtime linker and loader used in GNU/Linux systems is itself a special form of shared library, called ld.so. Hence, there are two ways of handling the problem of dynamically linked user mode program simulation. The first is to mimic the Linux conventional loader: to load an available target machine ld.so and transfer control to the library entry point. If the simulator correctly respond to the necessary system calls, the target machine ld.so will alone load and link all the necessary dynamic libraries. Although we can simulate the environment with more precision because of the use of the target linker library, the problem with this approach is the availability of ld.so for the target machine. If there is still no Linux port to the new architecture, then most likely there will not be any ld.so version either.

The second way, used by our approach, leverages the simulator loader with a complete runtime linker. It is important to stress that the code of the runtime linker and loader, therefore, is not compiled specifically for the target machine, but is instead executed by the host as auxiliary functions to the simulator. This method guarantees simulation independent of target and library availability.
5.2. Acrtld

Acrtld is the ArchC runtime linker and loader and is implemented as an auxiliary module of the ArchC simulator. It does not perform lazy binding because this technique is used primarily to improve runtime performance. In our method, we perform symbol binding in the simulator host at program load time. Still, the method for finding the requested dynamic libraries differs from the conventional one. We use a environment variable called \texttt{AC\_LIBRARY\_PATH} to avoid name conflicts with the host dynamic libraries. Another difference in the conventional method is the use of a vector of information passed as argument to the \texttt{ld.so} module. In acrtld, the information is obtained using ArchC methods. The general algorithm used in the ArchC linker and loader was not based in GNU \texttt{ld.so} source code, but aims at reproducing its behavior.

As discussed in section 4.1, we define our own relocation codes, incompatible with the different ABIs for each architecture. Hence, the dynamic libraries produced by our compile-time linker will not directly run in a target operating system because it does not comply with the ABI. To solve this problem, it was also developed a simple relocation code conversion tool. In fact, acrtld automatically detects when it loads original dynamic libraries, which does not contain ArchC expected relocation codes, and converts them. Due to this feature, ArchC runtime linker and loader can intermix executables produced by its own linker with shared libraries created using the ABI compliant original linker.

6. Experimental results

In order to show the validity of our approach and the correctness of the designed components, an ARM architecture model was used to generate a compile-time linker and the simulator, which contains the created runtime linker and loader. We chose to perform separated tests in each one of the two components, henceforth called simply linker and simulator.

Since we still do not have the compiler generator to produce an ARM compiler and use it to test the linker as in the intended scenario, we used gcc targeted to ARM. The validation of the generated ARM linker was done by comparing its output with a preexisting ARM linker available in the GNU binutils package. We used a gcc cross compiler version 3.4.5 for ARM with glibc 2.3.2 and binutils 2.15. In the cross compiler generation process, we replaced the original ARM assembler and linker with our own, creating a complete toolchain based on our generated tools.

To certify the simulator module, a preexisting gcc cross compiler version 3.4.5 for ARM was used to compile MiBench [Guthaus et al. 2001] programs linked against the dynamic glibc version 2.3.2. Table 1 and 2 list the MiBench programs evaluated. For each program, we show the number of instructions executed and the simulator speed (in millions of instructions per second) for two different input sizes: small and large. Table 1 refers to programs linked with the static libraries, whereas Table 2 refers to programs linked against the dynamic libraries produced with our approach.

Using these two tables as reference, we can observe two consequences of using dynamic libraries. Firstly, all programs had their number of instructions increased. This occurs because of the indirect referencing used in all global data accesses, as well as the
<table>
<thead>
<tr>
<th>Program</th>
<th>Small input # instructions</th>
<th>MIPS</th>
<th>Large input # instructions</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick Sort</td>
<td>34,946,894</td>
<td>10.58</td>
<td>128,544,142</td>
<td>8.93</td>
</tr>
<tr>
<td>Susan (Corners)</td>
<td>1,458,659</td>
<td>8.53</td>
<td>24,040,456</td>
<td>9.42</td>
</tr>
<tr>
<td>Susan (Edges)</td>
<td>2,941,524</td>
<td>6.30</td>
<td>74,635,179</td>
<td>9.59</td>
</tr>
<tr>
<td>Susan (Smoothing)</td>
<td>19,607,777</td>
<td>12.25</td>
<td>270,139,639</td>
<td>11.43</td>
</tr>
<tr>
<td>Basic Math</td>
<td>295,293,692</td>
<td>7.83</td>
<td>3,084,424,516</td>
<td>8.49</td>
</tr>
<tr>
<td>Bit Count</td>
<td>43,659,272</td>
<td>10.83</td>
<td>653,664,671</td>
<td>11.11</td>
</tr>
<tr>
<td>CRC32</td>
<td>65,753,584</td>
<td>9.55</td>
<td>1,278,150,006</td>
<td>9.84</td>
</tr>
<tr>
<td>ADPCM Coder</td>
<td>28,491,221</td>
<td>9.85</td>
<td>565,641,848</td>
<td>8.98</td>
</tr>
<tr>
<td>ADPCM Decoder</td>
<td>21,597,849</td>
<td>8.85</td>
<td>424,035,884</td>
<td>8.65</td>
</tr>
<tr>
<td>FFT</td>
<td>123,362,089</td>
<td>8.60</td>
<td>1,545,321,883</td>
<td>8.79</td>
</tr>
<tr>
<td>FFT Inv</td>
<td>216,291,416</td>
<td>8.87</td>
<td>1,394,091,519</td>
<td>8.64</td>
</tr>
<tr>
<td>GSM Decoder</td>
<td>14,540,897</td>
<td>7.50</td>
<td>791,551,892</td>
<td>9.80</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>53,148,886</td>
<td>9.82</td>
<td>244,912,043</td>
<td>9.85</td>
</tr>
<tr>
<td>Patricia</td>
<td>79,208,076</td>
<td>5.32</td>
<td>488,577,410</td>
<td>5.29</td>
</tr>
<tr>
<td>SHA</td>
<td>14,203,153</td>
<td>11.18</td>
<td>147,836,401</td>
<td>10.80</td>
</tr>
<tr>
<td>LAME MP3 Encoder</td>
<td>2,162,444,271</td>
<td>9.44</td>
<td>26,658,678,088</td>
<td>10.04</td>
</tr>
</tbody>
</table>

Table 1. Simulation and performance results for an ARM platform using statically linked code.

<table>
<thead>
<tr>
<th>Program</th>
<th>Small input # instructions</th>
<th>MIPS</th>
<th>Large input # instructions</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick Sort</td>
<td>36,484,522</td>
<td>9.91</td>
<td>132,122,771</td>
<td>8.96</td>
</tr>
<tr>
<td>Susan (Corners)</td>
<td>1,469,704</td>
<td>7.34</td>
<td>24,051,554</td>
<td>9.32</td>
</tr>
<tr>
<td>Susan (Edges)</td>
<td>2,957,142</td>
<td>9.85</td>
<td>74,836,512</td>
<td>9.49</td>
</tr>
<tr>
<td>Susan (Smoothing)</td>
<td>19,620,449</td>
<td>10.72</td>
<td>270,153,157</td>
<td>11.29</td>
</tr>
<tr>
<td>Basic Math</td>
<td>300,064,701</td>
<td>7.74</td>
<td>3,175,910,276</td>
<td>7.97</td>
</tr>
<tr>
<td>Bit Count</td>
<td>43,658,393</td>
<td>11.22</td>
<td>653,663,882</td>
<td>11.26</td>
</tr>
<tr>
<td>CRC32</td>
<td>93,138,162</td>
<td>9.35</td>
<td>1,810,523,030</td>
<td>9.08</td>
</tr>
<tr>
<td>ADPCM Coder</td>
<td>28,496,952</td>
<td>9.25</td>
<td>565,799,031</td>
<td>9.65</td>
</tr>
<tr>
<td>ADPCM Decoder</td>
<td>21,603,580</td>
<td>10.04</td>
<td>424,193,067</td>
<td>9.31</td>
</tr>
<tr>
<td>FFT</td>
<td>124,890,445</td>
<td>8.81</td>
<td>1,560,843,885</td>
<td>8.26</td>
</tr>
<tr>
<td>FFT Inv</td>
<td>218,688,089</td>
<td>8.80</td>
<td>1,406,845,274</td>
<td>8.96</td>
</tr>
<tr>
<td>GSM Decoder</td>
<td>14,970,178</td>
<td>9.72</td>
<td>814,912,612</td>
<td>10.44</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>54,285,442</td>
<td>9.82</td>
<td>247,900,888</td>
<td>10.15</td>
</tr>
<tr>
<td>Patricia</td>
<td>83,597,966</td>
<td>5.46</td>
<td>514,558,217</td>
<td>5.45</td>
</tr>
<tr>
<td>SHA</td>
<td>14,223,630</td>
<td>11.47</td>
<td>148,051,817</td>
<td>11.79</td>
</tr>
<tr>
<td>LAME MP3 Encoder</td>
<td>2,165,948,958</td>
<td>9.37</td>
<td>26,696,180,850</td>
<td>9.25</td>
</tr>
</tbody>
</table>

Table 2. Simulation and performance results for an ARM platform using dynamically linked code.

extra indirection level added to function calls to dynamic library code. Also, it is important to remember that dynamic library code is position independent and, to support this feature, compilers often dedicate a register to hold the global offset table address for the entire execution time of PIC code. This affects performance by reducing the total number of registers available for allocation. Secondly, it should be noted that dynamic linking impacts performance unevenly for different programs. CPU intensive applications, like ADPCM, GSM and, most notoriously the LAME MP3 encoder, had a small relative increase in the number of executed instructions. This happened because a major part of
the time spent in the execution was in internal functions, statically linked. In these applications, few glibc calls are made because the internal processing time is much bigger than standard library calls (like input/output) operations time. On the other hand, CRC32 increased the number of executed instructions by 40% when compared to its statically linked counterpart. This application heavily depends on I/O glibc functions which are in dynamic objects (compiled as PIC code and always dereferencing pointers to all function calls and global data accesses).

Finally, to measure the real benefits of using shared libraries, Figure 5 shows three charts for page sharing in different scenarios. The first one, Figure 5a, analyzes several glibc shared libraries exposing the percentage of bytes used in shared pages versus bytes in Copy-On-Write (COW) pages. COW is used in shared libraries data pages and its strategy is to initially share the page and, when some data needs to be written, the process receives an exclusive copy of the page. In our conservative analysis, we consider these pages to be not shared. These libraries have different sizes, but in total, about 80% of all glibc 2.3.2 shared libraries bytes is safely shareable. This analysis is static since it does not take into account dynamically allocated private data pages at runtime, which are not shareable. To analyze this scenario, the brk and sbrk syscalls (used by glibc malloc to request page allocation) were modified to record the amount of memory requested. Figure 5b and Figure 5c show the memory consumption profile at runtime for different MiBench applications using dynamic libraries, discriminating shareable and non-shareable bytes. The first uses small inputs and the last one uses large inputs.

In all cases of the dynamic analysis, the size of shareable memory pages exceeds the size of private data, with the exception of patricia for large inputs. This happens
because the Patricia trie [Guthaus et al. 2001], the data structure used in patricia for network routing, albeit being designed to compress sparse trees, uses too much space for the large input processing. Nevertheless, these results emphasize the potential for memory usage reduction and thus the need to support these techniques in the generated toolchain.

7. Conclusion

We have presented the development of mechanisms to enable automatic generation of a linker with dynamic library creation capability, based on ADL models. We also presented necessary modifications to the loader of an ADL based simulator so that it can run dynamically linked programs. Our ideas were tested and implemented using the ArchC ADL and the libbfd library available in the GNU binutils suite to build the linker.

We further presented experimental results using the MiBench benchmark for the ARM architecture. The results validates the proposed tools and show the importance of supporting dynamically linked libraries in platforms where memory consumption is critical. It is also important to point that all the software developed is available as free software and is part of the ArchC project, available in the official site\textsuperscript{1}.

8. Acknowledgments

This work was sponsored by FAPESP (grant 08/53624-3). We also thank Bruno Cardoso, Maxwell Garcia, Felipe Klein, Daniel Nicacio, Joao Correa, Raoni Fassina, Leonardo Piga, Leonardo Ecco and Rodolfo Azevedo for their comments and opinions.

References


\textsuperscript{1}www.archc.org


