FP7-215216

Architecture Paradigms and Programming Languages for Efficient programming of multiple COREs

Specific Targeted Research Project (STReP) THEME ICT-1-3.4

Core compiler

Deliverable D5.4, Issue 1.0

Workpackage WP5

Author(s):	Raphael Poss		
Reviewer(s):	Chris Jesshope		
WP/Task No.:	WP5	**Number of pages:**	26
Issue date:	2009-05-28	**Dissemination level:**	Confidential

Purpose: The purpose of this deliverable is to document the strategy and implementation of a a compiler for the intermediate language used as a target by the other Apple-CORE parallelizing compilers, especially to the target Microgrid architecture. It documents the approach taken in the delivery of this compiler and the advantages gained in providing a common interface to a range of development tools.
Results: The main results of this deliverable are a set of tools, a unit test suite, a language specification and additional documentation that facilitate development across a range of SVP implementations, including a compiler for the target Microgrid architecture.
Conclusion: A common intermediary language (SL) was defined, which is automatically translated and compiled to multiple targets. This allows cross-testing of programs across several SVP implementations, validates the SVP model as a general-purpose programming model, and allows benchmarking efforts to continue in Apple-CORE.

Approved by the project coordinator: Yes **Date of delivery to the EC:** 2009-05-28

Document history

When	Who	Comments
2009/05/18	Raphael Poss	Initial version
2009/05/19	Raphael Poss	Updated purpose and motivation

SEVENTH FRAMEWORK PROGRAMME

Project co-funded by the European Commission within the 7th Framework Programme (2007-11).

Table of Contents

1 Compiling for the Microgrid

1.1 Goals and motivation

The primary goal of the work described in the rest of this document was to close the gap between the NWO Microgrid and AETHER projects, on one side, and the initialization of Apple-CORE created by the delayed delivery of the Microgrid core compiler on the other side; i.e. the initial unavailability of a compiler to generate code from the announced intermediate language, μTC, to the Microgrid architecture.

The lack of a ready-to-use compiler at the start of Apple-CORE caused the following uncertainties:

- from an experimental perspective, this prevented the actual testing and benchmarking activities that were announced in other work packages;
- from a research perspective, this prevented further analysis of programs at a high-level, by lack of either a complete specification of the language and an implementation to serve as reference;
- from an engineering perspective, this prevented planning the software integration between the various tools and components in the project.

These uncertainties were recognised by the management team during the first year of the project, acknowledged by the reviewers during the first review and resources were allocated to determine a contingency course of action.

In addition to this position, an additional analysis in Apple-CORE has yielded the following secondary requirements which also justified supplemental compiler work within the Apple-CORE project itself:

- as an opportunity for model verification and testing of simulations, it is desirable to provide a toolchain which is able to retarget source programs expressed in the intermediatry language to *multiple back-end implementations* where the programs can run, especially to a deterministic environment where a sequential scheduling of programs can serve as a reference run to compare observable behaviors;
- as the hardware design allows for a large parameter space, and the code generation algorithms themselves can be tuned for different optimization strategies, it is desirable to provide tools that can easily *generate and run* the same input code across a wide range of parameters and efficiently compare the compilation output; in particular, all parameters should be *exposed at run-time* to allow for efficient *automation* of tests without lengthy build cycles;
- in order to respond quickly to evolutions of the FPGA prototype design, the code generation and optimization in the core compiler must exhibit a flexible design, where *modularity* and *extensibility* allow for the addition of new features or testing of alternate algorithms at a low engineering cost.

As a compiler framework exclusively based on GCC, the Microgrid core compiler does not, alone, match these requirements. In particular, while the GCC framework is modular and flexible, both the addition of new features and the duplication and modification of existing components require expensive engineering efforts. Also, experimentation across parameter spaces is made difficult since GCC — as most other compiler frameworks — is designed so that a single compiler build generates code for a single architecture with a set of designed parameters fixed at build time. Exposing build-time parameters as run-time parameters in GCC and the Microgrid core compiler, in particular at the level of assembly code generation, has also proved to be a major engineering effort.

1.2 Strategy and solutions

The reaction to the points described above was a combination of three decisions:

1. *encapsulate* the Microgrid core compiler, as well as other code generation tools, in an additional software layer which provides an *abstraction to the actual code generation* through a single interface with the higher-level compilers;

2. *capture μTC's language extensions to C* into a *different syntax* with no change in semantics that eases source-to-source translation, namely by entirely removing the need to implement or reuse a full compiler front-end to prototype additional code transformations on the input;

3. provide an *alternate code generation* path from the input language to microthreaded assembly, as a contingency plan for the missing dependency of Apple-CORE.

The first action motivated the addition of an new layer in the toolchain, the *SL compiler driver* (codenamed `slc`). This is implemented as a set of modular and extensible Unix shell scripts which invoke the actual tools based on run-time configurable toolchain parameters, and is described in section 4.1 and appendix A.

The second action motivated the creation of the *SL programming language*, whose semantics are mostly identical to μTC, entirely specified [6] but where the extensions to the C language that provide microthreading are expressed using *macro calls* in the M4 macro processing programming language [3] [9] [7] instead of a more traditional C-like grammar. This rerouting of the μTC extensions in syntax allows to implement source-to-source translation by a "simple" redefinition of M4 macros, and the SL compiler driver was designed to allow for selecting the set of expansion macros based on run-time parameters.

The third action, in essence the seminal motivation for this work, tapped on the two previous changes in strategy as follows:

- a new source-to-source translation was defined, which given SL code as input generates C code annotated with extended syntax, recognized by the standard GCC compiler, as output. This was prototyped as a set of M4 macros enhanced with some logic in the Python language [1], both of which are designed for fast prototyping; as such, the efforts could focus on researching optimal translation strategies, as opposed to integration issues that would be caused by the need to reuse a full compiler front-end such as the Microgrid core compiler's;

- this code is then provided to a "standard" compiler, namely the stock GNU C compiler configured as a standard cross-compiler for Linux/Alpha (SPARC/LEON3 is also possible) with *no change to the compiler itself*;

- while a standard compiler would generate "standard" assembly code (i.e. without microthreaded extensions), the annotations introduced in the input C source cause the compiler to propagate extra information from the source into the output assembly; this in turns allowed to design and prototype a *post-compilation assembly filter*, also using Python, which post-processes the program after code generation to transform it to valid microthreaded code;

- to make this new compilation chain transparent to the higher-level tools, *new rules* were then added to the SL compiler driver, together with a corresponding set of run-time configuration parameters.

This alternate compilation process, now entirely prototyped, is integrated side-by-side with the original core compiler process in the SL compiler driver. We intend to exercise both processes in the future for mutual testing and evaluation of compiler features.

1.3 Success, limitations and future work

This approach was initiated in November 2008 and proved successful by March 2009, where "large" SL programs could be entirely compiled and successfully run in Microgrid simulations, with relatively good execution performance.

As an expected side-effect, this prototype compiler became an invaluable tool in the evaluation and validation of the simulation environment, since compiler-genated code exercises the simulated hardware features to a far wider extent than manually crafted assembly source. Several updates in the architecture hardware design, such as multiple waits on a single I-structure register, were in turn prompted by compiler-generated "corner case" code, solved in design, and successfully implemented in simulations (cf. the separate overall progress report for Apple-CORE).

However, the following issues should be addressed, possibly by further efforts in Apple-CORE:

1. the choice to spread code generation across both the compiler front-end (in macro expansions) and the back-end has constrained the current prototype to allow only for *local optimizations.* This is because macro expansions are non-contextual to the surrounding source code, so the logic driving the expansion of each macro cannot depend on the structure of the whole program. In order to perform several desired *interprocedural optimizations* such as register file optimization, SVP block size optimization, flattening of tail recursions as single families of threads, scope analysis for memory allocations, etc., a different compiler design must be prototyped with a more potent compiler front-end including whole-program semantic analysis;

2. until this point, work on hardware design and simulation has focused on the extension of a 64-bit ISA, namely the DEC Alpha, since the extended number of bits per register provides the most promising opportunities for efficient use of multiple cores. The compiler efforts have naturally been following in that direction. However, in the context of Apple-CORE the hardware design prototyped on FPGAs is a different design, namely Gaisler's 32-bit LEON3. In order to generate code to run on this hardware prototype, the compiler needs to be adapted to produce assembly for the different instruction set.

From these two points, the first could be arguably defined as future work on the Microgrid core compiler. Indeed, the GCC core builds a full internal representation of the program, where whole-program optimizations can occur. The desired optimizations would fit naturally as additional components in this framework. However, considering the engineering efforts associated with any additions to the GCC core, it may be desirable to prototype these optimizations externally. An analysis will be conducted to determine which approach is most suitable.

The second point will be approached either in the Microgrid core compiler, the alternate compiler process, or both, in coordination with the work on the FPGA design performed by UTIA.

2 Divergence from μTC and implementation restrictions

As claimed in the previous section the new language SL *builds upon* the "original" planned μTC [2] by deriving the syntax and make it more easily transformable. As such, SL can be essentially described as a *derived* version of μTC instead of a wildly diverging replacement. This can be observed in the SL language specification [6] which refers directly to both the C and μTC specifications for most language aspects.

However, a number of implementation issues listed below that *already existed* with μTC prior to the start of Apple-CORE have since been *formalized* and integrated as "known restrictions" in the specification of SL. From the perspective of the project partners, this may appear at first sight as a new unexpected restriction from the initial announced language support; however, even without the existence of SL these points would still be causing restrictions in the initial use of μTC itself. We acknowledge these limitations as transitory implementation difficulties that will be *worked around* until they are fully resolved in the core compiler and operating system work.

The issues that have caused "known restrictions" in SL are, in decreasing order of visibility and impact:

- *support for C function calls*; this issue stems from the groundbreaking uses of the register file on the Microgrid architecture: it makes it impossible to compile, as in C, functions separately and then establish a unified calling convention for arbitrary composition between code compiled separately for heterogeneous register window layouts. The solution to this issue has been the design of a *call gate* between microthreaded code with limited register windows and the "standard" register environment used by regular C functions. The implementation of this mechanism will be finalized and integrated to the language before the end of 2009, and will thus allow immediate reuse of most of C's function libraries;

- *support for the C standard library*; this issue stems mainly from the previous point, but also from the current lack of a featureful operating system on the Microgrid. Proper support for the C standard library requires system abstractions such as file descriptors, signals, etc which have not yet fully specified in the design of the Microgrid architecture and a corresponding operating system. Gradual support for the C library will implemented as new architecture features are properly specified and implemented, at least in simulation; in particular minimal text output is already available and the following services are already planned and will be available before the end of 2009: memory allocation and math functions;

- *support for arbitrary data types for thread parameters*; this issue stems from the current lack of support in the Microgrid core compiler for data types wider than a word, and the large engineering effort required to achieve this support in the "alternate" compiler route. We plan that this support will be eventually available in the Microgrid compiler and lift the restriction; in the meantime, programs will be restricted to use and define (thread) functions whose parameters are word-sized (or smaller). This has not yet shown to be an issue in the benchmarking work for Apple-CORE.

Another two restrictions have also been integrated into the project, which stems from the choice of a macro processor to perform source-to-source SL translation, and is not a previously existing μTC implementation issue:

- as the macro processor does not perform semantic analysis of the input programs, it cannot determine the type of variables automatically; therefore, the macros whose expansion depend on data types need to be provided the type names of their operands explicitly. This prevents proper *support for implicit typing in calls and parameter reuse*, i.e. the possibility to write the name of a variable in several language contexts without specifying their type explicitly; in other words, the SL specification mandates that supplemental typing information should be provided in several contexts where they are not necessary in C. This restriction should stay as long as the source-to-source transforms in SL are implemented using a macro processor, which we have not planned to change within Apple-CORE yet. As a consequence, higher-level compilers need to propagate typing information in new locations when they generate SL code, as compared to μTC which was defined with more support for implicit typing;

- on at least one implementation of the Microgrid, namely the microthreaded Alpha simulation, the basic C integer division operator needs to be implemented by the compiler using a software trap. This is because the DEC Alpha design does not include a integer divide unit. To capture this requirement in the "alternate compiler" it was necessary to require SL code to use a syntax suitable for the M4 macro processor for integer divide, because simple uses of the C syntax (the binary "/" operator) cannot be overloaded in M4.

Despite these various restrictions, we believe that the current SL specification provides a strong and sound basis as a common intermediate language for the higher-level parallelizing compilers. Experimental testing has not revealed major obstacles in the rest of project stemming from these restrictions.

3 Validation and testing

One of the promises of the SVP model is to provide a general-purpose programming model where system features can be safely composed in programs. Such a widely scoped promise taps in the announced *orthogonal and fully composable feature set* of SVP, embodied in the Microgrid hardware architecture.

To validate this claim experimentally, the features need to be thorougly tested.

This is partly the role of performance benchmarks when the benchmarks are selected across a wide variety of application domains, such as those selected for Apple-CORE; however, as with any other compiler technology, an additional set of *unit tests* is needed to ensure that the full spectrum of low-level features are extensively tested. This is a requirement to ensure that the compiler is *trustable*, and that the compilation scheme is *sound* and *complete* across the specified range of input language semantics.

For this purpose we have worked in the following directions:

- to ensure *coverage of features*, as set of unit test programs have been implemented in the SL language exercising the various possible constructs of SVP; The suite is easily extensible and new tests will be added as new bugs are fixed in the compiler and simulation environment, for the purpose of *regression testing*;
- to address the fundamental problem of *issue discrimination* when troubleshooting a new architecture design, where new issues found during program execution must be properly separated into source program issues, compiler issues and hardware issues for further investigation, the SL compiler driver has been extended to cross-compile to other targets than the Microgrid; this is described in the following two subsections.

It should be noted that these two directions are independent. However, to ease and automate testing, the two test directions have been integrated in comprehensive and complementary command-line utilities: the *SL runner* which takes any executable file output by the SL compiler driver and runs it (transparently invoking the Microgrid simulation as needed), and the *SL unit tester* which uses the SL runner to validate unit tests across multiple implementations and reports synthetic results. These are described in sections 4.2 and 4.3 and appendices B and C.

Thanks to this double approach, *the implementation of benchmark programs could be successfully decoupled from the compiler efforts*, and both compilation bugs and architecture design issues can be rapidly isolated and addressed without impacting the benchmarking efforts.

3.1 Sequential execution

One of the landmarks of the SVP programming model is that any deterministic program using SVP primitives has a representation in a purely sequential programming language, namely using loops to execute families of threads.

As an illustration of this property, a translation from SL to plain sequential C has been implemented, and the SL compiler driver has been extended with appropriate rules to compile SL programs using this translation and the standard C compiler.

This “extra” work had negligible implementation costs, mainly due to the flexibility of using M4 as a source preprocessor. The net result of this work is threefold:

- since this approach runs programs deterministically with a static scheduling, it can be used to provide *reference observable outputs* for any deterministic programs;
- as described above, when run next to the Microgrid implementation it can help classify new issues into either source program issues (when the issue shows both in the sequential execution and the Microgrid) or environmental issues (when the issue shows only on the Microgrid);

- as a strategy for resource management on the Microgrid, discussion is ongoing to perform *dynamic automatic sequentialization* of microthreaded code when concurrent resources become sparse. This may involve the run-time selection and execution of alternate sequential code for thread functions; the sequential compilation described here could be used to prototype this strategy in further stages of Apple-CORE or in subsequent research projects.

3.2 Integration with the high-level simulation

One of the outputs of the related AETHER project is a high-level SVP simulation using POSIX threads. This simulation is based on the implementation of SVP primitives as special constructs in the C++ language, which encapsulate calls to POSIX thread management functions, and which are compiled along with SVP programs using a C++ compiler to produce simulation executables. The software is provided as a *C++ library* named μTC-ptl, and it therefore expects SVP programs to use a C++ syntax.

In order to tap into the high-level simulation opportunity represented by μTC-ptl in Apple-CORE, a translation from SL to C++ with μTC-ptl calls has been implemented, and the SL compiler driver has been extended with appropriates rules to compile SL programs using this translation and the standard C++ compiler.

This "extra" work also had negligible implementation costs, again due to the flexibility of using M4 as a source preprocessor. The net result of this work is that *any valid SL program can be retargeted to the high-level simulation with no changes.* This provides immediately an additional validation platform for deterministic programs.

4 Description of the toolchain

The SL toolchain exposes the following items to users and higher-level compilers and testing engines:

- `slc`, the SL compiler driver, which transparently handles the compilation of SL code to various target architectures using various compilation processes;
- `slr`, the SL runner, which transparently handles the execution of a binary executable generated by `slc`;
- `slt`, the SL unit tester, which repeatedly compiles and runs a single SL program across a range of compile and run-time parameters.

As described in the previous sections, the SL compiler driver is abstracting from multiple compilation processes. Figure 1 illustrates the four processes currently supported and used during testing, and the corresponding selection performed by the SL runner.

4.1 Compiler architecture

4.1.1 Abstract compilation process

The SL compiler driver is based on an abstraction of the compilation process into the following generic phases:

1. *source-to-source translation*: the SL source code is transformed into a *SL low-level dialect* using the M4 processor;
2. *compilation*: the low-level dialect is compiled into *raw assembly* source by a *code generator*;
3. *post-compilation filtering*: the raw assembly is *refined* to produce the actual assembly suitable for the target architecture;
4. *assembly*: the filtered assembly is transformed into *object code*;

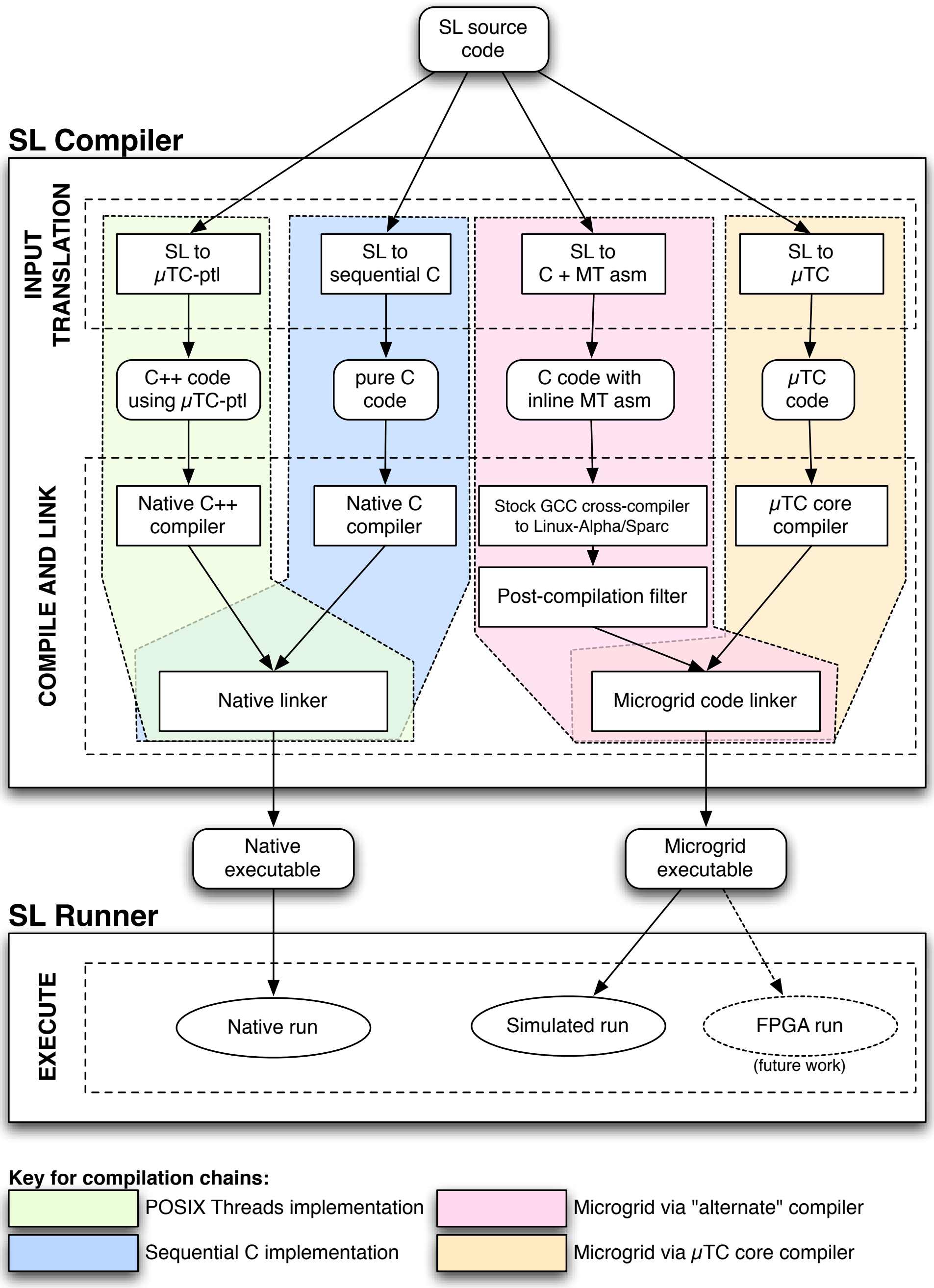


Figure 1: Overview of the SL compiler and runner

5. *linking*: objects are linked with the necessary library dependencies into an *executable file.*

This approach is similar to the approach taken by traditional compilers, where multiple source languages are transformed into a common intermediate representation and then compiled, assembled and linked. However our approach decouples input translation and post-compilation assembly filtering from code generation to allow for rapid prototyping of alternate compilation routes, as described at the beginning of this report.

4.1.2 Parameterized tool selection

The abstract compilation process is implemented as a single generic program in the SL compiler driver itself (`slc`). This generic program is *parameterized* by the actual tools to be used to perform the individual phases. By selecting different set of parameters, back-end tools can easily be recomposed in different ways to experiment with alternate compilation processes. From a functional perspective, the SL compiler driver can be described as a meta-function which, given a set of tool parameters, produces a compiler which in turn is a function from source code to compiled code.

The parameter signature allow for the separate selection of:

- the *set of M4 macro definitions* for source-to-source translation;
- the *code compiler* for actual compilation;
- the *post-compilation filter* for assembly refinement;
- the *binary utilities* for assembly and linking of objects, together with the set of standard *libraries* for the target architecture;

as well as the full range of configuration options for each of the individual tools.

To make the large resulting parameter space amenable to testing, and since the parameters are not truly independent, a helper utility has been designed and implemented: the *Configuration Chain Editor* (`cce`). This is invoked transparently by `slc` and abstracts the tool selection for `slc` to yet another level by providing an automated translation from a simplified *configuration tag* to a selection of tools.

A configuration tag is a concatenation of the following:

- a *dialect tag*, shorthand for a source-to-source translation; examples include `seqc` (sequential C) and `ptl` (C++ with μTC-ptl);
- an *architecture tag*, shorthand for an instruction set architecture and binary executable format; examples include `mtalpha` (Microthreaded Alpha) and `host` (native ISA to the host system where the compiler is run);
- an *environment tag*, shorthand for the execution environment where programs are run, possibly including bits about the operating system; examples include `sim` (simulation) and `host-seqc` (host system with entry point into the program suitable for the sequential C scheduling of SVP).

The `cce` utility processes a configuration tags and performs the following:

- expand configuration tags into a full set of tool parameters for `slc`;
- associate shorthand “aliases” for tool selections; this reflects the concept that certain compilation processes will be used more frequently and need to be identified using a simplified mnemonic. For example the full compilation process that flattens a deterministic SVP program to a purely sequential program and compiles it natively is identified by the single word “`seqc`”;

- validate selection of tools provided by the user or higher-level tools actually match the internal constraints of the system of parameters. For example a selection for a source-to-source translation which produces C++ code should not be associated with a compiler that takes plain μTC as input; this is verified and reported descriptively by `cce`;
- select defaults values for configuration parameters that are not considered for systematic parameter space exploration during tests. For example during linking each SL program should be linked with object code for the SL library services, much like standard C programs need to be linked with the C library; in the case of SL this implies selecting for each SL target architecture the right object code for the SL library.

Figure 2 illustrates the interaction between `slc` and `cce`. Note that the invocation of `cce` from `slc` is automated and thus not visible to users and higher-level tools.

As another illustration, the processes described in Figure 1 are identified by the following full configuration tags and their aliases:

Compilation process	Configuration tag	Alias
Sequential C	`seqc-host-host-seqc`	`seqc`
μTC-ptl	`ptl-host-host-ptl`	`ptl`
Alpha MGSim using MG core compiler	`utc0-mtalpha-sim`	`utc0`
Alpha MGSim using alt. compiler	`ppp-mtalpha-sim`	`ppp`

By encapsulating the tool selection from the actual execution of the compilation chain, true *separation of concerns* is effectively achieved and ensures easy future extensibility and maintainability of the tools as new compilation strategies are devised and prototyped.

4.1.3 Compiler interface

The interface provided to the higher-level compilers was chosen to be nearly identical to GCC's compiler driver interface (the `gcc` utility itself). This means that `slc` appears as a Unix shell utility, can be used on the Unix command line with similar options as `gcc` and reports processing status and messages with a format similar to `gcc`. This is believed to ease integration with other development tools such as automated build utilities (e.g. `make`) and allows to reuse the existing compiler's documentation [8].

The addition of new features not present in traditional compilers has in turn justified the introduction of new command-line parameters. These are extensively described with examples, in the Unix tradition, as a Unix manual page. A copy is included in appendix A.

4.1.4 Library support

Despite the restrictions in supporting the C library directly in SL as described in section 2, there is support for the concept of a "standard library" in SL as it exists in C. These are a set of standard services made usable to programs, using the same software protocol as in C:

- *header files* are included by programs and declare system services; this way the *names* (symbols) of said services are exposed to program code and create a transparent external dependency from programs to the separately defined implementation of the services;
- *libraries of precompiled objects* used by the link editor at the end of the compilation process, where the actual implementation of the services used by programs are found and collected to append to the eventual executable files.

The current distinction between SL and μTC, and by extension C, is the *set of services* currently available through this mechanism to SL programs. At this point only the following services are defined and implemented in the SL library:

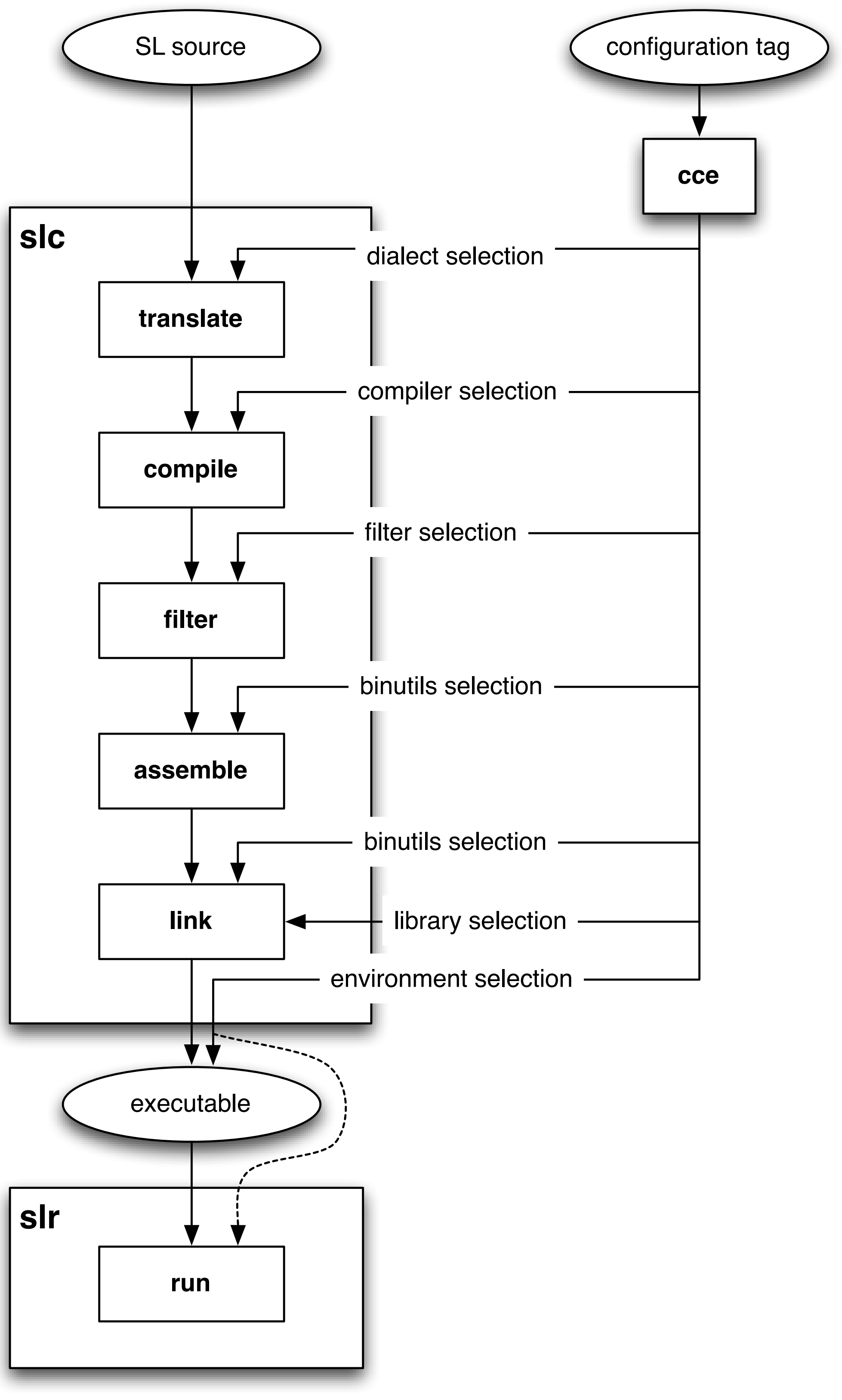


Figure 2: Integration between the SL compiler, SL runner and Configuration chain editor

- debugging utilities (assertions, breakpoints);
- text output from programs (to a simple character-based text interface);
- data types, mathematical constants and language limits;
- data input for programs from the SL runner;
- performance counters (access to the cycle counter of the underlying CPU).

The following have been designed and prototyped, but not yet fully tested at the time of this writing:

- global memory allocator from heap memory;
- preliminary support for math functions.

4.2 SL runner

The work that resulted in the design of the SL runner stemmed from two requirements:

- *Encapsulation of the execution environment.* The SL compiler's support for both the Microgrid architecture and legacy systems as targets creates an asymmetry in the way resulting binaries should be executed: programs compiled for the host environment become native executables, whereas programs compiled for the Microgrid architecture should be either run by a simulation or sent to a FPGA board. To simplify the automation of testing and the experimental framework, a *unified interface to run compiled SL programs* was necessary.
- *Data input for programs.* At least in the initial phase of Apple-CORE, the Microgrid systems (either simulated or FPGA) where programs are to be run will not be interactive, most especially will lack a programmable run-time input channel with their environment. This is because work to design and implement such input channels with their software support is part of Apple-CORE in later milestones. This situation raises the problem of program input: many SL programs, especially benchmarks, that are to be tested during the project allow to set input data and parameters, and the exploration of the parameter space is a large part of the experimental benchmarking process. To simplify the automation of testing by avoiding lengthy build-test-execute cycles where benchmarks are recompiled for each set of input data and parameters, some system allowing for *run-time data input* is required.

The SL runner was designed and implemented to answer these two requirements as follows:

- given any compiled SL program, the corresponding binary executable file can be run from the Unix command line by *prefixing* it with "`slr`", the name of the SL runner utility. This causes the whole command line for the SL program to be provided as input to the SL runner, which then selects the runtime environment to use by examining the content executable file, where the configuration tag (see section 4.1.2) was embedded textually by the SL compiler during compilation. This is illustrated in Figure 2;
- for data input, a software protocol was designed to allow SL programs to interact with the SL runner:
  - in the program source code, an arbitrary number of *typed and named input variables* can be declared using a special syntax, with the expectation that run-time values will be loaded into them at the point a program starts. For added scalability and to suit the requirement of benchmarks that deal with e.g. signal processing, each of these variables allows for input in the form of an *array of values* of arbitrary size;

– at run-time, the SL runner extracts from the executable file the name and data type of the declared input variables, and then handles the command line used to invoke the program accordingly. Namely, the values provided on the command line (or indirectly loaded from a file if a filename is specified) are converted to binary data matching the declared data types, and then loaded at a known location in memory together with the program code before execution starts.

This mechanism is more fully documented in [6].

The command-line interface of the SL runner is also documented as a Unix manual page, a copy of which is included in appendix B.

4.3 SL unit tester

As described in the beginning of this report, testing of the compiler can occur both across the feature set (coverage) and across multiple target SVP implementation (portability). The automation of both under a simplifed testing interface is required to ensure that running unit tests and checking results has a low overhead and can be transparently integrated in a fast development cycle.

As an additional requirement stemming from the very nature of unit testing, some *automated identification of the origin of issues*, when any appear, should be provided as well. For this purpose unit testing for compilers is traditionally decomposed as follows:

1. compile the test program, and check that the compilation succeeds; if a problem occurs, report at *which phase of the compilation* the problem first appeared;
2. run the test program, possibly using *multiple sets of input parameters*, and check that the execution succeeds; if a problem occurs, report *which kind of execution failure* occurred;
3. compare the program output against a reference output (to validate the operational semantics of deterministic programs); if a difference occurs, report *where in the program the divergence first occurred.*

The SL unit tester provides a single utility named `slt` which performs these three steps of unit testing *across multiple compilation processes and run environments* supported by the SL compiler driver and runner. For this purpose `slt` takes as input parameters:

- the SL source code for the unit test program itself,
- the list of input data sets to use for multiple test runs,
- the list of configuration tags and extra compilation and runtime settings across which unit testing for this program should occur,

and performs the test cycle by repeatedly invoking `slc` and `slr` and collecting output and exit codes.

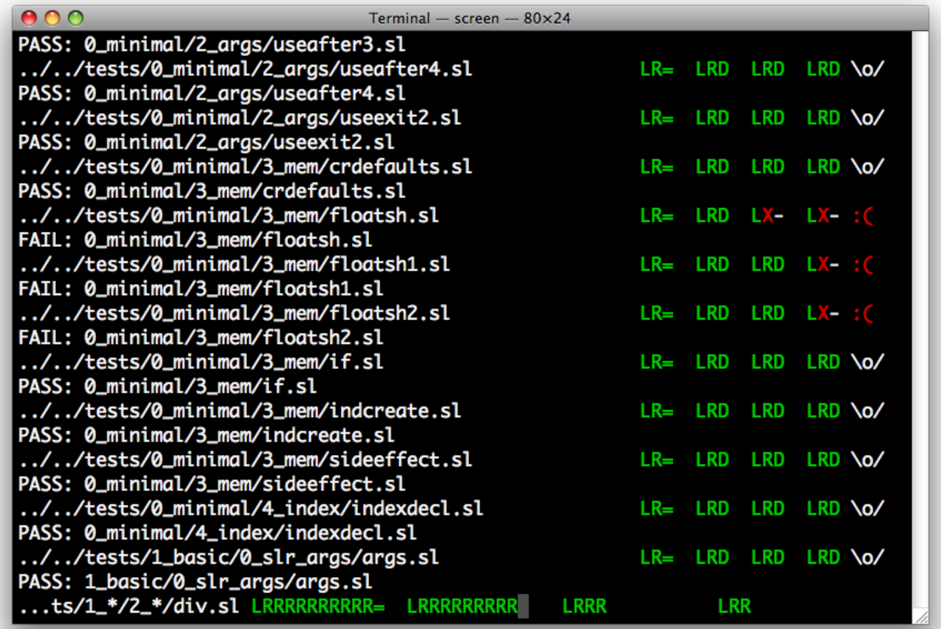
From this point, given a collection of N unit test programs and a selection of M compile and run environments, with an average of P input data sets per run per program, the validation of the entire test suite requires a total of $N \times M \times P$ unit test cycles. The potentially large test space creates two additional requirements for unit testing in this context: some form of *test parallelism* to reduce the total time required to run the full test suite and the availability of a *sumarized output for test results*, without preventing detailed analysis of issues where they are detected.

To respond to these additional requirements, the following approach was taken:

- to capture all the test results for a single test program in a condensed form, the `slt` utility uses one-character result codes combined with colorized output to fit all test results on a single line of terminal text;

- to reduce test times, the `slt` utility runs the test cycle for all configurations concurrently, as different Unix processes. This ensures that multiple cores in the testing environment are efficiently used;
- to automate unit testing across the full range of unit test programs, the built-in test process from GNU Automake [5] [10] has been reused. This allows the automated invokation of a command (here `slt`) across a range of input source files, and the automated collection of test results with a final summary.

An illustration run of `slt` by GNU Automake is presented in Figure 3.



```
PASS: 0_minimal/2_args/useafter3.sl
../../tests/0_minimal/2_args/useafter4.sl              LR=  LRD  LRD  LRD \o/
PASS: 0_minimal/2_args/useafter4.sl
../../tests/0_minimal/2_args/useexit2.sl               LR=  LRD  LRD  LRD \o/
PASS: 0_minimal/2_args/useexit2.sl
../../tests/0_minimal/3_mem/crdefaults.sl              LR=  LRD  LRD  LRD \o/
PASS: 0_minimal/3_mem/crdefaults.sl
../../tests/0_minimal/3_mem/floatsh.sl                 LR=  LRD  LX-  LX- :(
FAIL: 0_minimal/3_mem/floatsh.sl
../../tests/0_minimal/3_mem/floatsh1.sl                LR=  LRD  LRD  LX- :(
FAIL: 0_minimal/3_mem/floatsh1.sl
../../tests/0_minimal/3_mem/floatsh2.sl                LR=  LRD  LRD  LX- :(
FAIL: 0_minimal/3_mem/floatsh2.sl
../../tests/0_minimal/3_mem/if.sl                      LR=  LRD  LRD  LRD \o/
PASS: 0_minimal/3_mem/if.sl
../../tests/0_minimal/3_mem/indcreate.sl               LR=  LRD  LRD  LRD \o/
PASS: 0_minimal/3_mem/indcreate.sl
../../tests/0_minimal/3_mem/sideeffect.sl              LR=  LRD  LRD  LRD \o/
PASS: 0_minimal/3_mem/sideeffect.sl
../../tests/0_minimal/4_index/indexdecl.sl             LR=  LRD  LRD  LRD \o/
PASS: 0_minimal/4_index/indexdecl.sl
../../tests/1_basic/0_slr_args/args.sl                 LR=  LRD  LRD  LRD \o/
PASS: 1_basic/0_slr_args/args.sl
...ts/1_*/2_*/div.sl LRRRRRRRRRR=  LRRRRRRRRR   LRRR          LRR
```

Figure 3: Exceirpt from an example SL unit test suite run by GNU Automake

The format of the result output, as well as all the command-line options of the SL unit tester, are also documented as a Unix manual page. A copy is included in appendix C.

5 Packaging and distribution

During the realization of this work extra management constraints have guided the engineering efforts in additional ways not covered in the previous sections:

- *portability*: the development efforts from the various Apple-CORE partners occur on a variety of host systems. The SL toolchain must therefore be usable on the full range of platforms used in Apple-CORE, namely Mac OS X, FreeBSD, CentOS and Ubuntu Linux;
- *maintainability of installations*: during compiler developments several versions of the toolchain will be disseminated among partners. It is necessary that upgrades can be automated and require minimal manual intervention; it must be also possible to let multiple versions of the toolchain co-exist for regression testing.

To answer these requirements, the SL toolchain has been integrated using GNU Autoconf [4] and Automake. GNU Autoconf allowed to express a single implementation without the need to check for environmental difference between user systems; GNU Automake allowed to simplify the description of the build and test system.

The following points can be highlighted as items of interest when using the SL toolchain:

- the utilities themselves (`slc`, `cce`, `slr`, `slt`);
- a small colletion of medium-length demonstration SL programs to show various SL/SVP features;
- the compiler unit test suite as a categorized collection of small SL programs;
- automated build rules for rebuilds, installation, uninstallation, testing, and the creation of new distribution archives after changes (this is provided by GNU Automake directly);
- the Unix manual pages for each utility;
- a software overview to serve as introduction for first-time users (`README`);
- the list of principal user-visible changes to the toolchain over time (`NEWS`);
- a developer's manual for maintainers of the SL toolchain (`HACKING`).

5.1 Integration to Unibench

During the benchmarking work of Apple-CORE the Unibench system developed at UH will be used. This system automates the compilation of benchmark programs, the persistence of experimental setups over time (for reproducibility of results) and offers an integrated approach to parameter space exploration.

The automated process in Unibench is parameterized as follows:

- a combination of a *compiler environment* and *library environment* able to generate code;
- an *execution environment* which drives the compilation process;
- *observation scripts* that drive the execution of programs.

To integrate the SL toolchain into Unibench the following are defined:

- a mapping from the *tool requirements* of the SL compiler driver (the actual back-end tools used to perform SL compilation) to the Unibench library environment. As the set of back-end tool to the SL compiler driver is stable we expect a small number of library environments configured in Unibench;
- a mapping for the SL toolchain itself to the Unibench compiler environment. This will cause as many versions to be configured in Unibench as there will be new versions of the SL toolchain distributed to UH;
- a mapping of the various compilation processes supported in the SL compiler driver to the Unibench execution environments;
- a strategy to run benchmark programs by invoking the SL runner repeatedly inside a Unibench observation script.

These mappings are expected to be refined as more benchmark code is added to the benchmark repository.

6 Summary of efforts

The design and implementation costs during the period can be summarized as follows:

Component	% efforts expensed (approx.)
Autoconf/Automake integration	5%
Translation to μTC-ptl and sequential C	5%
Documentation and reporting	5%
SL runner	5%
SL compiler driver	10%
SL unit tester	10%
Alternate Microgrid compiler	20%
Debugging and testing	40%

APPENDIX A - Manual page for the SL compiler

This appendix is the Unix manual page for slc 2.2073.

A.1 Synopsis

slc [*ARG*]...

A.2 Description

SVP Language Compiler.

'slc' is a generic compiler driver for SVP programs. It provides a unified command line interface for compiling SVP code. The actual tools used for compilations are selected automatically depending on the selected SVP implementation, using the separate utility 'cce'.

The following transforms are applied depending on the type of the input (determined by the file name suffix):

- translate: the SVP language is translated to an actual SVP dialect. This transform uses the M4 preprocessor.
- preprocess: the source code is filtered by the C/C++ preprocessor.
- compile: the source code is transformed to raw assembly; the compiler used depends on the SVP dialect and the target architecture.
- filter: (for the Microgrids implementations) the raw assembly is further distilled.
- assemble; the assembler used depends on the target architecture.
- link; the linker used depends on the target architecture.

Except for the two additional transforms (translate and filter), 'slc' mimics the driver of the GNU Compiler Collection ('gcc'). For general information about default behavior, additional flags and processing order, see `gcc(1)`.

A.3 Options

-check Check the environment; do not process anything. In this mode slc checks that the required tools are available.

-EE Stop after the translation stage; do not preprocess. The output is in the form of translated source code, which is sent to the standard output.

-E Stop after the preprocessing stage; do not compile. The output is in the form of preprocessed source code, which is sent to the standard output.

-S0 Stop after the first stage of compilation proper; do not perform post-compilation filtering on assembly. The output is in the form of an unprocessed assembler code file for each non-assembler input file specified.

-S Stop after the second stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified.

-c Compile or assemble the source files, but do not link. The linking stage is not done. The output is in the form of an object file for each source file.

A.3.1 SVP implementation selection

-b spec Generate code for the SVP implementation specified by "spec". "spec" is passed through 'cce' for validation and alias expansion. See '**cce –help**' for details. The default implementation is "seqc".

A.3.2 Input file types

file.sl SVP code which must be translated.

file.utc muTC source code which must be preprocessed.

file.iu muTC source code which should not be preprocessed.

file.s0 Assembler code which should be further distilled.

The regular GCC file types are also recognized and are transformed accordingly.

A.3.3 General options

-Idir Add the directory "dir" to the list of directories to be searched for header files. This applies both to the initial translation by M4 and to the underlying compiler.

-P Inhibit generation of linemarkers in the output from both the initial translation by M4 and the C preprocessor.

-combine=*[PHASES*] Group input files into a single output for each of the specified PHASES.

-combine Synonymous with **-combine**=*C* (behavior similar to 'gcc').

-o file Place output in "file". This implies **-combine** for the phase where 'slc' is configured to stop.

A.3.4 Compiler debugging options

-v Print (on standard error output) the commands executed to run the stages of compilation. Also print the settings and version number of the tools involved.

-ftrace-m4=*file* Dump the m4 macro expansion trace to file "file".

-save-temps Keep intermediate files. If this option is not set, intermediate files generated during compilation are deleted before the program terminates.

-t dir Output directory for intermediate files. If this option is not set, a directory is automatically created in the system-wide default temporary directory (usually `/tmp`).

A.3.5 Other options

-h, **–help** Print this help; do not process anything.

–version Print slc's version number; do not process anything.

All other options are passed transparently to the appropriate tools; in particular, the following options may prove useful:

-Dmacro, **-Umacro** Set/unset C/C++ preprocessor macros.

-Wa,option Pass "option" as an option to the assembler.

-Wl,option Pass "option" as an option to the linker.

-Ldir Search ”dir” for libraries when linking.

-llibrary Link with the library named ”library”.

-nostartfiles Do not use the standard startup files when linking.

-nodefaultlibs Do not use the standard system library when linking.

-nostdlib Implies options **-nostartfiles** and **-nodefaultlibs**.

A.3.6 Environment variables

M4 M4 preprocessor to use. Default is ”m4”.

KEEP If set and not empty, implies option **-save-temps**.

A.3.7 Diagnostics

0 No error occured.

1 Invalid parameters passed to ‘slc’ or invalid configuration.

2 Translation failed.

3 Preprocessing failed.

4 Compilation failed.

5 Post-compilation filtering failed.

6 Assembly failed.

7 Linking failed.

127 A tool was not found.

126 A program could not be executed.

A.4 Examples

```
# compile "test.sl" and produce "a.out"
slc test.sl

# compile "test1.sl" and "test2.sl" together and produce "test"
slc -o test test1.sl test2.sl

# translate "test.sl" on standard output
slc -EE test.sl

# generate assembly code for "test.sl"
slc -S -o test.s test.sl

# compile "test.sl" and "ccode.c" together and produce "test"
slc -o test test.sl ccode.c
```

APPENDIX B - Manual page for the SL runner

This appendix is the Unix manual page for slr 2.2073.

B.1 Synopsis

slr [*OPTION*] *PROGRAM* [*ARGS*]...

B.2 Description

SL Runner.

The ‘slr’ utility provides a single interface to run a SVP program compiled with ‘slc’. ‘slr’ analyses the executable file to determine how to run the program, and invokes the simulator if needed.

This uses the SL input mechanism described in CSA note [sl3].

B.3 Options

-dVAR Define the program input VAR as an empty array.

-dVAR=..., VAR=... Define the program input VAR as specified (see CSA note [sl3] for details).

-l List variables required as input by the program.

-c Show configuration string embedded in program, if any.

-r runner Use the specified runner (use when autodetect fails).

-Ws,arg Pass "arg" as extra command line parameter to the simulator, when the simulator is used.

-x Trace: print commands before running them.

-g Debug: try to run debugger around program. Implies **-x**.

-h, **–help** Print this help, then exit.

-v, **–version** Print version number, then exit.

B.3.1 Environment variables

VERBOSE If set and not empty, implies option **-x**.

DEBUG If set and not empty, implies option **-g**.

DEBUGGER Command prefix to use as debugger. Default is ‘gdb **–args**’.

SIMARGS Combines with the occurrences of option **-Ws** on the command line when running the simulator.

B.4 Examples

```
# shows runner string stored in ‘a.out’:
slr -c a.out

# runs fibo.x, possibly using the MT simulator, and set program input
# variable N to an array of 1 element with value "15".

slr fibo.x -dN=15

# Runs fibo.x, possibly using the MT simulator, and set program input
# variable N to an array of 1 element with value "10". If the simulator
#  is used, add "-o NumProcessors=7" to the simulator command line.
slr fibo.x -dN=10 -Ws,-o -Ws,NumProcessors=7
```

APPENDIX C - Manual page for the SL unit tester

This appendix is the Unix manual page for slt 2.2073

C.1 Synopsis

slt [*SOURCE*] [*IMPLEMENTATION*]...

C.2 Description

SL Unit test and portability checker.

'slt' is a generic tester script for SL programs. It tries to compile and execute the program given as input on multiple SVP implementations, comparing results.

A test is considered to succeed when the following conditions are met:

- the program compiles and links successfully on all specified SVP implementations, or it fails to compile on all implementations and it is marked as not compilable (see below);
- the program runs successfully on all specified SVP implementations, or the program fails to run on all implementations and it is marked as not runnable (see below);
- on all runs but the first, the console output of the program is identical to the output on the first run.

C.2.1 Output

The program performs the tests and display results as they become available using the following format:

```
SOURCE XXYZ XXYZ XXYZ ...
```

The first column contains the name of the SOURCE specified on the command line, possibly truncated to fit the results on the rest of the line. Each subsequent column contains a string of tests results for the corresponding implementation specifier on the command line.

Each string of test results has 3 sections:

- results for compilation;
- results for program execution;
- result for output comparison.]

C.2.2 Output codes for compilation

The compilation results column can contain any of the following:

L	An executable was produced successfully.
8	A timeout occurred during compilation.
!	A signal occurred during compilation.
?	Compilation succeeded, but a failure was expected.
XT	Translation failed.
XP	Preprocessing failed.
XC	Compilation failed.

XF	Post-compilation failed.
XA	Assembly failed.
XL	Linking failed.
/T	Translation failed, as expected.
/P	Preprocessing failed, as expected.
/C	Compilation failed, as expected.
/F	Post-compilation failed, as expected.
/A	Assembly failed, as expected.
/L	Linking failed, as expected.

C.2.3 Output codes for execution

The execution results column(s) can contain a repetition of the following:

R	Execution succeded without errors.
8	A timeout occurred.
!	A signal occurred.
?	Execution succeeded, but a failure was expected.
o	A timeout occurred, as expected.
.	A signal occurred, as expected.
-	The program was not run.

C.2.4 Output codes for output comparison

The output comparison results can contain any of the following:

=	The output will be reused as reference.
D	The output is identical to the reference.
X	The output is different from the reference.
-	The output was not checked.

C.2.5 Output comparison

For program output comparison, the test assumes that the first implementation tested produces the reference output; subsequent implementations are checked against the results of the first run.

C.2.6 Temporary files

During testing a number of intermediate files are generated. If the SOURCE file is named test.sl, the following files are generated:

IMPL/test.log Test log

IMPL/test.{c,cc,utc} Source code after transformation

IMPL/test.{i,ii,ui} Source code after preprocessing

IMPL/test.s0 Raw assembly output

IMPL/test.s Assembly output

IMPL/test.o Object file

IMPL/test.x Executable to be used as input to 'slr'

IMPL/test.out Output during execution

If a test is successful the corresponding files are erased after the test completes, unless the environment variable KEEP is set and non-empty.

C.3 Options

-h, **–help** Print this help.

-v, **–version** Print a version number.

Multiple implementations can be specified, and each multiple times. If no implementation is specified on the command line, the environment variable SLT_IMPL_LIST is used.

Implementations names are passed to 'slc' as-is via the **-b** command-line parameter. See '**slc –help**' for details.

C.3.1 Configuring expected failures for tests

To indicate expected failures in specitic tests, the test source code can be annotated as follows:

XFAIL: pattern... Indicate that an error or signal is expected at the implementation/phase(s) matched by the patterns (see below)

XIGNORE: pattern... Indicate that any result at the implementation/ phase(s) matched by the patterns should be considered a success.

XTIMEOUT: pattern... Indicate that a timeout is expected at runtime in the implementation(s) matched by the patterns.

The annotation can appear anywhere; for example in source comments.

Patterns have the following form: IMPL:PHASESPEC. IMPL is a pattern that matches the implementation name being tested; PHASESPEC matches the phase(s) being tested. The pattern matches when at a test point both the current implementation and the current phase match the pattern.

Example pattern combinations:

- `*x:*` Match any phase in any implementation ending with ”x”.
- `ptl*:D` Ignore the program output for implementations whose name start with ”ptl”.

C.3.2 Test-specific runtime arguments

To run the program the helper script 'slr' is used. By default, a single run is performed without extra arguments to 'slr'. When extra arguments are required, the program should contains annotations of the following form, one per (commented) line:

```
# SLT_RUN: args...
```

For each occurrence of SLT_RUN in the program source code, 'slr' is run once with the arguments specified after SLT_RUN. There are as many "execution" columns in the output as there are SLT_RUN specifiers in the program. For example:

```
# SLT_RUN: -dN=5 -Ws,-o -Ws,NumProcessors=1
# SLT_RUN: -dN=10
```

C.3.3 Implementation-specific extra compilation and execution arguments

It may be desirable to test the same implementation multiple times for all tests, with a range of compile-time and/or runtime arguments. For example to test multiple optimization levels in the compiler. For this purpose each implementation alias specified on the command line or the SLT_IMPL_LIST environment variable can be specified as a group of three components separated by colons:

```
 IMPL:CFLAGS:RFLAGS
```

Where CFLAGS is a list of extra command-line parameters that are passed to slr'. To specify multiple arguments separated by spaces, replaces occurrences of spaces by the tilde character ($\tilde{}$).

For example:

```
slt test.sl seqc ppp:-O3 ppp::-Ws,-o~-Ws,NumProcessors=1
```

Indicates that three implementations should be tested; the second with **-O3** added to the compilation command line, the third with "**-Ws,-o -Ws,NumProcessors=1**" added to the execution command line.

C.3.4 Environment variables

SLC Location of the SL compiler. Default is "slc" in the same directory as 'slt'.

SLR Location of the SL runner. Default is "slr" in the same directory as 'slt'.

KEEP If set, do not erase intermediary files when a test succeeds. Default is not set.

TEXT_ONLY If set and not empty, use plain text messages instead of colored/positioned characters on the console.

XIGNORE If set and not empty, prepend to the XIGNORE pattern list set by programs. Can be used to mark specific implementations as not relevant for overall test results. Default is "XIGNORE=*x:*"

SLT_IMPL_LIST Used when no implementation names are supplied on the command line.

C.3.5 Diagnostics

0 The test succeeded.

1 The test failed.

2 Invalid test parameters.

C.4 Examples

```
for i in *.sl; do slt $i seqc ptl utc0; done
10threads.sl      LR=  LRD  LX- :(
answer.sl         LR=  LRD  LX- :(
empty.sl          LR=  LRD  LRD \o/
fibonacci.sl   LRR=  LRRD XC--- :(
helloworld.sl     LR=  LRD  LRX :(
quicksort.sl      LX-  LX- XP-- :(
roman.sl          LR=  LRD  LX- :(
```

References

[1] Python Software Foundation. What is Python? Executive Summary. `http://www.python.org/doc/essays/blurb/`.

[2] Chris R. Jesshope. mutc - an intermediate language for programming chip multiprocessors. In *Asia-Pacific Computer Systems Architecture Conference*, pages 147–160, 2006.

[3] Brian W. Kernighan and Dennis M. Ritchie. The M4 macro processor. Technical report, Bell Laboratories, 1977.

[4] David MacKenzie, Ben Elliston, and Akim Demaille. Autoconf: Creating automatic configuration scripts. `http://www.gnu.org/software/autoconf/manual/autoconf.pdf`.

[5] David MacKenzie, Tom Tromey, and Alexandre Duret-Lutz. GNU Automake. `http://www.gnu.org/software/automake/manual/automake.pdf`.

[6] Raphael Poss, Mike Lankamp, Thomas Bernard, and C.R. Jesshope. SL language reference. `https://mac-chris.science.uva.nl/csa/notes/www/book5.pdf`.

[7] René Seindal, François Pinard, Gary V. Vaughan, and Eric Blake. GNU M4, version 1.4.13: A powerful macro processor. `http://www.gnu.org/software/m4/manual/m4.pdf`.

[8] Richard M. Stallman and the GCC Developer Community. Using the GNU Compiler Collection, for GCC version 4.3.3. `http://gcc.gnu.org/onlinedocs/gcc-4.3.3/gcc.pdf`.

[9] Kenneth J. Turner. Exploiting the m4 macro language. Technical Report CSM-126, Department of Computing Science and Mathematics, University of Stirling, 1994.

[10] Gary V. Vaughan, Ben Elliston, Tom Tromey, and Ian Lance Taylor. *GNU Autoconf, Automake and Libtool*. Sams Publishing, 2000.